Fakultät für Elektrotechnik und Informationstechnik
Technische Universität München

# Near-Memory Acceleration of Inter-Process Communication on Tile-Based Many-Core Architectures

**Sven Rheindt**

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

**Doktors der Ingenieurwissenschaften (Dr.-Ing.)**

genehmigten Dissertation.

**Vorsitzender:**
Prof. Dr. rer. nat. Franz Kreupl

**Prüfende der Dissertation:**
1. Prof. Dr. sc.techn. Andreas Herkersdorf
2. Prof. Dr.-Ing. Dr.h.c. Jürgen Becker,
   Karlsruher Institut für Technologie (KIT)

Die Dissertation wurde am 13.07.2021 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 26.02.2022 angenommen.

# Danksagung

Besonderer Dank gilt zuallererst meinem Doktorvater Prof. Dr. Andreas Herkersdorf für seine hervorragende Anleitung und Betreuung über den gesamten Prozess dieser Doktorarbeit. Während meiner wissenschaftlichen Forschungstätigkeit zu diesem spannenden und zukunftsrelevanten Thema konnte ich stets von seiner ausgezeichneten fachlichen Expertise profitieren und wurde durch viele seiner wertvollen Anregungen und Rückfragen vorangebracht. Für die zwischenmenschlich äußerst angenehme, von hoher Wertschätzung geprägte sowie sehr produktive Zusammenarbeit bin ich ausgesprochen dankbar. Durch die Möglichkeit im DFG SFB Invasive Computing [1] zu forschen, hat Prof. Dr. Herkersdorf es mir ermöglicht, ein breites wissenschaftliches Spektrum kennenzulernen und interdisziplinäre Erfahrungen zu sammeln. Er hat ein Umfeld geschaffen, in dem ich kreativ, produktiv und mit Begeisterung arbeiten konnte. Zudem bleiben mir die gemeinsamen Teilnahmen an Fachkonferenzen, wie beispielsweise das MPSoC Forum in den Bergen von Salt Lake City, sowie die Lehrtätigkeit und Zeit in Singapur in besonders positiver Erinnerung.

Ganz herzlich möchte ich mich auch bei Prof. Dr. Dr.h.c. Jürgen Becker als Zweitprüfer der vorliegenden Arbeit bedanken. Den Austausch und die Zusammenarbeit im Invasive Computing SFB empfand ich stets als äußerst bereichernd und bedanke mich für die gelungene Kooperation. Des Weiteren gilt mein besonderer Dank auch Prof. Dr. Franz Kreupl für den Vorsitz der Prüfungskommission.

Großen Dank möchte ich außerdem Dr. Thomas Wild aussprechen, der mir stets mit Rat und Tat zur Seite gestanden hat. Besonders schätze ich die fachlich sowie zwischenmenschlich exzellente Zusammenarbeit. Ich bedanke mich ausdrücklich für das oftmals detaillierte und sehr hilfreiche Feedback zu meinen Forschungstätigkeiten und Publikationen. Zudem bedanke ich mich für das große Vertrauen, das mir entgegengebracht wurde, indem ich bereits zu Beginn meiner Promotion viel Verantwortung für den Hardware-Demonstrator im Invasive Computing SFB übernehmen durfte.

Des Weiteren möchte ich mich bei Prof. Dr. Walter Stechele für die Betreuung, während meiner Bachelor- und Masterarbeit bedanken. Bereits während meines Studiums konnte ich dadurch einen positiven Kontakt zum Lehrstuhl aufbauen. Besonders dankbar bin ich für die hohe fachliche Kompetenz, von der ich vielfach profitieren konnte, sowie dem hervorragenden Zusammenwirken.

Diese positive und konstruktive Atmosphäre, die von der Lehrstuhlleitung vorgelebt wird, spiegelt sich auch im Mitarbeiterkreis am Lehrstuhl wider. Ich bin äußerst dankbar für das tolle Miteinander, die inspirierenden Diskussionen sowie die zahlreichen Tassen Kaffee, die wir gemeinsam getrunken haben. Im Besonderen möchte ich mich bei Akshay Srivatsa bedanken, der über viele Jahre ein ausgezeichneter Bürokollege und reger Diskussions- und Kooperationspartner war. Ich bin ausgesprochen dankbar für die gegenseitige Motivation und Unterstützung sowie die gemeinsamen Publikationen.

[1] Die Forschung für diese Arbeit wurde von der Deutschen Forschungsgemeinschaft (DFG) durch den Sonderforschungsbereich (SFB) 89 Invasive Computing (Projektnummer 146371743) finanziert.

Der Platz reicht nicht aus, um allen Beteiligten am Invasive Computing Projekt zu danken. Mein Dank gilt zunächst allen für das, was sie bereits vor meiner Zeit auf die Beine gestellt haben. Besonderer Dank geht dabei an Christoph Erhardt von der Betriebssystemgruppe aus Erlangen, der mir in meiner Anfangszeit eine ausgesprochen große Unterstützung für das Verständnis des Gesamtprojekts und insbesondere der Hardware-Software-Interaktion war. Außerdem erwarteten ihn fast täglich meine Anrufe, damit wir erneut gemeinsam einige Bugs, sowohl in Hardware als auch in Software, fixen konnten. Weiterer Dank geht an die Mitwirkenden im Invasive Computing Demonstrator-Integrationsteam, mit denen ich erfolgreich zusammenarbeiten durfte: Dr. Leonard Masing, Nidhi Anantharajaiah, Marcel Brand, Dr. Frank Hannig, Dr. Thomas Wild, Dr. Manuel Mohr, Dr. Gabor Drescher, Florian Schmaus, Sebastian Maier, Tobias Langer, Jonas Rabenstein, Andreas Fried, Dr. Alexander Pöppel und alle, die ich an dieser Stelle vergessen habe.

Besonderen Dank möchte ich auch allen aussprechen, mit denen ich gemeinsam an Publikationen arbeiten durfte. Hardwareseitig danke ich Prof. Dr. Herkersdorf, Dr. Thomas Wild, Akshay Srivatsa und einigen, der ehemals von mir betreuten, Studenten: Andreas Schenk, Oliver Lenke, Lars Nolte, Temur Sabirov, Tim Twardzik und Nora Pohle. Ich hatte das große Privileg, eine Vielzahl herausragender Studenten zu finden und in einem außergewöhnlichen Team gemeinsam mit ihnen Konzepte zu entwickeln, Hardware zu bauen, Bugs zu fixen, Paper zu verfassen und viel Zeit in und außerhalb der Universität zu verbringen. Ohne sie wären die Projekte und Publikationen nicht in diesem Umfang möglich gewesen. Ganz besonders hervorheben möchte ich dabei Lars Nolte und Oliver Lenke, mit denen ich über einen Zeitraum von mehr als drei Jahren zusammenarbeiten durfte. Ihren Enthusiasmus, ihre außerordentliche Leistungsbereitschaft und ihre Freundschaft möchte ich nicht missen. Ich wünsche ihnen, und auch Tim Twardzik, der mich regelmäßig beim Tennis in die Knie zwingt, viel Erfolg bei ihren eigenen Promotionen.

Betriebssystemseitig gilt mein großer Dank Prof. Dr. Wolfgang Schröder-Preikschat vom Lehrstuhl für Verteilte Systeme und Betriebssysteme der Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU) und seinen Mitarbeitern Sebastian Maier und Florian Schmaus. Unsere Zusammenarbeit an Publikationen war ausgesprochen konstruktiv, produktiv und jederzeit von höchster Wertschätzung geprägt. Ich durfte enorm viel von ihnen lernen und habe die Zeit in und neben der Arbeit sehr genossen. Vielen Dank für die Mitentwicklung des `SHARQ` Konzepts (Chapter 4) und der softwareseitigen Implementierung.

Des Weiteren geht mein besonderer Dank an Andreas Fried vom Lehrstuhl für Programmierparadigmen des Karlsruher Instituts für Technologie (KIT). Seine fachübergreifende Expertise, ausgeprägte Hilfsbereitschaft sowie unermüdliche Integrations- und Paperschreibarbeit haben die gemeinsamen Projekte nicht nur inhaltlich, sondern auch zwischenmenschlich sehr bereichert. Ich bin mir sicher, dass ausschließlich die begrenzte Zeit weitere gemeinsame Projekte und Publikationen verhindert hat. Vielen Dank für die Mitentwicklung des `NEMESYS` Konzepts (Chapter 5) und der softwareseitigen Implementierung.

Ganz besonders möchte ich mich zudem bei meiner wundervollen Frau Eva bedanken, die mich in jeglicher Hinsicht unterstützt und mir den Rücken freigehalten hat. Zusammen mit unserem wunderbaren Sohn Levi waren sie mir in den letzten Monaten, des zugegebenermaßen manchmal zähen Schreibens, Motivationsspritze und unendliche Freude. Zu guter Letzt bin ich meinen Eltern ausgesprochen dankbar, die mich auf meinem bisherigen Lebensweg stets und auf unbeschreibliche Art unterstützt, geliebt und gefördert haben.

# Zusammenfassung

Der unersättlich steigende Bedarf an Rechenleistung war die treibende Kraft hinter der enormen Entwicklung von Computersystemen. Um mit dem Mooreschen Gesetz Schritt halten zu können, war es unerlässlich, Mehr- und Vielkernarchitekturen mit komplexen Cachehierarchien, skalierbaren Interconnects und mehreren verteilten, aber gemeinsam genutzten, Speichern zu entwickeln. Dennoch stellen die begrenzte Speicherbandbreite und die zunehmende Komplexität der Systeme enorme Herausforderungen für deren Programmierung und die Ausnutzung ihrer nominalen Rechenleistung dar. Datenübertragung und Interprozesskommunikation tragen erheblich zum Stromverbrauch und der beschränkten Performanz heutiger Systeme bei. Um den Schwierigkeiten fehlender Datenlokalität und der begrenzten Speicherbandbreite entgegenzuwirken, wurde die speichernahe Prozessierung erforscht. Trotz alledem ist effiziente Interprozesskommunikation weiterhin essenziell – insbesondere in kachelbasierten Architekturen mit verteilten Rechen- und Speicherkacheln.

Daher leistet diese Arbeit Forschungsbeiträge, inwiefern die vielversprechende speichernahe Prozessierung angewendet werden kann, um einige Schlüsselherausforderungen der Interprozesskommunikation in kachelbasierten Architekturen zu verringern. Da Kommunikations- und Synchronisationsmechanismen in der Regel auf Betriebssystem- oder Bibliotheksebene integriert und für den Anwendungsprogrammierer nicht sichtbar sind, liegt der Schwerpunkt dieser Arbeit auf der speichernahen Beschleunigung von Funktionalitäten des Betriebs- und Laufzeitsystems. Um den Nutzen der Beiträge zu evaluieren, werden ganze Benchmark-Anwendungen, wie die NAS-Parallel oder IMSuite Benchmarks, verwendet. Diese nutzen die vorgeschlagenen Beschleuniger-Erweiterungen des Betriebs- und Laufzeitsystems.

Zunächst wird ein speichernaher Synchronisationsbeschleuniger vorgeschlagen, der den Overhead der Interprozesssynchronisation zwischen Kacheln verringert. Durch die speichernahe Ausführung komplexer atomarer Operationen wird eine bessere Daten-zu-Task-Lokalität für gängige nebenläufige Datenstrukturen wie Warteschlangen, Stapel oder Semaphoren erreicht. Die Evaluierung zeigt den Nutzen dieses Ansatzes, insbesondere bei Verwendung für remote-ausgeführte Operationen sowie in größeren Systemen. Während sich dieser erste Beitrag auf die Interprozesssynchronisation konzentriert, zielen die folgenden Lösungen auf Interprozesskommunikation inklusive Datentransfer und Empfängerbenachrichtigung ab.

Der zweite Beitrag erweitert das Konzept zu einem Beschleuniger für speichernahe Warteschlangenverwaltung, da Warteschlangen ein wichtiges Interprozesskommunikationsmittel in Bibliotheken und im Laufzeitsystem sind. Der vorgeschlagene Hardware-Software-Co-Design-Ansatz kombiniert die Performanz von hardwarebeschleunigter Warteschlangen- und Speicherverwaltung, einschließlich Datentransfer und Empfängerbenachrichtigung, mit der von Software-Warteschlangen bekannten Flexibilität und Konfigurierbarkeit. Die Auswertung mit den MPI-basierten NAS-Parallel-Benchmarks zeigt eine Leistungsverbesserung von bis zu 43 % für kommunikationsintensive Anwendungen.

Während der zweite Beitrag Warteschlangen-basierte Kommunikation behandelt, die durch DMAs effizient unterstützt werden kann, wird im Folgenden ein komplexeres Interprozesskommunikationsszenario untersucht, welches den Transfer von Objektgraphen erfordert. Aufgrund dessen wird im dritten Beitrag der vorliegenden Arbeit ein speichernaher Graphkopierbeschleuniger vorgeschlagen, der in den Interprozesskommunikationsmechanismus des PGAS-Programmierparadigmas (Partitioned Global Address Space) integriert ist. Der vorgeschlagene Beschleuniger führt die Cache-unfreundlichen und speicherintensiven Graphkopieroperationen speichernah aus, anstatt remote über das Network-on-Chip (NoC) und die Cachehierarchie. Die Auswertung mit den PGAS-basierten X10 IMSuite-Benchmarks zeigt eine Beschleunigung der Anwendungsperformanz zwischen 1,35x und 3,85x mit einem Durchschnitt von 2,2x.

Die drei originären wissenschaftlichen Beiträge wurden erforscht, auf einer FPGA-Prototypen-Plattform implementiert und evaluiert. Die Entwicklungen und Publikationen erfolgten in enger Zusammenarbeit mit Hardware- und Softwareingenieuren des Sonderforschungsbereichs (SFB) 89 Invasive Computing, der durch die Deutschen Forschungsgemeinschaft (DFG) finanziert wird. Die Hauptkooperationspartner hinsichtlich Betriebs- und Laufzeitsystem waren dabei der Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme) der Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU) sowie der Lehrstuhl für Programmierparadigmen des Karlsruher Instituts für Technologie (KIT). Darüber hinaus wurde das verwendete NoC-Interconnect vom Institut für Technik der Informationsverarbeitung des KIT entwickelt.

Nach meinem besten Wissen ist die vorgestellte Arbeit die erste, die sich interdisziplinär mit den Herausforderungen der Interprozesssynchronisation und -kommunikation in kachelbasierten Vielkernarchitekturen befasst, indem speichernahe Beschleuniger eng in das Betriebs- und Laufzeitsystem integriert werden.

# Abstract

The ubiquitous need for evermore computing performance has been the driving force behind the astonishing evolution of computer systems. Multi- and many-core architectures with sophisticated cache hierarchies, scalable interconnects, and multiple distributed shared memories became inevitable to keep up with Moore's law. However, the limited memory bandwidth and growing complexity of systems pose formidable challenges to multiprocessor programming and the exploitation of the architectures' nominal computing performance. Data movement and inter-process communication contribute significantly to today's systems' power consumption and performance limitation. Near-memory computing emerged to counteract data locality challenges and limited memory bandwidth. Nevertheless, the need for efficient inter-process communication persists, particularly in tile-based architectures with distributed processing and memory nodes.

Therefore, this thesis contributes research results on applying the promising near-memory computing paradigm to mitigate several key challenges of inter-process communication on tile-based many-core architectures. As communication and synchronization mechanisms are typically integrated at the operating system or library level, opaque to the application programmer, this thesis focuses on the near-memory acceleration of run-time support functionalities. The benefit of the contributions is evaluated with entire benchmark applications like the NAS-Parallel or IMSuite benchmarks, which utilize the proposed accelerator enhancements of the operating and run-time system.

First, a near-memory synchronization accelerator is proposed, which alleviates the overhead of inter-process synchronization on tile-based architectures. Through remote near-memory execution of complex atomic operations, a better data-to-task locality is achieved for well-established concurrent data structures such as queues, stacks, or semaphores. The evaluation revealed the benefit of this approach, especially when used remotely and in bigger systems. While the first contribution focuses on inter-process synchronization, the following solutions target inter-process communication, including data transfer and receiver notification.

The second contribution extends the concept to a near-memory queue management accelerator since queues are an important means of inter-process communication in libraries and the run-time system. The proposed hardware-software co-design approach combines the performance of hardware-accelerated queue and memory management, including data transfer and receiver notification, with the flexibility and configurability known of software queues. The evaluation with the MPI-based NAS parallel benchmarks demonstrates a performance improvement of up to 43% for communication-intensive applications.

While the second contribution targets queue-based communication, which can be assisted by DMAs efficiently, a more complex inter-process communication scenario requiring object graph transfers is researched next. Therefore, the third contribution of this thesis proposes a near-memory graph copy accelerator integrated into the inter-process communication mechanism of the partitioned global address space (PGAS) programming paradigm. The proposed accelerator performs the cache-unfriendly and memory-intensive graph copy

operation near-memory instead of remotely over the Network-on-Chip (NoC) and the cache hierarchy. The evaluation with the PGAS-based X10 IMSuite benchmarks illustrates an application performance speedup between 1.35x and 3.85x with an average of 2.2x.

The three original scientific contributions were researched, implemented, and evaluated on an FPGA prototyping platform. The contributions were developed and published in close collaboration with hardware and software engineers of the Transregional Collaborative Research Center (TCRC) 89 Invasive Computing, funded by the German Research Foundation (DFG). Thereby, the main cooperation concerning the operating and run-time support system has been with the Distributed Systems and Operating Systems group (Department of Computer Science 4) from the Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU) and the programming paradigms group from the Karlsruhe Institute of Technology (KIT). Furthermore, the underlying NoC interconnect has been developed by the Institute for Information Processing Technologies from KIT.

To the best of my knowledge, the presented work is the first that tackles inter-process synchronization and communication challenges on tile-based many-core architectures in an interdisciplinary manner by tightly integrating near-memory accelerators into the operating and run-time system.

# List of Author's Publications

## Publications Contributing To This Disseration

**Conference Paper.** [1] S. Rheindt, A. Schenk, A. Srivatsa, T. Wild, and A. Herkersdorf. CaCAO: Complex and Compositional Atomic Operations for NoC-Based Manycore Platforms. In *Architecture of Computing Systems - ARCS 2018 - 31st International Conference, Braunschweig, Germany, April 9-12, 2018, Proceedings*, pages 139–152, 2018. doi:10.1007/978-3-319-77610-1_11

**Conference Paper.** [2] S. Rheindt, S. Maier, F. Schmaus, T. Wild, W. Schröder-Preikschat, and A. Herkersdorf. SHARQ: Software-Defined Hardware-Managed Queues for Tile-Based Manycore Architectures. In *Embedded Computer Systems: Architectures, Modeling, and Simulation - 19th International Conference, SAMOS 2019, Samos, Greece, July 7-11, 2019, Proceedings*, pages 212–225, 2019. doi:10.1007/978-3-030-27562-4_15

**Conference Paper.** [3] S. Rheindt, A. Fried, O. Lenke, L. Nolte, T. Wild, and A. Herkersdorf. NEMESYS: Near-Memory Graph Copy Enhanced System-Software. In *Proceedings of the International Symposium on Memory Systems, MEMSYS 2019, Washington, DC, USA, September 30 - October 03, 2019*, pages 3–18. ACM, 2019. doi:10.1145/3357526.3357545

**Conference Paper.** [4] S. Rheindt, A. Fried, O. Lenke, L. Nolte, T. Sabirov, T. Twardzik, T. Wild, and A. Herkersdorf. X-CEL: A Method to Estimate Near-Memory Acceleration Potential in Tile-Based MPSoCs. In *Architecture of Computing Systems - ARCS 2020 - 33rd International Conference, Aachen, Germany, May 25-28, 2020, Proceedings*, pages 109–123, 2020. doi:10.1007/978-3-030-52794-5_9

**Conference Paper.** [5] S. Rheindt, T. Sabirov, O. Lenke, T. Wild, and A. Herkersdorf. X-Centric: A Survey on Compute-, Memory- and Application-Centric Computer Architectures. In *MEMSYS 2020: The International Symposium on Memory Systems, Washington, DC, USA, September, 2020*, pages 178–193. ACM, 2020. doi:10.1145/3422575.3422792

**Journal Paper.** [6] S. Rheindt, S. Maier, N. Pohle, L. Nolte, O. Lenke, F. Schmaus, T. Wild, W. Schröder-Preikschat, and A. Herkersdorf. DySHARQ: Dynamic Software-Defined Hardware-Managed Queues for Tile-Based Architectures. *International Journal of Parallel Programming*, pages 1–35, 2020. doi:10.1007/s10766-020-00687-7

**Book Chapter.** [7] S. Rheindt, A. Srivatsa, O. Lenke, L. Nolte, T. Wild, and A. Herkersdorf. *Tackling the MPSoC Data Locality Challenge*, chapter 5, pages 85–117. John Wiley & Sons, Ltd, 2021. doi:10.1002/9781119818298.ch5

## Publications Beyond the Scope of This Disseration

**Conference Paper.** [8] A. Srivatsa, S. Rheindt, T. Wild, and A. Herkersdorf. Region-Based Cache Coherence for Tiled MPSoCs. In M. Alioto, H. H. Li, J. Becker, U. Schlichtmann, and R. Sridhar, editors, *30th IEEE International System-on-Chip Conference, SOCC 2017, Munich, Germany, September 5-8, 2017*, pages 286–291. IEEE, 2017. doi:10.1109/SOCC.2017.8226059

**Conference Paper.** [9] A. Srivatsa, S. Rheindt, D. Gabriel, T. Wild, and A. Herkersdorf. CoD: Coherence-on-Demand - Runtime Adaptable Working Set Coherence for DSM-Based Manycore Architectures. In *Embedded Computer Systems: Architectures, Modeling, and Simulation - 19th International Conference, SAMOS 2019, Samos, Greece, July 7-11, 2019, Proceedings*, pages 18–33, 2019. doi:10.1007/978-3-030-27562-4_2

**Journal Paper.** [10] A. Srivatsa, M. Mansour, S. Rheindt, D. Gabriel, T. Wild, and A. Herkersdorf. DynaCo: Dynamic Coherence Management for Tiled Manycore Architectures. *International Journal of Parallel Programming*, pages 1–30, 2021. doi:10.1007/s10766-020-00688-6

# Contents

# List of Figures

# List of Tables

# List of Acronyms

| | |
|---|---|
| ACM | Association for Computing Machinery. |
| AMD | Advanced Micro Devices, Inc.. |
| API | Application Programming Interface. |
| APU | Accelerated Processing Unit. |
| ARM | Advanced RISC Machines Ltd.. |
| AS-NMA | Application-Specific Near-Memory Acceleration. |
| | |
| BFS | Breadth-First Search. |
| BRAM | Block Random Access Memory. |
| | |
| CaCAO | Complex and Compositional Atomic Operations. |
| CAS | Compare-and-Swap. |
| CPU | Central Processing Unit. |
| CS | Critical Section. |
| | |
| DDR | Double Data Rate. |
| DFG | Deutsche Forschungsgemeinschaft (German Research Foundation). |
| DFS | Depth-First Search. |
| DM | Distributed Memory. |
| DMA | Direct Memory Access. |
| DRAM | Dynamic Random Access Memory. |
| DSM | Distributed Shared Memory. |
| DSP | Digital Signal Processor. |
| | |
| FeRAM | Ferroelectric Random Access Memory. |
| FIFO | First In, First Out. |
| FPGA | Field Programmable Gate Array. |
| FSM | Finite State Machine. |
| | |
| GPU | Graphics-Processing Unit. |
| | |
| HBM | High-Bandwidth Memory. |
| HMC | Hybrid Memory Cube. |
| HPC | High Performance Computing. |
| HPP | Heterogeneous Programming Paradigm. |
| HSA | Heterogeneous System Architecture. |
| | |
| IDC | International Data Corporation. |
| IF | Interface. |

| | |
|---|---|
| ILP | Instruction-Level Parallelism. |
| IoT | Internet of Things. |
| IPC | Inter-Process Communication. |
| ISA | Instruction Set Architecture. |
| JEDEC | Joint Electron Device Engineering Council. |
| LIFO | Last In, First Out. |
| LL/SC | Load-Linked/Store-Conditional. |
| LUT | Look-Up Table. |
| LWP | Light-Weight Processor. |
| MCDRAM | Multi-Channel DRAM. |
| MIC | Many Integrated Core. |
| MMU | Memory Management Unit. |
| MP | Message Passing. |
| MPI | Message Passing Interface. |
| MPSoC | Multi-Processor System-on-Chip. |
| MRAM | Magnetoresistive Random Access Memory. |
| MUX | Multiplexer. |
| NA | Network Adapter. |
| NAS | Numerical Aerodynamic Simulation. |
| NCA | Near-Cache Accelerator. |
| NEMESYS | Near-Memory Graph Copy Enhanced System-Software. |
| NMA | Near-Memory Accelerator. |
| NMC | Near-Memory Computing. |
| NMCORE | Near-Memory Core. |
| NoC | Network-on-Chip. |
| NPB | NAS Parallel Benchmarks. |
| NUMA | Nom-Uniform Memory Access. |
| OS | Operating System. |
| PCIe | Peripheral Component Interconnect Express. |
| PGAS | Partitioned Global Address Space. |
| PIM | Processing-in-Memory. |
| POSIX | Portable Operating System Interface for Unix. |
| Pthread | POSIX Thread. |
| RAM | Random Access Memory. |
| RBCC | Region-Based Cache Coherence. |
| RISC | Reduced Instruction Set Computer. |
| RNMA | Remote Near-Memory Acceleration. |
| RoI | Region of Interest. |
| RPC | Remote-Procedure Call. |
| RTS-NMA | Run-Time Support Near-Memory Acceleration. |

| | |
|---|---|
| RTTI | Run-Time Type Information. |
| SCC | Single-Chip Cloud Computer. |
| SFB | Sonderforschungsbereich. |
| SHARQ | Software-Defined Hardware-Managed Queues. |
| SM | Shared Memory. |
| SMQ | Software-Managed Queue. |
| SoC | System-on-Chip. |
| SRAM | Static Random Access Memory. |
| TAS | Test-and-Set. |
| TCRC | Transregional Collaborative Research Center. |
| TLM | Tile-Local Memory. |
| TLP | Thread-Level Parallelism. |
| TSV | Through-Silicon Via. |
| UMA | Uniform Memory Access. |
| VLIW | Very Long Instruction Word. |
| VM | Virtual Machine. |

# 1 Introduction

> " Surely there must be a less primitive way of making big changes in the store than by pushing vast numbers of words back and forth through the von Neumann bottleneck. Not only is this tube a literal bottleneck for the data traffic of a problem, but, more importantly, it is an intellectual bottleneck that has kept us tied to word at-a-time thinking instead of encouraging us to think in terms of the larger conceptual units of the task at hand. Thus programming is basically planning and detailing the enormous traffic of words through the von Neumann bottleneck, and much of that traffic concerns not significant data itself but where to find it."
>
> — John Backus, 1977 ACM Turing award lecture [11]

## 1.1 Motivation

"Data is the new oil" is a recurring phrase over the last decade, first pronounced by Clive Humby, a British mathematician, in 2006 [12]. Since then, it has been widely adopted with various nuances by several experts from industry and academia, culminating in an article by David Parkins in the May 2017 issue of The Economist with the title: "The world's most valuable resource is no longer oil, but data." [13] These statements are pointing to a rapidly and exponentially growing global data sphere, which is still in its infancy [14]. The International Data Corporation (IDC) predicts that the global data sphere will grow from 33 Zettabytes back in 2018 to 175 Zettabytes [1] by 2025 [15]. The ever-growing challenge is to extract meaningful information from the vast amount of often highly unstructured big data, which originates from e.g., mobile and IoT (Internet of Things) devices, cyber-physical systems, sensors, social networks, multimedia applications [14, 16]. It needs to be dealt with since data is "valuable, but if unrefined it cannot really be used" (Clive Humby [12]). Similarly, in 2011, Peter Sondergaard assessed: "Information is the oil of the 21st century, and analytics is the combustion engine." [17].

Consequently, data processing has always been the driving force behind the evolution of computer architectures and the ubiquitous need for evermore computing performance. Although empowered by Moore's law (i.e., the doubling of the transistor count per area roughly every 24 months [18]), performance scaling of computer architectures has been a challenging and multifaceted endeavor throughout its history. Solely providing nominal computing performance is not the universal remedy since there exist many important influencing factors that impede the full exploitation of parallel architectures. First of all, the

[1] Put in perspective, 175 Zettabytes correspond to roughly 28 million years of uncompressed RGB $3 \times 8$ Bit full HD video material with a resolution of $1920 \times 1080$ pixels and 25 frames per second.

theoretical speedup of an application consisting of sequential and parallel parts is bounded by Amdahl's law [2]. Second, data movement at large and inter-process communication, in particular, contribute significantly to the power consumption and performance limitation of today's systems [19, 20, 16, 21]. The performance of parallel programming highly depends on efficient and scalable cache coherence and memory consistency, as well as inter-process communication and synchronization [22]. These pose formidable challenges, especially on architectures with physically distributed memories and processing nodes. Third, the memory wall is still not conquered, and, to date, the bulk of systems keep on suffering from the von Neumann bottleneck, i.e., processors exhibiting limited memory bandwidth [23, 24]. This can be compensated by elaborate cache and memory hierarchies to a certain degree. In the past, applications offered plenty of spatial and temporal data-to-task locality, thus being a good fit for caching. However, the data sphere and application requirements have been rapidly evolving over the past years. Application classes with large, unstructured, and irregular data sets have emerged. As these are immensely cache-unfriendly and by far exceed the capacity of modern cache hierarchies, the efficiency of caching is limited [16, 19].

In conjunction with the architectural developments of today's many-core architectures, these changing application characteristics lead to an ever-growing data-to-task dis-locality, especially for memory-intensive applications [23, 19, 16, 25]. This *locality wall*, as called by Peter Kogge, University of Notre Dame, at the MEMSYS 2017 conference [19], is closely related to the von Neumann bottleneck [23, 19]. Kogge showed evidence that the overall energy consumption and significant performance limitations of today's architectures are mainly caused by data transfers between the main memory and the processor cores. This is owing to the fact that energy cost and access latencies to global memory as compared to local registers are several magnitudes larger [19, 20, 16]. Many experts from industry and academia have further confirmed the data locality wall [23, 25, 26, 27, 16, 21] and even identified the understanding of data-to-task affinity and the management of data locality as yet unsolved problems, particularly for high-performance computing applications [26, 27].

Already in 1977, in his ACM Turing award lecture, John Backus pondered on this issue [11]: "`Surely there must be a less primitive way of making big changes in the store than by pushing vast numbers of words back and forth through the von Neumann bottleneck.`" Since then, it took almost 40 years until a major rethinking of computer architectural design had its breakthrough. Instead of building mainly compute-centric architectures (i.e., designing the systems around the processor and hence fetching and caching data close to it), memory-centric designs adopt an opposite approach. They leverage the increased proximity and bandwidth of in- or near-memory computing rather than moving data through the von Neumann bottleneck [19, 16, 28]. Thereby, data movement and interconnect traffic can be reduced, the gap between limited cache sizes and huge data sets can be bridged, and the energy footprint of applications can be lowered. In- and near-memory computing help to mitigate the *locality wall* by harnessing the increased memory bandwidth and the improved data-to-task locality when performing computations in or close to memory.

Consequently, a tremendous amount and diversity of in- and near-memory computing approaches have been emerging in recent years [29, 28]. Although the idea of in-memory computing dates back to the 1960s and experienced a research hype in the 1990s, an ac-

[2] In its basic form Amdahl's law defines the theoretical speedup $\eta$ of an application with a parallelizable fraction $p$ running on $N$ cores as: $\eta = \frac{1}{(1-p)+\frac{p}{N}}$ with an upper bound of: $\eta_\infty = \lim_{N\to\infty} \eta = \frac{1}{(1-p)}$

tual breakthrough and revival occurred during the past decade [28]. Most notably, the seminal innovation of through-silicon vias (TSVs) in 2009 [30] enabled the invention of 3D-stacked memory devices like the Hybrid Memory Cube (HMC) [31] and the High Bandwidth Memory (HBM) [32] in 2011 and 2013, respectively. The HMC offers a 3D-stacked and TSV-connected custom logic layer below several memory layers, enabling a tight coupling of processing elements and memory. The wide field of application and the variety of academic and commercial systems leveraging the memory-centric approach display its enormous potential [29, 16].

## 1.2 Problem Statement

Nevertheless, in- and near-memory computing [3] is still facing several challenges. First, numerous approaches use application-specific accelerators implemented in the logic layer of 3D-stacked memories and are tailored for a particular class of workloads. These systems, such as standalone special-purpose logic-enhanced memories or a host device connected to a logic-enhanced memory, are mostly custom designs to exploit the benefits of near-memory computing fully [29]. This is closely tied to the second challenge. The lack of a programming paradigm, including compilers, libraries, programming languages, or even operating systems, specific to near-memory computing impedes its standardization and ease of use. The memory view and providing cache coherence and memory consistency between the host processors and the near-memory units are challenging, and the integration of MMUs and virtual memory is not widespread [29, 28]. As Patterson et al. [33, 34, 28] pointed out, this was already an obstacle in the 1990s, when a clear vision for integration and programmability was missing. Whereas the innovations mentioned above have overcome many technological problems, programming challenges remain. A similar indication can be deduced from the fact that the HMC and HBM, the latter being a JEDEC standard, are often used in a compute-centric manner (i.e., as fast and high bandwidth memory technology with, for instance, an advanced memory controller integrated into the HMC's logic layer) in many recent commercial devices [35, 36, 37, 38].

Thus, the question will be how to circumvent that the current memory-centric innovations wind up just another higher bandwidth memory technology node. At the MEMSYS 2015 conference, Radulovic et al. [24] assessed that 3D-stacked memory, although it is a game-changing innovation, might merely be moving the memory wall if the concepts are not embedded in a holistic system-level approach.

Lessons learned from decades of computer engineering teach us that the compliance of architectural innovations with well-established architectures, commoditized programming paradigms, and legacy code applications are crucial for their enduring breakthrough [5, 28]. However, it is unlikely that the longed-for revolution of the programming paradigm will come fast [24]. This is in line with two statements by David Patterson from 2010: "`One of the biggest factors, though, is the degree of motivation. In the past, programmers could just wait for transistors to get smaller and faster, allowing microprocessors to become more powerful. So programs would run faster without`

[3] The recent trend of logic-enhanced 3D-stacked memories is often referred to as *in-memory computing* or *processing-in-memory* (PIM). However, it is intrinsically closer to *near-memory computing* as the logic is integrated below the memory stack instead of in the memory arrays themselves. Therefore, in the remainder of this thesis, it is referred to as *near-memory computing*.

any new programming effort, which was a big disincentive to anyone tempted to pioneer ways to write parallel code" and "Chipmakers are busy designing microprocessors that most programmers can't handle" [39]. Although these statements were made in the context of multi-core architectures, they are all the more relevant for heterogeneous systems consisting of multiple processors and near-memory computing nodes. Nevertheless, the extent of more efficient and architecture-aware programming models is limited, especially in legacy code for compute-centric architectures [39].

Based on these challenges and assertions, it is likely that the compute-centric paradigm will stay. Hence, the ambitious question is how the promising near-memory computing paradigm can be employed and embedded into existing design approaches to effectively mitigate the memory wall and equivalently and, more precisely, the more recent locality wall. In other words: a vision of how to apply near-memory computing to scalable compute-centric architectures with physically distributed processing and memory nodes is necessary.

On those architectures, significant power consuming, performance hindering, and scalability limiting effects are data movement, in general, and traffic of inter-process communication, in particular [19, 20, 16, 21]. This is because traditional compute-centric systems operate based on data movement between the processor cores and memory, which becomes a challenge much more demanding when scaling these systems requires incorporating numerous physically distributed shared memories and processor cores. Therefore, efficient inter-process communication and synchronization solutions are crucial for better exploiting the nominal computing performance. As communication and synchronization mechanisms are typically integrated at the operating system or library level, opaque to the application programmer, this thesis focuses on near-memory acceleration of run-time support functionalities. Although this approach would benefit a wide range of applications, it has to date experienced limited research, in contrast to the plurality of application-specific near-memory computing approaches [29].

Following the statement of John Backus, who said [11]: "Surely there must be a less primitive way of making big changes in the store than by pushing vast numbers of words back and forth through the von Neumann bottleneck", a solution should increase data-to-task locality and decrease data movement instead of simply speeding it up by widening the bottleneck. This problem is most likely to be tackled holistically at the system level, and the approach should be independent of the actual memory technology (be it standard DRAM or advanced 3D-stacked HBM or HMC) [24].

In conclusion, the middle part of Figure 1.1 summarizes the problem statement with the culminating research question: How to apply the promising near-memory computing approach to inter-process communication challenges of tile-based many-core architectures?

## 1.3 Contributions

The goal of this thesis is to study how to mitigate inter-process communication challenges of tile-based many-core architectures through near-memory acceleration. The contributions help to tackle the data-to-task locality wall and profit a wide range of parallel applications since the proposed accelerators are integrated into synchronization primitives, communication libraries, and the operating and run-time support system.

All contributions and results of this thesis were published at five international conferences [1, 2, 3, 4, 5], in a journal article [6], and a book chapter [7]. A graphical representation

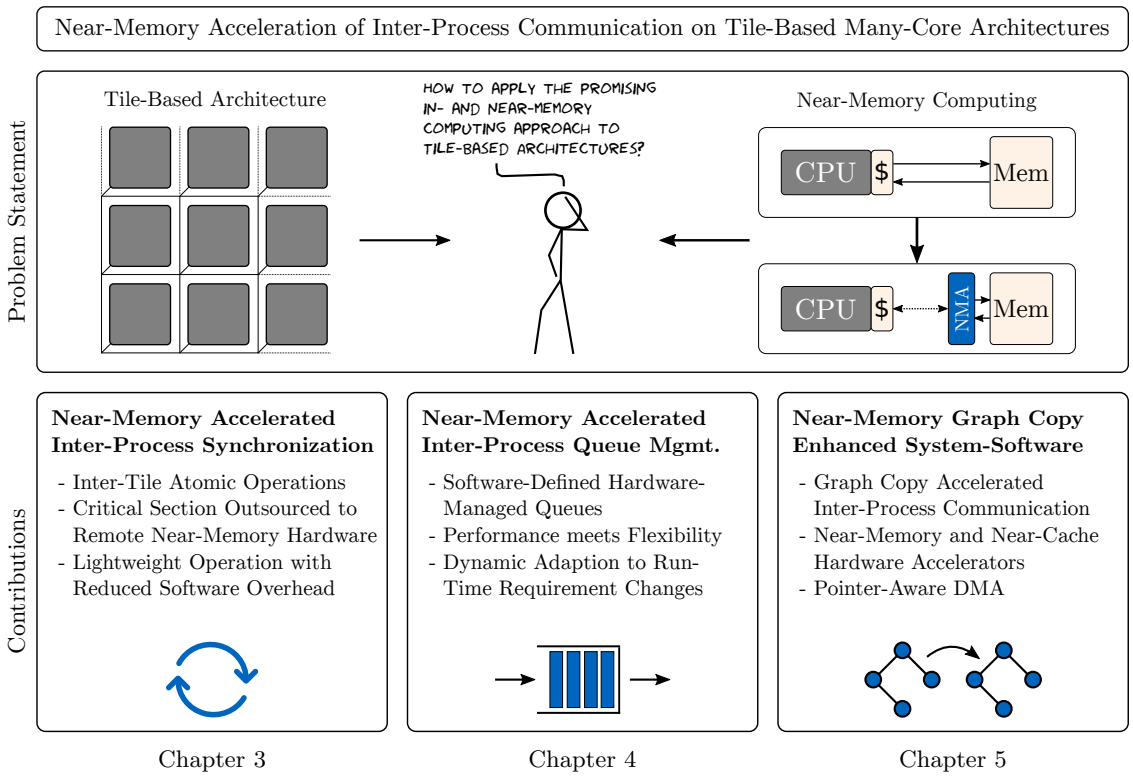



**Figure 1.1: Problem statement and contributions.** Contributions to the field of near-memory acceleration to mitigate inter-process communication challenges on tile-based architectures.

of the contributions is depicted in Figure 1.1. They were developed and published in close collaboration with hardware and software engineers of the DFG TCRC/SFB 89 Invasive Computing [40]. Thereby, the main cooperation concerning the operating and run-time support system and the hardware-software co-design has been with the Distributed Systems and Operating Systems group (Department of Computer Science 4) from FAU and the programming paradigms group from KIT.

**First, based on my publications [1, 7], a near-memory synchronization accelerator is presented in Chapter 3 which alleviates the overhead of inter-process synchronization of concurrent data structures.** This contribution fills the void of missing inter-tile support for well-established synchronization primitives extensively employed in central run-time support functionalities and libraries (e.g., queues, stacks, semaphores). Through remote near-memory execution, a better data-to-task locality is achieved. As thread synchronization is prevalent in the critical path of parallel applications, its improvement affects the overall performance positively. The evaluation of the near-memory accelerator integrated into a tile-based many-core architecture demonstrates the benefit of this approach, especially when used for remote operations and in bigger systems. The proposed synchronization accelerator solely deals with pointer operations on the concurrent data structures without handling inter-tile payload transfers. The next contribution will incorporate the latter as well.

**Second, based on my publications [2, 6, 7], a near-memory queue management accelerator is presented in Chapter 4 which facilitates efficient inter-process communication.** As queues are a beneficial and widely employed means of

communication in central places of the run-time system or libraries (e.g., for message passing, data distribution, task scheduling, or in general, data transfers with subsequent processing), it is essential to provide an efficient queue implementation in order to avoid performance bottlenecks. The proposed software-defined hardware-managed queues combine the performance of hardware-accelerated queue management with the configurability and adaptability known of software queue implementations. The conducted study revealed that coupling the queue management with efficient data transfers is crucial for data-to-task locality, particularly in tile-based systems with physically distributed memory.

**Third, based on my publications [3, 4, 7], a near-memory graph copy accelerator is presented in Chapter 5 which mitigates data-to-task locality challenges in the inter-process communication of object-oriented PGAS programming languages.** In contrast to the queue management approach, introduced in Chapter 4, in which low latency data transfers are enabled by utilizing a DMA engine, the latter is not capable of copying more complex data structures, such as object graphs. The proposed accelerator performs the cache-unfriendly and memory-intensive graph copy operation near-memory instead of remotely over the NoC and the cache hierarchy. Even for systems with physically distributed memory hierarchies, this contribution introduces a mechanism to efficiently copy graphs between memories located in different tiles without sacrificing essential near-memory benefits. Since the near-memory accelerator is required to be correctly synchronized with the processor cores (i.e., establishing cache coherence and memory consistency), Chapter 5 further proposes a near-cache accelerator capable of performing cache writebacks and invalidations of object graphs. Both types of accelerators are integrated into the run-time support system instead of being exposed to the application programmer to maintain ease of programmability and profit a wide range of applications without requiring changes to the application code itself.

To the best of my knowledge, the presented work is the first that tackles data-to-task locality challenges in tile-based many-core architectures in an interdisciplinary manner by integrating near-cache and near-memory accelerators into the run-time system in order to mitigate inter-process communication and synchronization challenges.

I contributed to three further publications that are not within this thesis's scope. While this thesis targets inter-process communication and synchronization, those publications authored by Srivatsa et al. [8, 9, 10] focus on dynamic and scalable inter-tile cache coherence.

## 1.4 Outline

The remainder of this thesis is structured as follows. As the contributions apply near-memory acceleration to mitigate inter-process communication challenges of tile-based many-core architectures, Chapter 2 provides state of the art on these three topics. Then, Chapter 3 presents the accelerator for inter-process synchronization. Chapter 4 introduces the queue management accelerator to facilitate efficient message-passing inter-process communication. Next, the near-memory graph copy accelerator is proposed in Chapter 5 which optimizes the inter-process communication of the PGAS programming paradigm. Finally, Chapter 6 concludes this thesis and gives an outlook on future work.

# 2 State of the Art

> " Those who cannot remember the past are condemned to repeat it."
>
> — George Santayana [41]

This thesis applies near-memory acceleration to mitigate inter-process communication challenges of tile-based many-core architectures. Therefore, this chapter introduces the state of the art in these three domains. First, Section 2.1 provides an overview of compute-centric architectures, including their key evolutionary steps and the most prominent programming paradigms. A particular focus is put on tile-based many-cores as they are the target architecture of this thesis. Next, the memory-centric in- and near-memory computing paradigm is presented in Section 2.2. It arose to counteract data locality challenges and limited memory bandwidth. Normally, the compute- and memory-centric approaches are contrasting. However, this thesis leverages and integrates near-memory acceleration in the context of a compute-centric design and programming paradigm. The objective is to optimize inter-process communication and synchronization mechanisms. Section 2.3 presents them and covers several related approaches. As communication and synchronization are typically integrated at the operating system or library level, opaque to the application programmer, this thesis focuses on near-memory acceleration of run-time support functionalities. Finally, Section 2.4 presents the tile-based prototype platform utilized to demonstrate the contributions of this thesis.

## 2.1 Compute-Centric Architectures

The majority of systems throughout the history of computer engineering have been compute-centric [23]. These systems are basically designed around the processor, and data is fetched from memory whenever required by the application.

This section first describes the major evolutionary steps of compute-centric architectures in Section 2.1.1 followed by a taxonomy in Section 2.1.2. Then, several related tile-based many-core designs are presented in Section 2.1.3. An excursion to heterogeneous multi- and many-core architectures incorporating various accelerators or special-purpose processors is conducted in Section 2.1.4. They exhibit similarities to the accelerator-enhanced architecture envisioned in this thesis. These architectural elaborations are complemented by the corresponding programming paradigms in Section 2.1.5. Finally, this section is concluded by discussing the pros and cons of the compute-centric approach.

### 2.1.1 Evolution

The voracious demand for evermore compute performance has been the driving force behind the remarkable evolution of computer architectures. Crucial architectural innovations have often been sparked by hitting prominent *walls* of computer architecture. The most significant evolutionary steps and the walls which necessitated them are shown in Figure 2.1.

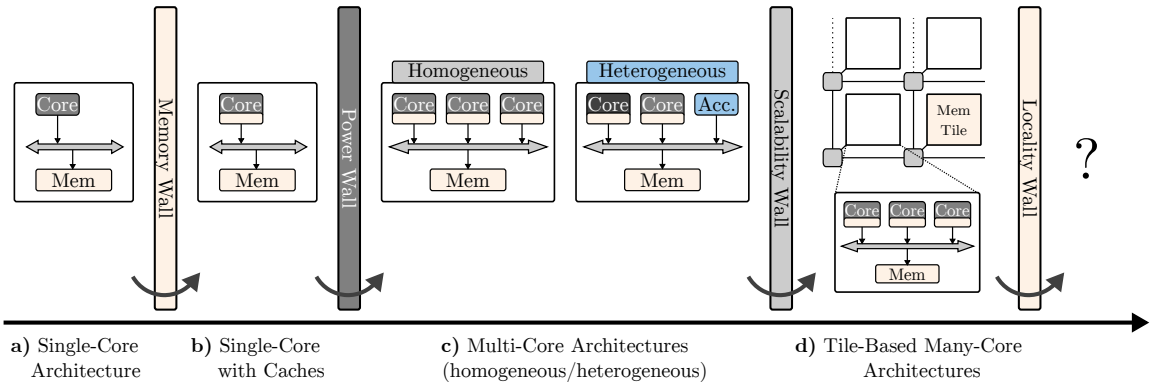



**Figure 2.1: Evolution of Computer Architectures.** Significant evolutionary steps of compute-centric architectures and the major walls of computer engineering.

The *memory wall* in the '90s (i.e., the growing disparity between faster processor and slower memory performance growth [33]) triggered many micro-architectural improvements and led to sophisticated cache hierarchies to counteract the limited off-chip memory bandwidth [33, 42]. Nevertheless, already in 1994, Wulf et al. [43] predicted that the memory wall would not be overcome entirely as the disparity between processor and memory performance exhibits exponential growth [42]. Furthermore, Kogge et al. [28, 19] pointed out that the off-chip memory bandwidth loss cannot be fully compensated by caching.

Empowered by Moore's law and Dennard scaling (i.e., scaling the clock frequency while keeping the power density constant [44]), the evolution of processor performance continued despite the memory wall. However, in 2004, the *power wall* denoted the end of Dennard scaling as increased power dissipation and design complexity hindered the further improvement of clock frequencies [45]. As the demand for performance scaling did not cease, an orthogonal approach to frequency scaling was explored. The power wall was tackled by leveraging multi- and many-core architectures instead of single-core processors. Through the achieved parallelism by splitting the workload to several cores on a single chip, a multi-core processor provides higher nominal performance while being operated at lower voltage and frequency [43, 46]. Nevertheless, the actual performance is bounded by Amdahl's law, the challenges of parallel programming, including cache coherence maintenance and limited memory bandwidth. Furthermore, sophisticated on-chip interconnect architectures are required to guarantee scalability.

The scalability has been introduced by the shift towards e.g., tile-based architectures with physically distributed memories and processing nodes to mitigate memory access contention and hot spots [47, 48, 49]. Nevertheless, data-to-task locality challenges between physically distributed memories and processing nodes arose. In 2017, Peter Kogge described this uprising challenge as the *locality wall* [19].

### 2.1.2 Taxonomy

Although compute-centric systems had formidable evolutionary steps, the advantages of particular architectures are thwarted by antagonistic effects of other optimizations. Therefore, since this led to a huge variety of architectures existing today, this section presents a taxonomy of parallel architectures. The classification is based on four dimensions: (1) Which **processing** elements are used? (2) Which **interconnect** is employed? (3) How is **memory** organized? (4) What **programming paradigm** is applied?

**Processing.** The processing aspect can be subdivided into homogeneous and heterogeneous architectures [45]. Homogeneous systems employ several identical cores. These can be either multiple (to date up to 64 [50]) powerful, out-of-order, multi-instruction issue cores, designed for high-performance general-purpose computing and typically used in e.g., desktop, cloud, or server processors like Intel's Core processor family or AMD Ryzen processors [51, 52, 53, 54]. A large number (e.g., several thousand for GPUs [55]) of massively parallel simple, in-order, narrow-issue cores is utilized in the *many-thread* approach.

In contrast, heterogeneous architectures combine different core types or incorporate additional accelerators. Examples are a combination of *strong* and *weak* cores like ARM's big.LITTLE or DynamIQ architectures [56, 57]. Or the additional use of special-purpose co-processors like GPUs, DSPs, FPGAs, or dedicated hardware accelerators.

**Interconnect.** In recent years, the ever-increasing number of cores poses the scalability and interconnect challenges mentioned above. Conventionally, multiple cores were connected by a shared bus. However, its efficiency and scalability are limited to a few cores. There also exist ring buses, point-to-point connections, or vendor-specific advanced interconnects. Furthermore, highly scalable Network-on-Chip (NoC) interconnects are on the rise in many academic but also recent commercial products like Intel's Knights Landing or Skylake server architectures [58]. Examples of related tile-based architectures based on such NoC interconnects are presented in Section 2.1.3 as they are the target architecture of this thesis. Also, systems with a hybrid interconnect architecture exist, e.g., a global NoC connecting several tiles while buses are utilized to connect multiple cores inside a tile [40, 48].

**Memory Organization.** Formerly, computer systems used to have a single *centralized memory*. However, all cores have to share the available memory bandwidth through a single physical memory interface. This may lead to a memory bottleneck and access hot spots. Thus, *physically distributed memory* is employed to counteract the disadvantages, especially in larger systems. There exist architectures with several global memories like Tilera's TILE64 [59] or Intel's Single Chip Cloud Computer (SCC) [60]. Nevertheless, also tile-based systems with tile-local memories in addition to global memories are present, like the Stanford DASH Multiprocessor [48].

In conjunction with the memory access permissions defined by the accompanied programming paradigm, the memory model [1] can be either shared memory (SM), distributed shared memory (DSM), or distributed memory (DM). In SM architectures, all cores typically exhibit the same memory access latencies, i.e., uniform memory access (UMA). DSM architectures combine physically distributed memory with shared memory programming. Although access from every core to any memory is permitted, the access latencies and memory bandwidth are non-uniform (NUMA), i.e., local memories can be accessed faster than remote ones. Thus, data-to-task locality becomes more relevant and, at the same time, more challenging to handle. Finally, distributed memory architectures (DM) assign private address spaces to individual processors or tiles and prohibit shared access to the other address spaces. Communication between cores or tiles requires explicit message-passing.

**Programming Paradigm.** The appropriate programming paradigms will be presented in more detail in Section 2.1.5. A programming model is an abstraction of a computer archi-

[1] The memory model is defined as the combination of the physical memory organization (i.e., centralized or distributed) and the applied communication model (i.e., shared memory or message-passing).

tecture and especially of the system's memory organization [61]. Shared memory programming is used on systems with a single address space and requires implicit cache coherence. In contrast, DM architectures are mostly programmed via explicit message-passing.

DSM architectures cannot be fully exploited by pure shared memory programming since it lacks the notion of data locality. Therefore, programming models taking account for the NUMA properties are desirable. One such is the partitioned global address space (PGAS) programming paradigm. It partitions the global address space, assigns a partition to each core or tile, and is aware of local and remote accesses. Thus, it retains the benefits of shared memory programming but adds the awareness of physically distributed memory.

Moreover, there exist heterogeneous programming models tailored for heterogeneous architectures and the incorporation of co-processors and dedicated accelerators.

**Summary.** In conclusion, there are various homogeneous and heterogeneous systems with different interconnect architectures, memory organizations, and programming paradigms.

This thesis targets tile-based many-core architectures with NoC interconnect and physically distributed memory (cf. Section 2.4). The contributions in the different chapters address inter-process synchronization in a shared memory programming environment (Chapter 3), message-passing-based inter-process communication (Chapter 4), as well as inter-process communication in the PGAS programming paradigm (Chapter 5). Moreover, incorporating hardware accelerators transforms the homogeneous tile-based many-core architecture into a heterogeneous system.

### 2.1.3 Tile-Based Many-Core Architectures

Tile-based architectures are assembled by replicating and interconnecting their basic building blocks, so-called tiles. While the global interconnect is mostly a 2D-mesh NoC, the tiles' internal structure and interconnect architecture are manifold. This section presents several exemplary designs.

The **Stanford DASH multiprocessor** [48], proposed in 1992, was the trailblazer for tile-based architectures. Each tile consists of four processor cores with private level-1 caches and a shared level-2 cache. A portion of the distributed shared memory is located in each tile. Up to 16 homogeneous tiles (i.e., 64 cores) are connected by a 4x4 2D-mesh interconnect. The Stanford DASH is kept fully cache-coherent and supports shared-memory programming.

**Tilera's** [2] **TILE64** [59] is another well-known tile-based many-core architecture consisting of 64 compute tiles interconnected by a 5-channel 8x8 mesh Network-on-Chip. Each compute tile contains a 3-way VLIW general-purpose processor and a two-level cache hierarchy which is kept globally coherent to enable shared-memory programming. Moreover, four physically distributed global memories are used in total. A DMA unit is employed to mitigate data-to-task locality challenges by performing efficient cache-to-cache, cache-to-memory, and memory-to-cache copies. Tilera's low-power and high-throughput chips are widely deployed in servers [62, 63].

**Intel's Single Chip Cloud Computer** (SCC) is a third example of a homogeneous tile-based 48-core architecture [60]. A 6x4 NoC connects its 24 tiles. Each tile contains two x86 cores and a private 2-level cache hierarchy which is kept coherent by hardware,

[2] In the past decade, Tilera was acquired by EZChip Semiconductor, which in turn was acquired by Mellanox in 2016. Finally, Nvidia acquired Mellanox in 2019.

neither intra- nor inter-tile. Moreover, the SCC contains four globally shared memory controllers residing at the corners of the chip. These can be used for shared memory communication. However, as no hardware coherence support exists, it has to be ensured by software. Alternatively, each tile is equipped with a message buffer to facilitate inter-process communication via explicit message-passing.

Furthermore, **Intel's Xeon Phi** processors [64] are widely employed in High-Performance Computing (HPC). Two product generations, which are based on Intel's Many Integrated Core (MIC) architecture [65], have been proposed. The second-generation *Knights Landing* (KNL) offers up to 72 multi-threaded, out-of-order cores, organized in 36 tiles interconnected by a 2D-mesh that supports full cache coherence. It uses two types of globally shared memory. Besides six channels for off-chip DDR4 memory, it also contains 16GB of on-package, high-bandwidth MCDRAM [38].

Another example is **Intel's Skylake** server processor architecture. It exists with up to 28 compute tiles and two memory tiles connected by a 6x5 mesh interconnect. Each tile contains one core with a 3-level cache hierarchy kept globally coherent among all cores.

Furthermore, **Kalray's Massively Parallel Processor Array** (MPPA) was presented in 2013 [49]. The system consists of 16 clusters connected by two separate bi-directional 2D-torus networks, one for data traffic and one for control signals. Each cluster or tile comprises 16 compute cores and one system core. Furthermore, each tile is equipped with a level-1 cache, a DMA unit, and a portion of the shared memory.

In summary, a variety of tile-based many-core architectures exist, which are interconnected mainly by a scalable 2D-mesh. While the majority of tile-based systems are homogeneous from a global perspective, the tile-internal structure is heterogeneous. Nevertheless, there also exist tile-based architectures combining different tile types like the Invasive Computing architecture [40].

### 2.1.4 Heterogeneous Architectures

Heterogeneous systems complement the trend of scaling homogeneous architectures with increased customization and application-centric designs. Especially the past decade revealed that diverse and ever-growing application scenarios benefit from a heterogeneous combination of different processing elements [66, 67]. Cores of different sizes, complexity (e.g., ISA, TLP/ILP, and in-/out-of-order execution), and performance are combined with application-specific accelerators. A classification into three types of heterogeneous architectures can be made [68]: 1) a mixture of heterogeneous cores with different performance or power efficiency, 2) multiple homogeneous cores plus hardware accelerators, and 3) a hybrid composition of the types mentioned above.

One example of the first type of heterogeneous systems is the IBM Cell processor, known of the Sony PlayStation 3 [69], which combines one big core alongside eight weaker cores. Probably the most prominent example is Arm's big.LITTLE architecture which combines several *big* cores for maximum compute performance with *little* cores to enable power efficiency [56, 70]. Its successor technology DynamIQ [57] enhances the flexibility and scalability as it allows to build and configure heterogeneous clusters of *big* and *little* cores. Furthermore, the Invasive Computing architecture demonstrates that not only heterogeneous cores but also configurable heterogeneous tiles, including normal Leon3 and specialized Leon3 cores with integrated re-configurable accelerators ($i$-Core) are beneficial. Or even tiles consisting of tightly coupled processor arrays (TCPAs) are proposed [40].

A well-known representative of the second type of heterogeneous architectures is CPU-GPU systems. Different strengths of CPUs and GPUs are combined in a flexible manner [71]. Massively parallel and throughput-critical functions are outsourced to the GPU. Whereas initially, separate discrete CPUs and GPUs were connected via PCIe, their on-die integration became popular in the past decade. AMD first produced these so-called *Accelerated Processing Units* (APUs) in 2011 (AMD Llano) [72]. Similarly, Intel's Sandy Bridge [73] and Ivy Bridge APUs followed in 2012 [74].

Besides CPU-GPU systems, also on-die CPU-FPGA System-on-Chips are widely used to accelerate computations in e.g., bioinformatics [75], finance [76], digital signal processing [77, 78], or machine learning [79]. A recent example is Intel's Agilex System-on-Chip FPGA [37] which can be integrated with a multi-core processor as a 3D system-in-package or attached to the system via a cache coherent interconnect.

Finally, the third type of heterogeneous architecture is prominent in the mobile domain. Besides, Arm's big.LITTLE or DynamIQ CPUs, a multitude of application-specific cores and accelerators are harnessed.

In summary, heterogeneous architectures target the specific needs of particular applications or domains. They are often tailored and optimized to operate under extremely tight thermal, power, and energy constraints. The lessons learned from these systems are applied in this thesis. Integrating dedicated hardware accelerators into a tile-based many-core architecture will enable optimizations unattainable without them.

### 2.1.5 Programming Paradigms

In the single-core era, performance improvements of serial applications were achieved by faster and more efficient single-core processors. However, the advent of multi- and many-core architectures ceased this effect. In 2009, Feng et al. [80] pointed out that "`serial computing is now dead.`" Software development has undergone a significant rethinking to exploit the advantages of parallel architectures and be aware of their challenges. Software architects started to develop applications in a parallel manner by dividing the whole algorithm into *tasks* that can be executed independently on different cores. The so-called *concurrency revolution* has led to a variety of parallel programming models and languages [23, 81, 82]. The most prominent will be presented in the remainder of this section. Moreover, since cache coherence and memory consistency are essential aspects of parallel programming, they will be addressed in this section as well. Furthermore, programming models need to support appropriate inter-process communication and synchronization mechanisms. As these are the focus of this thesis, they will be covered in detail in Section 2.3.

**Shared-Memory Model.** The shared-memory model offers a shared address space to the threads of an application. They access the shared memory concurrently and communicate via loads and stores on shared data structures. Although the programmer does not have to handle data movement, he needs to be aware of race conditions and synchronization of concurrent accesses [83, 84]. Since the shared-memory model assumes cache coherence and certain memory consistency, it needs to be provided by hardware or handled in software explicitly. Thus, e.g., POSIX threads provide dedicated synchronization primitives like semaphores [85]. Another prominent example is the industry-standard OpenMP [86] which offers a higher level of abstraction and eases the programming of parallel architectures.

**Message-Passing Model.** In contrast, the message-passing model does not assume or provide a shared address space. Its main field of application is distributed memory architectures. Nevertheless, it is applicable and beneficial for distributed-shared memory systems as well [84]. Threads communicate by exchanging explicit messages between their corresponding memories or message buffers. Besides not requiring cache coherence, the MP model couples data transfer and synchronization between sender and receiver. The most prominent example is the Message Passing Interface (MPI), which provides a wide range of message-passing primitives [87]. As it exists as a standard library for C, C++, and Fortran, parallel programs must solely be linked against it, and no special compiler support is required. Most architectures support message-passing via a dedicated DMA unit. However, when regarding modern object-oriented programming languages with pointer-based data structures, message-passing gets complicated as these data structures cannot be transferred by a DMA unit [88].

**Partitioned Global Address Space.** The partitioned global address space (PGAS) programming model combines the benefits of shared-memory and message-passing programming. It is tailored for DSM architectures as it is aware of data-to-task locality regarding local and remote memories. The global address space, known of the SM model, is harnessed to improve programmability, while the notion of data locality is added to exploit the DSM architecture [89] better. In its early days, PGAS languages often were an extension of existing programming languages. Examples are Unified Parallel C [90], Co-array Fortran [91], and Titanium [92]. More recently, ad hoc and object-oriented PGAS languages have been developed, such as X10 [93], which will be used in this thesis, or Chapel [94]. Both were developed in the context of DARPA's High Productivity Computing Systems (HPCS) program in 2004 and 2009, respectively.

**Heterogeneous Programming Paradigms.** The rise of various heterogeneous architectures requires appropriate heterogeneous parallel programming (HPP) models and languages to exploit their capabilities. HPP models have a different programming style since the heterogeneous processing elements often operate in a hierarchical host-device relationship. Several companies developed HPP models and languages, which became industry standards [82, 84]. Nvidia presented CUDA (Compute Unified Device Architecture) in 2006 [95]. It allows programming Nvidia GPUs in various languages such as C, C++, or Fortran. The most prominent and versatile HPP standard is OpenCL since it enables the integration of various processing elements like hardware accelerators, FPGAs, or DSPs [96]. Belikov et al. classified these as *low-level* HPP models since a detailed knowledge of the hardware architecture and memory organization is required [84]. A higher level of abstraction is offered by OmpSs, which is based on OpenMP and can be used as an extension of OpenCL [97].

### 2.1.6 Cache Coherence and Memory Consistency

Maintaining cache coherence and memory consistency goes hand in hand with parallel programming models. Multiple shared caches must be kept coherent by enforcing necessary cache invalidation and writeback operations. While coherence can be handled relatively easily in conventional bus-based architectures via a snooping protocol, directory-based coherence for tile-based distributed-shared memory architectures pose more challenges. The Stanford DASH Multiprocessor and Tilera's TILE64 architecture provided hardware-managed global coherence. However, the scalability of such approaches with larger system

sizes is limited due to immensely large coherence directories. The community is unclear whether global coherence is here to stay [98, 99]. Therefore, Srivatsa et al. [8] tackles the scalability issue with a flexible and dynamically re-configurable *region-based cache coherence* mechanism, which provides coherence only to a user-defined subset of tiles. Also, the Intel Xeon Phi [64] allows for maintaining coherence for certain NUMA domains only. Nevertheless, there also exist tile-based many-core architectures that do not provide inter-tile cache coherence like the Intel SCC [60].

Besides cache coherence, the memory consistency model has a considerable impact on the system. While stronger consistency models (e.g., total-store order), when provided by the underlying hardware, ease the programming effort, they sacrifice some performance. In contrast, relaxed consistency models (e.g., acquire-release) require more explicit attention from the programmer while enabling more performant implementations [100].

### 2.1.7 Discussion, Pros and Cons

In conclusion, compute-centric systems exhibit the following advantages and disadvantages. The evolution towards multi- and many-core architectures enabled powerful systems with high nominal compute performance. They have a wide area of application as they are employed for both general-purpose computing and in application-specific scenarios in heterogeneous combinations with accelerators and co-processors. Moreover, well-established, commoditized programming paradigms that ease programmability and comply with legacy code applications exist. Existing high-level languages require less architecture awareness of the programmer.

However, the cost is less efficient exploitation of the nominal compute performance, which is limited by the *von Neumann bottleneck* of data transfers between processor and memory. It can only be compensated by caches to a certain degree. This becomes even more evident when regarding the trend towards tile-based architectures with physically distributed memory and processing nodes. Data-to-task locality challenges and increased data movement diminish the compute efficiency of these architectures. Especially the maintenance of cache coherence and inter-process synchronization and communication is challenging in these highly scalable systems with dozens or hundreds of cores. A trade-off exists between user-friendly, on the one hand, and efficient, architecture-aware programming paradigms, on the other hand. Another evolutionary step might be necessary due to the ever-growing complexity and heterogeneity of computer architectures.

## 2.2 Memory-Centric Architectures

The challenges described above remind us of the famous statement of John Backus from 1977: "`Surely there must be a less primitive way of making big changes in the store than by pushing vast numbers of words back and forth through the von Neumann bottleneck.`" [11] It persisted for roughly 40 years until a major rethinking of computer architectural design had its breakthrough. Instead of building mainly compute-centric architectures (i.e., designing the systems around the processor and hence fetching and caching data close to it), memory-centric designs adopt an opposite approach. They leverage the increased proximity and bandwidth of in- or near-memory computing rather than moving data through the von Neumann bottleneck [19, 16, 28].

This section is structured as follows. First, Section 2.2.1 elaborates on the evolution of in- and near-memory computing and the corresponding memory technologies. Then, a taxonomy of in- and near-memory computing is presented in Section 2.2.2 followed by exemplary architectures in Section 2.2.3. Next, Section 2.2.4 describes the corresponding memory-centric programming paradigms and how cache coherence is handled. In Section 2.2.5, several advantages and challenges of memory-centric systems are discussed. Finally, Section 2.2.6 describes how near-memory computing is interpreted in this thesis.

### 2.2.1 Evolution of In- and Near-Memory Computing

Although the efficiency, process technology, and bandwidth of DRAM architectures continually increased, the widening gap between memory and processor performance could not be closed [28]. Burger et al. [101] (1996) and Patterson [102] (2004) made the off-chip memory bandwidth bottleneck responsible for the limited efficiency of computer architectures.

Independent of these remarks, the origins of in- and near-memory computing date back to the 1960s. Hand in hand with the invention of the DRAM cell by Dennard in 1966 [103] go the first *in-memory* computing approaches. Stone et al. [104] proposed cellular logic-in-memory (CLIM) in 1966. Besides this special-purpose architecture, they also developed a general-purpose logic-in-memory processor in 1970 [105]. Several years passed until the first big processing-in-memory (PIM) hype occurred in the 1990s. Prominent representatives are computational RAM proposed by Elliot et al. [106] in 1992, EXECUBE proposed by Kogge in 1994 [25], and intelligent RAM (IRAM) presented by Patterson et al. [34] at the first PIM research peak in 1996. Moreover, Kogge discussed the potential of PIM architectures in 1997 [107]. All these designs tried to integrate logic in the DRAM memory arrays themselves. However, since DRAM technology is optimized for memory density, it is not easy to embed performant logic into it. Thus, the breakthrough of processing-in-memory in the '90s was impeded by the technical limitations of this 2D integrated approach. Furthermore, a clear vision for user-friendly PIM-programming was lacking at the time [28, 108, 34]. The end of the first PIM research hype was sealed by the advent of GPUs, which provide high memory bandwidth, around the turn of the millennium [109]. Nevertheless, this trend only postponed the problem of limited off-chip memory bandwidth.

The revival of in- and near-memory computing occurred in the past decade owing to the seminal innovations in DRAM and chip packaging technology. The first step to overcoming the limited off-chip bandwidth is the invention of 2.5D integration techniques. In contrast to 2D integration, which tries to embed logic into memory, 2.5D integration places processing elements and memory on the same silicon interposer [110]. Thereby costly off-chip memory accesses were reduced. Nevertheless, the breakthrough was achieved by the 3D-stacking technology [28]. Instead of only placing processing and memory chips on the same die, 3D integration stacks them vertically on top of each other. This technique was enabled by the game-changing invention of through-silicon vias (TSVs) in 2009 [30]. Such architectures are commonly referred to as 3D-stacked *processing-in-memory* (PIM). Nevertheless, *processing-near-memory* or *near-memory computing* would be a more precise description since the processing elements are integrated *near* and not *in* the memory arrays. However, due to the high memory bandwidth and low access latencies, PIM is the most commonly used term. [3]

[3] In contrast to its original meaning, the meaning of the term processing-in-memory shifted to this interpretation. A clarification will be made in the taxonomy in Section 2.2.2.

Based on the TSV technology, Micron developed the Hybrid Memory Cube (HMC) in 2011, which integrates several memory layers vertically stacked upon a dedicated logic layer [31, 111]. The second example of 3D integrated memory is the second generation of High Bandwidth Memory (HBM) [112, 113]. Although the HBM2 does not contain integrated processing elements, it is still often listed in the context of in- and near-memory computing due to its high bandwidth.

In summary, these innovations in memory technology, especially the HMC and HBM, have been the trail-blazers of the recent near-memory computing research hype. Both technologies enable combining sub-modules of different process technologies on one chip [28]. Nevertheless, challenges remain like the complex manufacturing and testing of 3D-stacked architectures, as well as dissipation of thermal energy out of the interior of the chips [114].

Besides DRAM systems, novel non-volatile memory experiences increasing attention. In 2013, AMD research asserted that these new memory technologies can "`fundamentally change the landscape of the future computer architecture design`" [115]. Ferroelectric memory (FeRAM) [116] and magnetic memories (MRAM) [117] provide comparable latencies to SRAM and DRAM while they even improve efficiency and power consumption [118]. Nevertheless, to date, they are only employed in systems with low storage requirements since these technologies are still more challenging to scale and stack [119, 118].

### 2.2.2 Taxonomy

This section presents a taxonomy of in- and near-memory computing which is illustrated in Figure 2.2. The classification distinguishes three dimensions: (1) the location relative to memory, (2) the type of function implementation, and (3) the application scenario.

The first dimension distinguishes between *in-memory* and *near-memory* approaches. The former, also known as "processing-in-memory" (PIM), are nowadays typically implemented as 3D-stacked memory devices [28, 120, 121], where logic and memory layers are stacked vertically on top of each other. In contrast, near-memory approaches place the processing node as close as possible to the memory controller, typically behind the cache hierarchy, but still physically separated from the actual memory device [122]. This makes it easier to integrate and allows compatibility with a broader range of architectures and memory technologies. However, it faces a performance loss due to the limited off-chip bandwidth compared to in-memory solutions [25]. Still, both promise higher bandwidth and lower latency compared to classical compute-centric "far-from-memory" approaches via the interconnect and cache hierarchy.

The second dimension distinguishes the type of function implementation between dedicated in- or near-memory hardware accelerators and software programmable logic or cores [123]. A programmable core provides higher flexibility through its general-purpose usability. In the past, complex in-memory cores were difficult to integrate due to the conflicting technological optimizations for fast logic and dense memory [108]. It got easier through 3D-stacked PIM integration. In contrast, despite a hardware accelerator being often limited to a small range of applications, it promises lower power consumption and more efficient operation. The variety of accelerators is huge and encompasses a range from fine-granular operations (e.g., atomic operations) to coarse-grain accelerators of bigger functional blocks [31, 123, 28]. Which of both types of function implementation is preferable depends on the use case and additional constraints.

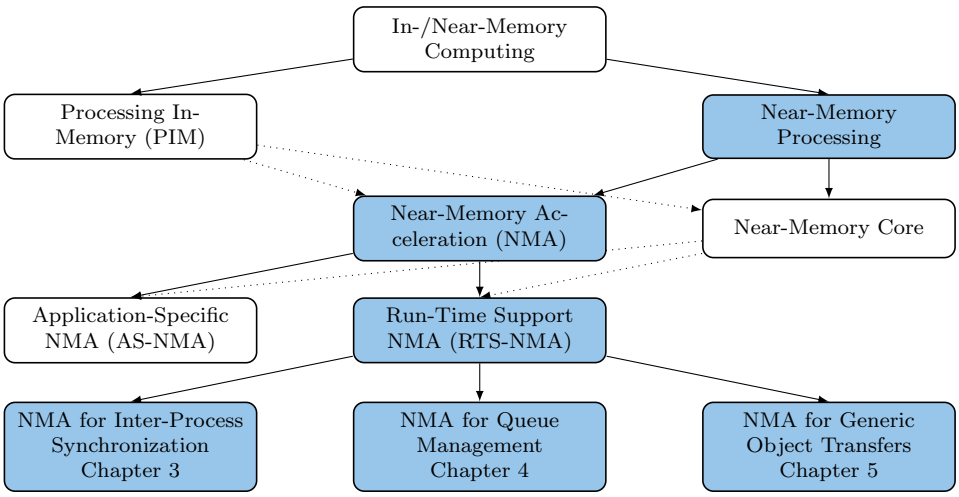



**Figure 2.2: Taxonomy of in- and near-memory computing.** Three dimensions of classification: (1) location w.r.t. memory, (2) type of function implementation, and (3) application scenario. The taxonomy is highlighted for run-time support near-memory acceleration (RTS-NMA). The dotted arrows indicate the same differentiation for PIM, and the near-memory core.

As the third dimension of classification, in- and near-memory approaches can be subdivided into accelerating application-specific tasks (AS-NMA [4]) and accelerating run-time support functionalities like operating system routines for inter-process communication or synchronization (RTS-NMA). While AS-NMAs are very specific and lead to a huge variety of architectures, RTS-NMAs are prone to profit a wide range of applications and systems since they are more or less opaque to the programmer. Nevertheless, these accelerators can also be used in an application-specific manner when called from the user-space.

This thesis focuses on run-time support near-memory acceleration in the context of tile-based many-core architectures. The upcoming chapters target inter-process synchronization (Chapter 3), queue-based message-passing inter-process communication (Chapter 4), and graph copy based PGAS inter-process communication (Chapter 5), respectively.

### 2.2.3 In- and Near-Memory Computing Architectures

A huge variety of in- and near-memory computing architectures exist, primarily employing the Hybrid Memory Cube (HMC) or High Bandwidth Memory (HBM). This section presents several approaches focusing on their interaction between the 3D-stacked memories and the system's processing elements. Figure 2.3 illustrates three approaches of interaction with the HMC or HBM.

The first type ⓐ leverages the logic layer of the HMC to build application-specific designs. A host CPU outsources operations or functions to the, so to say, logic-enhanced memory. Especially in academia, several architectures have been proposed that vary in functionality and type of integrated processing elements. The functionalities range from fine-granular operations (e.g., atomic-operations) to accelerators of coarse-grained functional blocks [31, 123, 28]. If processors are used, they are often energy-efficient or lightweight processors (LWP) due to thermal limitations of the HMC [114, 124, 120]. In 2016, Yitbarek et al. [125]

[4] Note that although the term near-memory accelerator (NMA) is used, the same differentiation applies to in- and near computing in general.

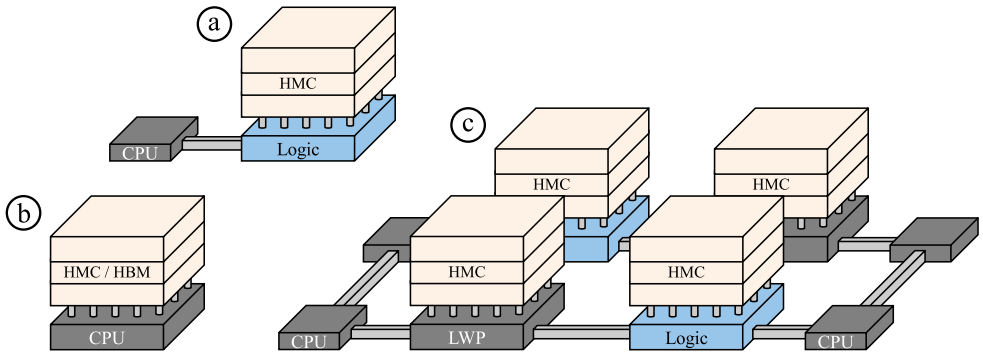



**Figure 2.3: Memory-centric architecture variants employing the HMC or HBM.** ⓐ Host-CPU with logic-enhanced HMC. ⓑ HBM or HMC used as stacked high bandwidth memory. ⓒ Heterogeneous architecture incorporating CPUs and HMCs enhanced with logic or lightweight cores (LWPs).

explored several HMC architectures in a simulation environment. Furthermore, application-specific in- or near-memory accelerators have been presented for simple memory-intensive tasks [126] but also for more complex scenarios. Examples are matrix multiplications [127], training of neural networks [128], and processing of irregular graph data structures [129, 130, 131]. Many of these examples are implemented in the HMC.

The second widespread approach ⓑ is found in many commercial products which use the HBM2 or the HMC as high bandwidth memory. Basically, the HBM2 or HMC is used as a better memory technology in a, so to say, compute-centric manner. In this case, the logic layer of the HMC serves as a 3D-stacked memory controller and interface to the host processor. An example of this type of HMC usage is Intel's MCDRAM, which is employed in e.g., Intel's Xeon Phi Knights Landing architecture [38]. Moreover, HMCs can be connected externally to Intel's Stratix 10 MX and Arria 10 FPGAs [132, 133]. Similarly, the HBM2, which is a JEDEC standard, is employed in many recent devices. It is used as high bandwidth memory in e.g., Nvidia's GA100 GPU [35], Intel's Stratix 10 MX [36], and Agilex FPGAs [37].

A third alternative is to build heterogeneous systems consisting of several processors and 3D-stacked logic-enhanced memories. Such a design has similarities to a heterogeneous tile-based many-core architecture with distributed-shared memory. However, such distributed 3D-stacked architectures are rarely seen to date, most likely because of two reasons. First, as described above, most memory-centric systems employ the HBM2 or HMC combined with multi-core processors. The scalability and bandwidth limits are not yet reached, so memory-centric tile-based architectures are not yet necessary. Second, a specifically memory-centric programming paradigm that would help exploit such architectures better is still missing [28].

In summary, while many academic approaches leverage logic-enhanced 3D-stacked HMC memories, most commercial products use the HBM2 or HMC as the next high bandwidth memory technology node, often in a compute-centric manner. The following section elaborates on the reasons for this as it discusses memory-centric programming approaches.

### 2.2.4 Programming Paradigms and Cache Coherence

Despite the revival of in- and near-memory computing architectures in the past decade, the challenge remains of how to reap the benefits of increased memory proximity and bandwidth. Therefore, this section addresses memory-centric programming and, closely tied to it, cache coherence.

**Memory-Centric Programming.** When regarding the architecture types presented Figure 2.3, different implications for their programming arise. Type ⓐ designs often use dedicated API calls to offload functions to the application-specific logic of the HMC [29]. While this enables efficient architecture exploitation, it creates tremendous programming effort due to the customized functionality. In contrast, the second type ⓑ does not require changes to the programming model since, in this case, the HBM2 or HMC are used as better memory technology in a compute-centric manner. The third type ⓒ is a heterogeneous architecture with several processors, memories, and near-memory computing units. In the compute-centric domain, e.g., CUDA and OpenGL exist to incorporate GPUs, and OpenCL or AMD's HSA concept help to cooperate with various accelerators and co-processors. Nevertheless, to date, a specifically memory-centric programming paradigm is still missing [28]. Thus, in- and near-memory computing units are often integrated into existing programming models.

A game-changer could be a memory-centric operating system as envisioned by Faraboschi et al. [134]. They argued for rethinking traditional operating systems as they do not exploit the benefits of memory-centric systems. They proposed to offload OS routines like memory allocation or thread synchronization closer to memory. Others, like Hsieh et al. [135], envision an architecture- and utilization-aware task offloading and mapping mechanism which relieves the programmer as much as possible. They propose identifying candidates for near-memory processing at compile-time and decide dynamically at run-time which of these to offload. In summary, it would be beneficial to outsource library and operating system routines as these profit a wide range of applications while being opaque to the application programmer. Therefore, this thesis targets run-time support near-memory acceleration in a heterogeneous accelerator-enhanced tile-based many-core architecture.

**Memory-Centric Cache Coherence.** Maintaining cache coherence and memory consistency between the standard processor cores and the in- and near-memory processing units goes hand in hand with the applied programming model. There exist several approaches that differ in their granularity, hardware support, and performance implications. They have in common that the near-memory processing units are mostly located behind the last level cache so that their operations bypass the cache hierarchy.

The first approach is to include the in- or near-memory units into the hardware coherence protocol. Necessary cache writeback and invalidation commands are handled automatically in a fine-grained manner. Although this approach would be opaque to the programmer, it may result in a lot of coherence traffic, hindering the near-memory units from unfolding their potential [123]. If no cache-coherent interconnect between the host processors and the near-memory units exists, the coherence handling will become a software task.

An alternative approach is to grant exclusive access to the near-memory units during their operation. The corresponding data sets would be marked as non-cacheable to guarantee undisturbed operation [136, 121].

A third approach is the coarse-grained LazyPIM proposed by Boroumand et al. [137]. The idea is to speculatively execute the near-memory operations concurrently with normal operations. Upon completion, a batched coherence message is sent to the processors. Only if cache conflicts occur the near-memory unit has to undo its operations, update the conflicting cache lines, and re-execute its operations. Thus, if no conflicts occur, the near-memory units can operate undisturbed without fine-grained coherence messages. The idea has some similarities to transactional memory.

### 2.2.5 Discussion, Pros and Cons

In conclusion, memory-centric architectures have several advantages but suffer from significant challenges still. In- and near-memory computing mitigates the data-to-task locality challenge as computations are performed as close to memory as possible. The von Neumann bottleneck between processor and memory is tackled as data movement and off-chip accesses are reduced. Consequently, the energy consumption and communication latency are lowered as well, increasing the system's overall efficiency [34]. The huge field of application reveals the potential of in- and near-memory computing [29]. For example, the HMC is employed as standalone logic-enhanced memory, as part of a heterogeneous many-core architecture, or simply used as a main memory with high bandwidth.

However, technological challenges still exist, such as limited cooling capabilities inside 3D-stacked memory devices, reduced yield, and more complicated testing [138, 139]. However, a more severe challenge is the lack of a specifically memory-centric programming paradigm. The existence of many custom HMC designs and the frequent use of the HBM and HMC as high bandwidth memory technology in a compute-centric manner point to a missing vision for memory-centric programming. Nevertheless, lessons learned from decades of computer engineering teach that ease of use is crucial for the assertiveness of a trend. Finally, the near-memory aspect might suffer when scaling memory-centric designs to incorporate multiple memories. Most benefits of in- and near-memory computing rely on the proximity and bandwidth inside or near the memory, Thus, when inter-memory accesses occur, a bandwidth bottleneck arises. Moreover, the lack of MMUs or virtual memory further impedes today's efficient use of multiple memories [28]. Thus, a vision for in- and near-memory computing on architectures with distributed shared memory is required.

### 2.2.6 Interpretation of Near-Memory Computing in This Thesis

Based on the elaborations of the previous sections, this section explains how near-memory computing is interpreted in this thesis. Run-time support near-memory accelerators (RTS-NMA) will be applied to optimize inter-process communication of tile-based many-core architectures with distributed shared memory. Thus, the memory-centric paradigm will be embedded into a compute-centric architecture and programming model. The state of the art of inter-process communication (IPC) will be presented in the next Section 2.3.

As mentioned above, distributed shared memory architectures like the employed tile-based many-core systems add a new dimension of complexity to *nearness*. Processing *near* to a particular memory is equivalent to being *far* from any other memory. Data-to-task locality is different for local versus remote memory accesses, potentially diminishing the *near-memory* aspect. Thus, for tile-based many-core architectures, this thesis interprets near-memory computing as *being on the right side of the interconnect.* Tile-local memory

accesses are declared as *near-memory* while remote inter-tile memory accesses are defined as *far-from-memory*. From the view of local cores, the accelerators might not be *closer* to the tile-local memory as the local cores themselves. However, since any local or remote core can call the accelerators, they are much closer to memory than all remote cores. This is the interpretation followed in this thesis. Therefore, the term *remote near-memory acceleration* (RNMA) applies. The operations are not executed remotely via the NoC and cache hierarchy but outsourced to the respective accelerator near the memory in which the respective data structure resides.

This more abstract view of near-memory acceleration is independent of the actual memory technology. Hence, no 3D-stacked memories are employed but rather standard DRAM and SRAM memory. The main focus is on increasing data-to-task locality and memory bandwidth for remote inter-tile accesses.

## 2.3 Inter-Process Communication

Herlihy and Shavit called multiprocessor programming an art since exploiting the ever-growing parallelism of multi- and many-core architectures is one of the most challenging topics of computer science [140]. Parallel applications split their tasks into several threads or processes that cooperate concurrently. In general, they require inter-process communication (IPC) to coordinate their work and exchange information which includes data transmission, notification, and synchronization.

This section presents the fundamentals of inter-process communication with a particular focus on tile-based many-core architectures. Thus, Section 2.3.1 describes the shared-memory and message-passing communication model [5] alongside their respective requirements and employed data structures. Upon this basis, the remainder of the section introduces the state of the art in inter-process synchronization (Section 2.3.3), queue-based message-passing inter-process communication (Section 2.3.4), and graph copy based PGAS inter-process communication (Section 2.3.5).

### 2.3.1 Communication Models

Inter-process communication is performed either implicitly via shared memory or explicitly via message passing [141]. Both models, with their strengths and weaknesses, will be presented in the following.

**Shared-Memory Communication.** In the shared-memory model, processes communicate implicitly via loads and stores to the shared memory address space. No dedicated messages have to be copied between the participants. However, since they operate concurrently on shared data structures, a particular need for synchronization exists. Besides, shared-memory communication is, by default, a polling interface so that the receiver has to poll on a synchronization variable to recognize newly available data.

Furthermore, from a programmer's point of view, the semantics of the shared-memory model is independent of the data's physical location. Thus, it is a relatively architecture-unaware communication model that does not have a notion of data-to-task locality. When

[5] The communication model is closely related to the programming model. While the programming model is an abstraction of the computer architecture, especially of the memory organization, the communication model is primarily concerned with the inter-process communication.

applied to tile-based many-core architectures, data locality becomes of much higher importance for inter-process communication. Efficient inter-process synchronization and cache coherence are crucial to facilitate shared-memory communication on those architectures. Srivatsa et al. addresses inter-tile cache coherence in several publications that I co-authored [8, 9, 10] but which are beyond the scope of this thesis. In contrast, inter-process synchronization is addressed in Chapter 3 and the respective related work is covered in Section 2.3.3.

**Message-Passing Communication.** Message-passing communication is based on sending explicit messages between communicating processes. Typically a sender transfers a message to a receiver and notifies it of the completed transfer. A dedicated DMA unit usually supports the data transmission. The notification (e.g., an asynchronous event) can be performed by triggering an interrupt or spawning a task. Moreover, the receiver itself may be polling for new incoming messages. Since the participants are not operating on shared memory concurrently, no synchronization is required other than the notification. Often dedicated message-passing buffers or FIFO queues are leveraged to facilitate unidirectional communication.

From a programmer's point of view, message-passing communication is aware of the distributed (shared) memory. Thus, it fits the inter-process communication on tile-based many-core architectures. A corresponding contribution is presented in Chapter 4 while the respective related work is described in Section 2.3.4. Moreover, there are more complex data structures in modern object-oriented programming languages, such as object graphs, that a DMA unit cannot transfer. Thus, Chapter 5 proposes a contribution that enhances graph copy-based inter-process communication in the PGAS programming paradigm. The respective related work is covered in Section 2.3.5.

**Hybrid Communication.** A hybrid combination of shared-memory and message-passing communication can be found e.g., in the PGAS programming model. Typically, shared-memory communication is used inside a tile (intra-tile) among the cores that share a tile-local memory, while message-passing is employed for inter-process communication between tiles (inter-tile). This combines the benefits of both approaches, namely ease of programmability and more architecture-awareness.

### 2.3.2 Notification Mechanisms

Both inter-process communication models require a mechanism to notify the receiver that new data has been provided. The receiver notification can be implemented in three ways.

The first variant is *receiver polling*, i.e., the receiver checks actively and regularly if new data is available. It might be busy waiting for the release of an occupied spinlock or checking a shared status variable. This type of notification mechanism is primarily found in shared-memory communication. Nevertheless, message-passing approaches may receive notifications by observing the fill level of a message queue.

The second approach is based on *interrupts*. Interrupts relieve the process from polling since the waiting process may go to sleep while another task can be executed in the meantime. However, interrupting a running task leads to a context switch, and thus, the receiver process also consumes time. Sometimes, wasting time through polling might be more beneficial [141].

The third option is *signaling and waiting*, e.g., based on condition variables or barriers. Typically it is a combination of the two previous approaches. The receiving process informs

the kernel that it is waiting and goes to sleep. It does not have to check for new data since it is woken up by the notifying task. The third option is closely tied to *task scheduling* since it requires kernel involvement to schedule a task that notifies the receiver process. Thus, task scheduling is essential in the context of inter-process communication. Utilizing a task to notify the receiver can be used for shared-memory and message-passing communication.

In large multi- and many-core systems, efficient task scheduling is important to avoid a bottleneck for inter-process communication. Although task scheduling is beyond the scope of this thesis, several hardware-based scheduling approaches are covered in Section 2.3.4 as they share similarities to the hardware-managed queue approach proposed in Chapter 4.

### 2.3.3 Synchronization Primitives and Concurrent Data Structures

Synchronizing the concurrent access of multiple processes to shared data structures is crucial for shared-memory communication. When at least two processes operate on the same shared variable or data structure concurrently, certain operations require temporarily exclusive access, i.e., atomicity [140]. Atomicity for the so-called *critical section* (CS) can be established by different classes of synchronization primitives. This section presents the state of the art in inter-process synchronization primitives with a particular focus on tile-based architectures. Additionally, prominent concurrent data structures will be covered, which form the basis of shared-memory communication.

**Lock-Based Synchronization.** The first type is mutual exclusion based on *locking*. The lock is acquired before entering the critical section and released after leaving it. Other threads cannot enter the critical section if the lock is contended. Several software approaches exist to locking, like Peterson's Lock, the filter lock, or Lamport's bakery lock. However, they do not scale well. Therefore, today's MPSoCs typically provide hardware support in the form of the atomic test-and-set (`TAS`) ISA extension. It can be employed to build simple spinlocks, i.e., a thread actively checks ("spins") if the lock variable is occupied until it can be acquired. The atomicity of the read-modify-write cycle is ensured by the `TAS` instruction. In 1991, Mellor-Crummey and Scott proposed the more advanced MCS-lock [142] which compensates for the limitations and performance bottlenecks of various simple, ticket, and queue-based spinlocks [143, 144, 145, 146]. The MCS-lock is an efficient software-implemented locking mechanism that requires hardware support of the atomic `TAS`. Moreover, it can unfold its entirety of features, like FIFO ordering and starvation freedom, when additionally the atomic compare-and-swap (`CAS`) instruction is available.

Semaphores are a generalization of mutual exclusion as more than two threads can be synchronized [147]. A semaphore is based on a shared counter initialized to the number of threads that can concurrently enter the critical section. The counter needs to be decremented or incremented atomically when the semaphore is acquired or released, respectively. Besides implementing semaphores and mutexes in a spinning/polling manner, they can be realized in a *blocking* fashion. It is advantageous for longer critical sections since the waiting thread goes to sleep instead of waiting actively and is woken up by the releasing thread [147]. Such functionality is provided, e.g., by condition variables [140]. Further examples of lock-based synchronization primitives are monitors or barriers. Moreover, a prominent example of a library that implements the primitives mentioned above is the POSIX thread library [85].

**Lock-Free Synchronization.** The second class of synchronization primitives is *lock-free*, i.e., locking of the critical section is avoided [140]. The still existing need for atomicity can be satisfied by the atomic `CAS` or `load-linked/store-conditional` (`LL/SC`) instructions. They have a conditional read-modify-write cycle so that they only write the modified value to memory if the read value does not change in the meantime. These universal atomic primitives enable transforming any lock-based implementation of a data structure into a lock-free equivalent [148, 143, 149, 150]. However, the authors of [143] admit that lock-free implementations may become quite complex.

Moreover, especially the `CAS` instruction, which is provided as an ISA extension in most systems, faces a critical limitation which may result in the *ABA problem*. The ABA problem can occur when multiple threads access the same memory location concurrently. One thread T1 reads the value A. Then, before T1 performs the `CAS`, another thread T2 modifies the value to B and back to A. When T1 finally executes the `CAS`, it wrongly appears as if nothing changed, and the `CAS` succeeds, although the assumption "nothing changed" was violated. The ABA problem is prevalent in many lock-free `CAS`-based implementations of concurrent data structures. It can be avoided with e.g., a multi-`CAS` operation or the `LL/SC` instruction. However, the multi-`CAS` does not exist in modern processors, and the `LL/SC` is, at least, not present on tile-based architectures.

Lock-free synchronization primitives enable the implementation of *non-blocking data structures*. Thus, threads always progress since they do not have to wait for a contender to leave the critical section. Due to their complexity, it is beneficial to obtain non-blocking variants of concurrent data structures from libraries.

**Special-Purpose Synchronization.** Besides lock-based and lock-free primitives, there exist so-called *special-purpose* atomic operations that outsource the entire critical section into dedicated hardware. A prominent example is the conglomerate of `fetch-and-`$\Phi$ operations, where $\Phi \in \{$`increment, decrement, add, sub, and, or, ...`$\}$ [148, 143, 149, 150]. While these, similar to the `CAS` and `LL/SC`, operate on one memory word, there is a demand for more complex and composed atomic primitives. A trend in this direction is the concept of *transactional memory* [151] which executes atomic blocks speculatively and commits all updates at once. In case a conflict occurs, the transaction is aborted. To date, hardware support for transactional memory did not experience a major breakthrough.

Nevertheless, more complex atomic operations to support concurrent data structures efficiently are desirable. The following paragraphs will discuss its reasons with a particular focus on tile-based architectures.

**Discussion of Synchronization Primitives.** A *lock-based* variant relies on locking the critical section so that no concurrent accesses of other threads can interfere. Whereas by design, the critical section inside the locked region is *retry-free* the lock acquisition itself is not since it is unclear when it will be successfully acquired. An operation is defined as retry-free if the read-modify-write cycle is non-conditional, i.e., no interfering concurrency on the data structure can occur that would require re-executing the operation. Furthermore, it is not wait-free since there is no upper bound for the number of retries until lock acquisition can be given. In contrast, a *lock-free* implementation is not retry-free since the read-modify-write cycle of the universal atomic primitives `CAS` or `LL/SC` is conditional. Hence, the operation must be repeated until it is successful. In the best case, this can be one try if no concurrent access interfered in the meantime. However, the average case depends heavily on the con-

currency, while even starvation cannot be excluded in the worst-case. Thus, wait-freeness is not guaranteed either. A *special-purpose hardware* implementation of more complex atomic operations promises the best performance. It is retry-free by design since the entire critical section is executed atomically in hardware. However, the fixed functionality of the dedicated hardware functionality limits its use case. The `fetch-and-increment` operation is one of few such dedicated implementations present as an ISA extension of CPUs.

**Synchronization Primitives on Tile-Based Architectures.** Since tile-based many-core systems are the target architecture of this thesis, this paragraph presents existing approaches applicable for inter-process synchronization between tiles. Thread synchronization is more challenging on these tile-based architectures with distributed shared memory. First, the prevalent NoC interconnect does not inherently support atomic memory accesses from the view of all cores, as a temporarily exclusive usage of the interconnect is not feasible. Thus, the inter-tile support for atomic operations is often omitted. This is different from providing local atomic operations since they typically achieve their atomicity via locked bus accesses. [6] Second, non-uniform memory access (NUMA) properties of distributed shared memory architectures require proper handling of data-to-task locality.

Research in the field of inter-process synchronization on tile-based architectures has two directions. On the one hand, efficient lock implementations are explored [152, 144, 153, 154] on architectures without support for the general-purpose `CAS` or `LL/SC` instructions. The authors of [144, 153] presented efficient lock-based synchronization for a tile-based distributed shared memory system. They started with optimizing simple and ticket spinlocks [153] before they advanced to MCS-locks [144]. A remote MCS-lock can be acquired with $\mathcal{O}(1)$ NoC operations since it spins on local variables only. They outsource the local polling to a hardware loop [144]. Moreover, they employ `fetch-and-inc/dec` operations to implement their semaphores and ticket spinlocks [153]. Nevertheless, their hardware lock implementation operates purely on special globally addressable registers present in their hardware synchronization module. They do not operate on normal memory and also do not support lock-free primitives. A similar idea is followed by the authors of [154] who optimize locking in a physically distributed memory environment. Their unique selling point is the avoidance of head-of-line blocking of independent synchronization requests.

On the other hand, lock-free synchronization capabilities in the form of the `CAS` exist in some many-core architectures [155, 148, 143, 150, 145]. A prominent example comes from Tilera (now Nvidia) as they provide the `CAS` instruction on the Tile-GX platform [155]. Another example of a tile-based architecture with `CAS` support is used by the authors of [148]. They use one core per tile and a 2D-mesh NoC interconnect. They propose a rather complex combination of several optimizations to minimize the `CAS` operations on the main memory. An in-cache `CAS` implementation is coupled with a write-invalidate coherence policy and a load-exclusive policy extension. Additionally, they suggest tackling the ABA problem with a hardware-based serial number extension of the `CAS` and envision the `fetch-and-increment` operation to implement efficient counters.

In summary, most related inter-process synchronization approaches on tile-based architectures either try to optimize locking mechanisms or analyze only lock-free implementations [156, 157]. However, special-purpose primitives like the `fetch-and-increment`

[6] Note that locked bus accesses should not be confused with a lock-based implementation. Locked bus accesses ensure atomic execution of a sequence of bus operations required for e.g., the `TAS`, `CAS`, or `fetch-and-`$\Phi$ instructions.

instruction do rarely exist for remote inter-tile operations [152, 148]. Moreover, hardware support for more complex and composed atomic operations does not exist in state-of-the-art tile-based many-core architectures.

**Concurrent Data Structures.** Concurrent or shared data structures are the prevalent means of shared memory communication and synchronization [22, 140]. Many prominent examples exist, such as shared counters, stacks, queues, pools, linked lists, hash tables, search trees, priority queues, and others. While these examples vary in their complexity, they have in common that their concurrent implementation is much more complicated than their sequential counterparts, particularly if a lock-free implementation is desired. While shared counters are the simplest and only require atomic increment and decrement operations, others are more involved and often require a sequence of dependent operations. Chapter 3 of this thesis focuses on three of these data structures: shared counters, stacks, and queues. The sequel will briefly cover their properties, operations, and atomicity requirements.

The concurrent `shared counter` can be used to build e.g., semaphores, barriers, or tickets. Atomicity has to be guaranteed for incrementing or decrementing the counter value. It can be implemented lock-based, lock-free using the `CAS` operation, or via the special-purpose atomic operations `fetch-and-increment` and `fetch-and-decrement`, or more general `fetch-and-add` and `fetch-and-sub`. The different implementations have varying complexity and performance implications which will be analyzed for tile-based architectures in Chapter 3. The more complex examples (i.e., queue and stack) are illustrated in Figure 2.4.

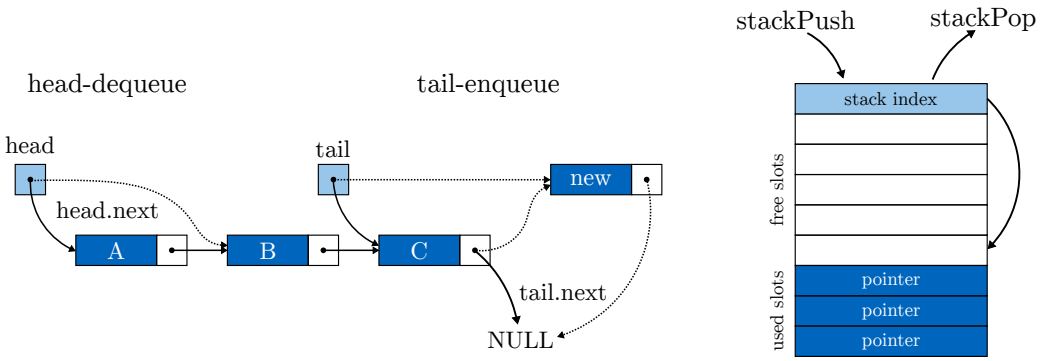



**Figure 2.4: Examples of concurrent data structures.** A concurrent queue implemented as a linked list (left) and a concurrent stack laid out in memory as a bounded buffer (right). Although the stack is presented as a contiguous block it could as well be implemented as a linked list.

The **concurrent linked-list based queue** requires atomic tail-enqueue and head-dequeue operations. During the `tail-enqueue` operation, the `tail.next` pointer has to be updated atomically and only if no concurrent enqueue operation succeeded in the meantime, i.e., if `tail.next` still points to `NULL`. Then, the tail pointer can be set to the new element atomically if no other enqueue operation did so in the meantime. Similarly, during the `head-dequeue` operations, the head pointer has to be updated atomically, which succeeds if no concurrent dequeue occurs in the meantime. Although the queue is arranged as a more generalized linked list where the queue elements can be dispersed in memory, this

is unnecessary. It could also be implemented as a bounded buffer at a contiguous chunk of memory.

The third example is the **concurrent stack** which is laid out in memory as follows. The first word holds an index that points to the first free stack slot, while the following words hold the pointers to the actual stack slots. Although the stack is presented as a contiguous block, it could also be implemented as a linked list. The stack is accessed by atomic `stackPush` and `stackPop` operations. The `stackPush` atomically increments the stack index before writing the element pointer to the stack slot corresponding to the incremented index value. Similarly, the `stackPop` atomically decrements the stack index before reading from the stack slot corresponding to the non-decremented index value. Thus, the critical section consists of incrementing or decrementing the index value, respectively.

**Summary.** In conclusion, several approaches to inter-process synchronization of shared-memory communication exist. Besides lock-based and lock-free implementations of concurrent data structures, there are a handful of special-purpose primitives. However, the inter-tile synchronization support is rare and more challenging. Thus, it will be addressed in this thesis.

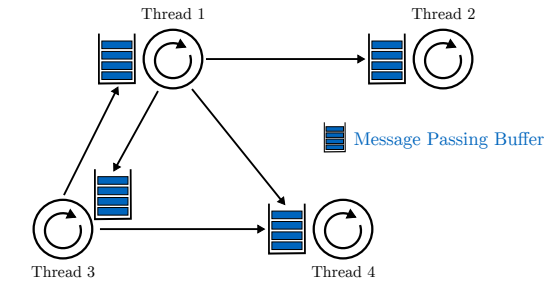
### 2.3.4 Queue-Based Message-Passing Communication

In contrast to shared-memory communication, where the data exchange between threads is implicit, message-passing communication is based on explicit messages. The communication typically consists of data transfer and subsequent notification to inform the receiver when the data can be processed. Queues are suitable, beneficial, and often used data structures for such communication. They retain the order of elements (first-in-first-out) and function as a buffer that decouples the sender and receiver to a certain extent. This section presents state of the art in queue-based inter-process communication with a particular focus on tile-based architectures.

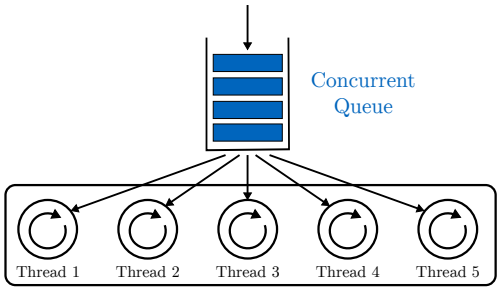
Figure 2.5 highlights three representative use-cases of queues. In Figure 2.5a, queues are used as message-passing buffers between multiple threads to facilitate their communication. In Figure 2.5b, a concurrent multi-producer multi-consumer queue is utilized for task scheduling where several threads receive their jobs from a shared queue. A scenario where each thread feeds from its own queue (multi-producer single-consumer) is also conceivable. Moreover, queues find their place between different stages of pipelined or streaming applications (Figure 2.5c) or to manage incoming and outgoing requests in network stacks.

As queues, in general, are not a new phenomenon, there exist quite many scalable software queue implementations like the well-known Michael-Scott queue [158]. However, since their main field of application is standard shared memory systems with a centralized memory, they are optimized for solely handling pointers to queue elements. There is no necessity to copy the actual element payload since all threads operate on the same physical memory. On such architectures, there further exists research into hardware acceleration of general-purpose queues [159, 160]. Furthermore, on various architectures, hardware accelerated queues are leveraged to support task scheduling [161, 162, 163, 164, 165]. Again, they are limited to handling only element references or small task descriptors.

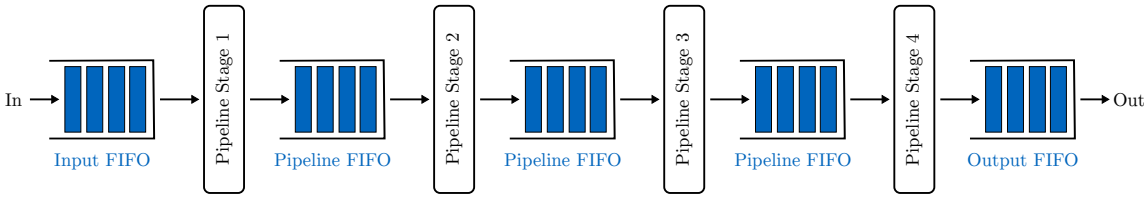
**Classification of Related Hardware Queue Approaches.** In the following, several hardware-accelerated queue management approaches will be presented since they are related to the contribution of Chapter 4. The classification of those approaches has two primary di-




**(a)** Queues as message-passing buffers.




**(b)** A concurrent queue as a thread pool.




**(c)** Queues between several stages of a pipelined application.

**Figure 2.5: Application scenarios for queues.** Three representative use-cases of queues.

mensions. The first dimension differentiates between hardware-accelerated queue concepts exclusively employed for task scheduling or whether the queues can be used in a user-defined general-purpose manner. Although task scheduling itself is beyond the scope of this thesis, those queue approaches are related to the contribution of Chapter 4. The second dimension classifies whether it is a distributed concept applicable to tile-based architectures or not. Furthermore, Table 2.1 provides an overview and summary of additional properties. The column *Data Transfer* denotes whether the concept is also concerned with transferring data or if only pointers are handled while the actual data resides in memory. This might work well in architectures with centralized memory or when solely task scheduling is in focus. However, when communication and data exchange between tiles and cores in tile-based architectures with distributed shared memory is considered, a higher emphasis on data-to-task locality is necessary. The column *Task Invocation* corresponds to the notification mechanism, which can be via task invocation/scheduling (e.g., in the form of a ready queue) or via interrupts or polling. Finally, the column *Q in reg. Memory* distinguishes if the queues reside in regular memory or are implemented in dedicated hardware resources inside the accelerator.

**Hardware Queues for Task Scheduling.** Many queue approaches with hardware support are designed for task scheduling and distribution. They cannot be used as general-purpose queues with arbitrary user-defined elements but are typically limited to handling task descriptors. Research on hardware-assisted scheduling is presented for classical shared-memory architectures and for tile-based architectures with distributed shared memory.

Carbon was proposed by Kumar et al. (Intel Corporation) in 2007 [161]. Carbon is a dynamic hardware task scheduling approach that enables highly parallel and fine-grained scheduling. It follows a hierarchical approach with a local task unit (LTU) added to each core and a global task unit (GTU) shared among all cores and located on the ring bus. The authors assert that if only the GTU were used, a bottleneck would arise for higher

**Table 2.1: Related hardware queue approaches and their properties.** Table taken from [6] with minor adaptions. The upper half characterizes hardware queues for task scheduling while the lower half addresses general-purpose hardware queues.

| Related Work | | General Purpose | Distr. Architecture | Data Transfer | Task Invocation | Q in reg. Memory |
|---|---|---|---|---|---|---|
| Carbon | [161] | ✗ | ✗ | ✗ | ✓ | ✗ |
| Sharma | [162] | ✗ | ✗ | ✓ | ✓ | ✗ |
| Sanchez | [163] | ✗ | ✓ | ✗ | ✗ | ✗ |
| IsoNet | [164] | ✗ | ✓ | ✗ | ✗ | ✗ |
| TCU | [165] | ✗ | ✓ | ✗ | ✓ | ✗ |
| CAF | [159] | ✓ | ✗ | ✗ | ✗ | ✓ |
| HAQu | [160] | ✓ | ✗ | ✗ | ✗ | ✓ |
| RQ | [166] | ✓ | ✓ | ✓ | ✗ | ✓ |

core counts. The LTU and GTU contain hardware queues that hold a limited number of 32 Byte task descriptors. Nevertheless, Carbon could support an unlimited amount of queues if the GTU's hardware queues are used as a task cache towards the main memory. Another scheduling approach that does not target tile-based architectures has been proposed by Sharma et al. [162]. Their hardware-managed work queue concept consists of a work queue unit that manages a dedicated scratchpad memory (the so-called thread store unit). The combination is added to each core and responsible for scheduling tasks and prefetching the corresponding data. The concept is similar but more proactive than caching.

Also, several hardware-assisted task scheduling or distribution approaches exist for tile-based architectures. Sanchez et al. [163] proposed a dedicated message-passing unit responsible for task distribution in large many-core systems. Their hardware accelerator is added to each core, and its internal queue can hold a limited amount of task descriptors. Another hardware-based scheduling approach has been presented by Lee et al. [164]. The IsoNet concept connects an IsoNet hardware node to each NoC router of the tile-based architecture and interconnects them with an additional IsoNet micro-network dedicated to communication between IsoNet nodes. Each node contains a part of the distributed job queue. The queue length in each node is fixed, and the queue elements are only pointers to the actual task descriptors residing in the main memory. IsoNet balances the load between the nodes and dynamically re-distributes jobs in case of imbalance. However, since the IsoNet micro-network has a fully combinatorial implementation, its scalability to larger system sizes is questionable. A third example of a queue-based hardware scheduler for tile-based architectures has been proposed by Pujari et al. [165]. The thread control unit (TCU) is located in each tile and is responsible for mapping lightweight tasks to the cores. The job queues are integrated into the TCU and hold up to 64 task descriptors (16 Byte in size). The TCU supports a variety of configurable scheduling policies. Moreover, task descriptors can be inserted by any core in the system, either locally via the bus or remotely via the NoC and network adapter. Note that the TCU is leveraged as the hardware scheduler in the prototype platform used in this thesis.

The presented approaches [161, 163, 164, 165] except [162] only manage pointers or task descriptors while the actual data has to be handled separately. Moreover, all concepts locate their queues in dedicated hardware resources, not in regular memory.

**General-Purpose Hardware Queues.** Besides the hardware queue concepts for task scheduling, some hardware-assisted approaches exist to manage general-purpose queues. The Core to Core Communication Acceleration Framework (CAF) proposed by Wang et al. [159] is a dedicated hardware accelerator to manage general-purpose queues in regular memory. It is connected to the on-chip interconnect. The queues themselves are not restricted in number or size, although their size is fixed at the time of queue creation and not dynamically adjustable at run-time. However, CAF only manages pointers to the actual data while the payload must be transferred separately. Furthermore, CAF is a centralized component limiting its scalability to larger systems. Lee et al. [160] proposed the hardware-accelerated queues (HAQu) concept. HAQu is a hardware module that accelerates queue operations on software queues residing in normal shared memory. The queues consist of a queue descriptor (i.e., head and tail pointer and the queue size) and a contiguous buffer for queue elements with a fixed size after queue creation. Nevertheless, HAQu does not target a tile-based architecture. Moreover, the authors acknowledged that HAQu is mainly advantageous for single-producer single-consumer queues. Another hardware-assisted queuing mechanism has been presented by Brewer et al. [166]. They proposed remote queues (RQ) for e.g., the tile-based Cray T3D platform. Their mechanism leverages DMA transfers coupled with `fetch-and-increment` operations to handle the remote data transfer and atomic queue management, respectively. However, this separation creates additional software overhead and network round-trips. Available notification mechanisms are receiver polling or interrupts. The queues can reside in the network interface or normal memory present in every tile.

**Discussion.** In conclusion, various hardware queue approaches exist for both task scheduling and general-purpose use-cases. The focus of the majority of concepts is the optimization of queue management via hardware accelerators. The transfer of the queue elements is mostly neglected or separated from the queue handling. Only handling pointers or task descriptors might be sufficient for classical shared-memory architectures. However, tile-based architectures with physically distributed memory and processing nodes require more attention to data locality of the actual data.

### 2.3.5 PGAS Inter-Process Communication

The previous section addressed inter-process communication via queue-based message-passing. Queue elements are *flat* data (i.e., contiguous chunks of memory) that can be directly copied by a DMA unit provided by most architectures. However, when considering modern high-level programming languages, many challenges arise in tile-based many-core architectures. First, many languages are missing architecture awareness for the physically distributed memory and processing nodes. Second, the architectures miss hardware support for object-oriented data structures like object graphs, in contrast to e.g., a DMA engine for copying flat data. These challenges are further complicated when tile-based architectures omit hardware support for inter-tile cache coherence and memory consistency. The PGAS programming model is a well-known attempt to tackle these challenges. PGAS associates specific memory partitions to each processor or tile and requires a special message-passing-like protocol for inter-process communication between tiles.

Without loss of generality, this thesis considers the object-oriented PGAS programming language X10 [93]. Therein, the entire global address space is divided into separate memory

partitions assigned to the individual tiles. In X10, a tile and its corresponding memory partition are referred to as a *place*. A place is required to be cache coherent, which is in line with the provided intra-tile hardware cache coherence of the used demonstrator platform presented in Section 2.4. However, X10 does not assume or require inter-tile cache coherence between places. Thus, a thread running on a specific tile is only allowed to freely access the memory partition associated with this tile (i.e., tile local memories and caches and its memory partition in the global shared memory). Whenever accesses to another tile's memory partition or the execution of a function on another tile need to be performed, a special programming construct, similar to a remote procedure call (RPC), is required [94, 93]. Thus, in X10, any inter-process communication between tiles/places (i.e., across cache coherence boundaries) is exclusively possible via this so-called `AT statement`.

Although X10 is used as the programming language to showcase the contribution of Chapter 5, the concept applies to other languages as well. Being quite similar to X10, the Chapel language [94], developed by Cray Inc., runs its activities on *locales* (the pendant to X10's places). To communicate with other locales, it employs the RPC-like *on statement* (i.e., the equivalent of X10's `AT statement`). Therefore, Chapel can also benefit from this work when applied to suitable hardware architectures.

In the following, first, the mechanism of the PGAS X10 `AT statement` is explained, which is the graph copy-based inter-process communication mechanism of X10. Thus, the basics of copying object graphs are described next. Based on these fundamentals, two related approaches of X10's inter-process communication between tiles are presented.

**Inter-Process Communication with the PGAS X10 AT Statement.** All inter-process communication between tiles/places in the PGAS programming language X10 is solely possible via the RPC-like `AT statement` construct. The mechanism, implemented in the run-time system, is illustrated in Figure 2.6:

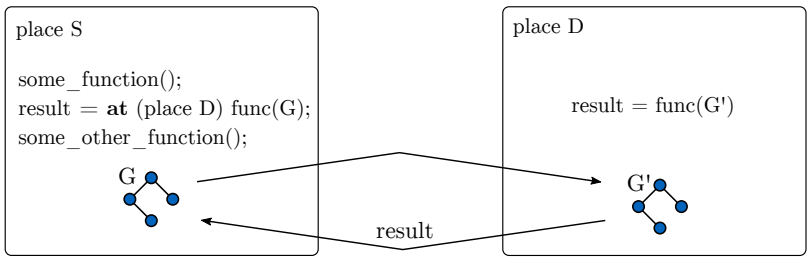



**Figure 2.6: Illustration of the X10 `AT statement`.** The remote execution of thread $S$'s function $func()$ on place $D$ is possible via the `AT statement` which requires the transfer of graph $G$.

A thread running on the initiating sender *place*[7] $S$ executes a function $func()$ on the destination place $D$ and expects back a result. On the remote place $D$, the function $func()$ must be able to access its *lexical environment* (i.e., variables defined outside the `AT statement`). Therefore, when executing the `AT statement`, the run-time system performs the following procedure. An object graph, the so-called *closure* $G$, is implicitly created consisting of all objects and variables defined outside and visible at the point of the `AT statement` on place $S$. The object graph $G$ is then transferred to the destination place $D$, where the function `func()` is executed on it while the sending place $S$ waits for its completion. Once completed, the result is transferred back to the source place $S$ before

[7] Reminder: in X10, a *place* comprises a tile and its corresponding memory partition.

the `AT statement` is finished. Since the transfer of $G$ is an object graph copy, the use of a DMA engine is not directly possible because all copied pointers of $G'$ would wrongly point to the objects in the original graph $G$. Moreover, in general, the objects of $G$ do not reside in contiguous memory. Therefore, a *graph copy* operation with proper pointer adjustment is required.

**Copying Object Graphs.** Since copying object graphs is essential for inter-process communication in the described scenario, the graph copy problem is described in the following.

An *object graph* is a directed graph consisting of vertices (i.e., objects) and edges (i.e., pointers to objects). An example is depicted in Figure 2.7: the left graph $G$ contains four objects $A, B, C, D$ and three pointers (black arrows) $A \to B$, $A \to C$ and $C \to D$. Graphs with a single root object (here object $A$) are sufficient for the considered scenario. In general, they may be cyclic.

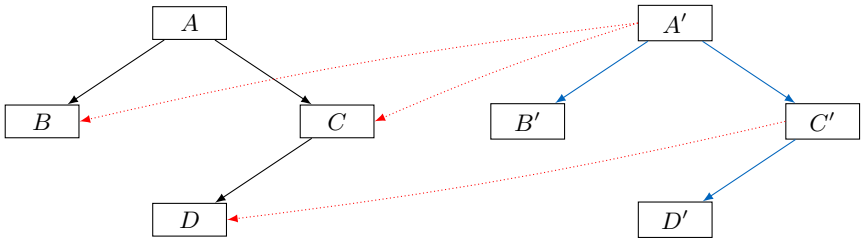
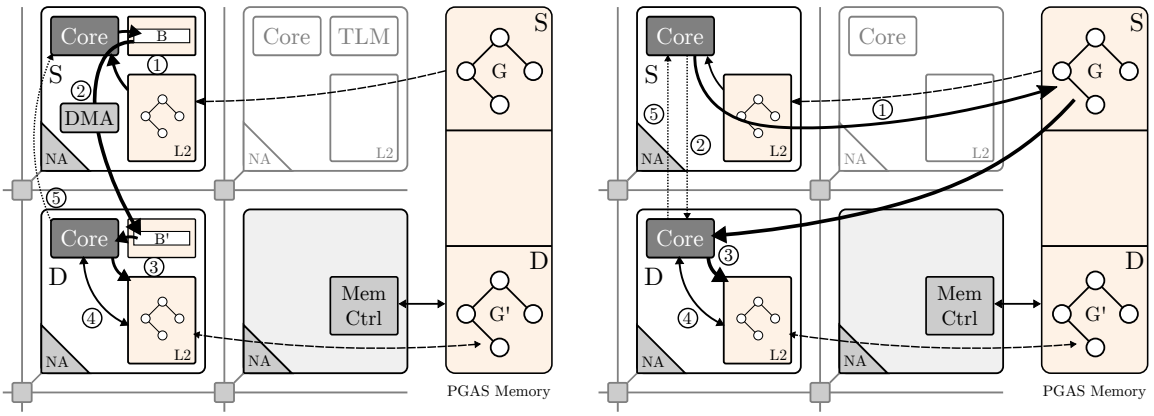



**Figure 2.7: Illustration of graph copy approaches.** Shallow graph copy is a bitwise copy of the each original object resulting in the dotted red arrows. In contrast, a deep graph copy adjust all pointers resulting in the solid blue arrows. The `AT statement` requires a deep graph copy.

Based on these definitions and shown in Figure 2.7, a graph copy can be shallow or deep. A shallow copy is a bitwise copy $A', B', C', D'$ of each original object $A, B, C, D$. Thus, all contained references would still point to the original objects (dotted red arrows) $A' \to B$, $A' \to C$ and $C' \to D$. However, this would not result in the desired and, for this use-case, required deep copy where all references have to be updated to correctly point to the newly created objects (solid blue arrows) $A' \to B'$, $A' \to C'$ and $C' \to D'$. The deep copy strategy clones the entire dependency graph, resulting in two independent graphs, $G$ and $G'$.

A well-established way of implementing the deep copy mechanism is to use the combination of **serialization** and **deserialization**. Therefore, the sender traverses the original object graph $G$ to trace all references. Then, each object is stored consecutively in a buffer $B$ with additional information to reconstruct the object afterward (serialization). Next, the buffer $B$ is transferred into a buffer $B'$ of the receiver by e.g., a DMA unit. Finally, the receiver reconstructs the destination graph $G'$ by deserializing the buffer $B'$ while updating all references to point to the new objects. In general, this is very complex since the entire graph has to be traversed, serialized, transferred, and deserialized. This approach is especially prevalent in distributed memory systems where the receiver has no access to the sender's memory partition.

However, in distributed shared memory systems, it is possible to skip the serialization and deserialization and instead use **cloning** to copy the object graph on-the-fly [167]. The receiver traverses the original object graph $G$ read-only and directly creates $G'$ with the updated pointer references to the newly allocated objects on the receiver side. Of course, this requires proper synchronization.




**(a) Serialization**. 1) Serialize $G$ into $B$, 2) DMA transfer of $B$ to $B'$, 3) Deserialize $G'$ from $B'$, 4) Execute $G'$, and 5) Send Result.

**(b) Cloning**. 1) Write back $G$ to memory, 2) $S$ signals $D$, 3) Copy graph $G$ to $G'$, 4) Execute $G'$, and 5) Send Result.

**Figure 2.8: Related graph copy approaches on a tile-based PGAS architecture.** Both mechanisms copy an object graph $G$ from place $S$ to place $D$. Bold arrows indicate bulk data transfer, dotted arrows designate control/metadata messages, and dashed arrows denote possible traffic due to cache misses/evictions.

**Related Graph Copy Approaches on a Tile-Based PGAS Architecture.** As mentioned above, the efficient transfer and handling of object graphs are crucial for object-oriented programming on distributed shared memory architectures. Thus, there exist several approaches to accelerate the transfer of object graphs [168, 169].

The following two paragraphs describe the copying of an object graph $G$ from place $S$ to place $D$ on a tile-based PGAS architecture for serialization and cloning approaches. They are further illustrated in Figure 2.8a and Figure 2.8b, respectively, and will be analyzed in more detail in Section 5.4. In those figures, bold arrows indicate bulk data transfer, dotted arrows designate control/metadata messages, and dashed arrows denote possible traffic due to cache misses/evictions.

**Serialization.** The serialization-based approach, which is the most common related work, performs the following steps as depicted in Figure 2.8a:

1) **Serialize:** A core on place $S$ serializes the object graph $G$ into the buffer $B$ residing in its tile local memory. The serialization requires a complete graph traversal of $G$ to reach every object. $G$ resides in partition $S$ of the main memory. However, it or parts of it may be cached in the L2 cache of tile $S$ so that the graph traversal does not result entirely in remote load operation of the NoC to the main memory. This is indicated by the dashed arrow belonging to step (1).

2) **Transfer:** The serialized buffer $B$ is copied via DMA transfers into the buffer $B'$ residing in the tile-local memory of the destination tile $D$, and the receiver $D$ is signaled that the DMA is completed,

3) **Deserialize:** A core on $D$ reconstructs the graph $G'$ by deserializing buffer $B'$. Moreover, when deserializing $G'$ out of the buffer $B'$ in step (3), the graph $G'$ is stored in the L2 cache of tile $D$ as long as no cache evictions occur. There is no need

to write back[8] or flush $G'$ back to the main memory. Thus, in step (4), the execution of graph $G'$ will profit from L2 cache hits.

4) **Execute:** The graph $G'$ is executed. *Although this step belongs to the* `AT statement` *it is not attributed to the inter-process communication but to the processing part of the application.*

5) **Result:** The result is transferred back to the $S$.

There exist serialization-based approaches which do not require the buffer $B'$ when $B$ and $B'$ reside in the same physical memory [170, 171, 169].

**Cloning.** Other approaches even entirely avoid the (de-) serialization overhead and use graph cloning instead [168, 169]. An efficient serialization-free software graph cloning mechanism, called `PEGASUS`, has been presented by Mohr et al. [169] which is the closest related work to this thesis. The mechanism depicted in Figure 2.8b performs the following steps:

1) **Write back:** A core on place $S$ writes back the graph $G$ with all its objects from its caches to its memory partition $S$. (Note that for a graph write back a complete graph traversal is required to reach every object.)

2) **Notify:** A core on $S$ signals the receiver $D$ and passes metadata to $D$.

3) **Clone:** A core on $D$ then copies the graph $G$ directly from partition $S$ into its partition $D$.

4) **Execute:**[9] The graph $G'$ is executed.

5) **Result:** The result is transferred back to the $S$.

**Discussion.** The serialization-based approach (Figure 2.8a), described above, follows a "1) place-local read – 2) inter-place DMA transfer – 3) place-local write – 4) execute" style regarding the cache coherence boundaries. Thus, no additional cache writeback or invalidation operations must be performed when crossing the cache coherence boundary between both places. In contrast, the cloning based `PEGASUS` approach follows a "1) place-local write back – 2) inter-place notify – 3) place-remote invalidate & read – 4) execute" style. Therefore, the involved caches need to be handled carefully when crossing the coherence boundary between the places. The L2 cache write back in step (1) of `PEGASUS` is mandatory as the L2 cache is configured in writeback mode. Moreover, in step (3), before reading the graph $G$ from the memory partition $S$, the corresponding memory range has to be invalidated in the L2 cache of $D$ in order to have a clean read from memory (the memory range might have been cached during a former graph copy). Like the serialization-based approach, the execution of the graph $G'$ in step (4) will profit from L2 cache hits.

## 2.4 Prototype Platform

The previous section presented the state of the art relevant for this thesis. The goal of this thesis is to apply near-memory acceleration to mitigate inter-process communication challenges of tile-based many-core architectures. Thus, Section 2.1 presented compute-centric

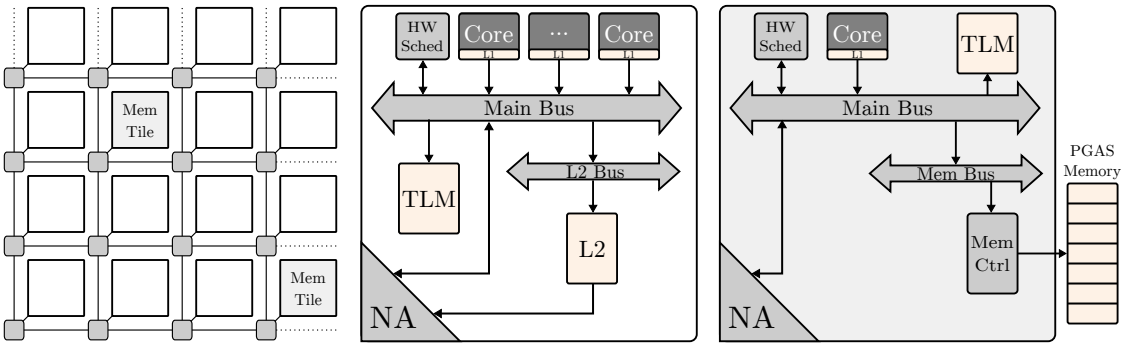
[8] This thesis complies with the following differentiation in cache terminology: 1) invalidate, 2) writeback without invalidation, and 3) flush, which is a combination of invalidate and writeback.

[9] Reminder: Although this step belongs to the `AT statement` it is not attributed to the inter-process communication but the processing part of the application.

systems and, in particular, tile-based architectures. Section 2.2 focused on near-memory computing while Section 2.3 addressed inter-process synchronization and communication. Now, this section presents the prototype platform and the software and benchmark environment which have been used to demonstrate the contributions of Chapter 3, Chapter 4, and Chapter 5. Although they have been developed in the context of the research project Invasive Computing [40] and make use of its hardware and software infrastructure, the concepts are applicable in general.

### 2.4.1 Tile-Based Many-Core with DSM

An overview of the tile-based many-core architecture is depicted in Figure 2.9. The 4x4 tile design consists of 14 compute tiles and two memory tiles with the arrangement shown in Figure 2.9a. Further details and parameters are described in the following paragraphs. The presentation is limited to the components and aspects relevant to this thesis.




**(a)** 4x4 tile design. **(b)** Compute tile overview. **(c)** Memory tile overview.

**Figure 2.9: Overview of the prototype platform.** Arrangement and tile-internal view of the 4x4 tile architecture with two memory tiles. Unmarked white tiles are compute tiles. The small gray boxes denote NoC routers. Only components and aspects relevant for this thesis are depicted.

**Architecture Overview.** The internal structure of a compute tile is depicted in Figure 2.9b. It contains five Leon 3 SPARC-V8 RISC cores [172, 173] of which up to four can be used as application cores, while the fifth core is dedicated for system tasks like interrupt handling. Whereas each core is equipped with a private L1 cache, the L2 cache is shared among the cores of a tile and caches accesses to remote memory. The L1 caches inside a tile maintain intra-tile cache coherence via bus snooping. However, no inter-tile coherence is used by the prototype setup [10] and contributions of this thesis.

Moreover, each tile contains 8 MByte of tile-local SRAM memory (TLM), which can be accessed locally by cores directly or remotely via the NoC and network adapter (NA). The TLM holds the text and data segments of the program and OS and temporary user data. Additionally, each tile is equipped with a hardware scheduler [165] which contains the run queues for the tile-local cores. 16 Byte task descriptors can be inserted by local cores and remotely via the network adapter. Since the program and OS segments are replicated in each TLM, remote task invocation/scheduling is seamlessly possible.

[10] In contrast, Srivatsa et al. [7, 8, 9, 10, 174] enhanced the platform by a coherency region manager (CRM), which enables inter-tile coherence for a dynamically configurable subset of tiles.

The network adapter, initially developed by Zaib et al. [175, 176], is the interface of each tile to the corresponding router of the scalable NoC, initially developed by Heißwolf et al. [177, 178]. The NA is connected to the back end of the L2 cache to facilitate its remote load-store operations. Besides, the NA and NoC enable DMA transfers to and from remote memories, bypassing the L2 cache. Furthermore, as mentioned above, the NA supports remote task invocation in cooperation with remote hardware schedulers. This thesis will enhance the NA by the functionality to trigger and communicate with the proposed remote accelerators.

Besides the compute tiles, the prototype platform also contains memory tiles whose internal structure is depicted in Figure 2.9c. Like the compute tiles, the memory tiles are also equipped with a hardware scheduler, a tile-local memory, and a network adapter. However, only the dedicated system core but no application core is present. Furthermore, the memory tile connects to global off-chip DDR-3 SDRAM via its memory controller. The entire global memory is physically distributed among the two memory tiles.

**FPGA prototype.** The entire design is synthesized on the prodesign proFPGA [179] prototyping platform consisting of four connected Virtex-7 2000T FPGAs [180]. Since the prototype platform is part of the Invasive Computing project and has all along been implemented in a single clock domain, it can be operated at only 50 MHz due to bottlenecks in several state-of-the-art components like the L2 cache. However, since the DDR-3 memory controller requires a higher minimum frequency, it is operated at 100 MHz. This leads to a clock domain crossing in the memory tile between the Main Bus and the Mem Bus.

Besides the frequency values, Table 2.2 provides an overview of selected cache and memory parameters relevant for this thesis. Furthermore, the Leon 3 cores operate with enabled branch prediction and floating-point units.

**Table 2.2: Memory and cache parameters of the prototype platform.** These parameters are used for all evaluation throughout this thesis if not specified otherwise explicitly.

| Parameter | Value | Parameter | Value |
|---|---|---|---|
| L1-I cache sets | 2 | L1 cache policy | write-through |
| L1-I cache set size | 16 kByte | L2 cache policy | write-back |
| L1-I cache line size | 32 Byte | L1 hit time | 1 cycle |
| L1-D cache sets | 2 | L2 hit time | 20 cycles |
| L1-D cache set size | 16 kByte | L2 miss time | 90 cycles |
| L1-D cache line size | 16 Byte | TLM acc. time | 20 cycles |
| L2 cache sets | 4 | LEON 3 freq. | 50 MHz |
| L2 cache set size | 128 kByte | L1 & L2 cache freq. | 50 MHz |
| L2 cache line size | 32 Byte | TLM freq. | 50 MHz |
| Tile-local memory (TLM) | 8 MByte | Tile freq. | 50 MHz |
| Main Memory | $2 \cdot 1$ GByte | MEM ctrl freq. | 100 MHz |

**Platform Variants.** Throughout this thesis, minor variations of the presented architecture are used to evaluate different aspects of the chapter's contributions. The platform variants are summarized in Table 2.3. The table contains the chapter, the tile configuration, and a comment to argue for the particular deviation of the default configuration. The upcoming chapters will present all further modifications to the prototype required for the proposed contributions.

**Table 2.3: Platform variants of the prototype.** Throughout this thesis, minor variations of the prototype architecture are used to evaluate different aspects of the chapter's contributions. By default four application cores are used in each compute tile.

| Chapter | Tile configuration | Comment |
|---|---|---|
| Chapter 3 | 4x4 design with 16 compute tiles | no memory tile |
| Chapter 3 | 4x4 design with 16 compute tiles | 7 application cores per compute tile to create higher concurrency, no memory tile |
| Chapter 3 | 4x4 design with 14 compute tiles and 1 memory tile | used for macrobenchmark evaluation, memory tile at (1,1), the second memory tile is unused |
| Chapter 4 | 4x4 design with 16 compute tiles | no memory tile |
| Chapter 4 | 2x2 design with 4 compute tiles | used for scalability analysis, no memory tile |
| Chapter 4 | 2x1 design with 2 compute tiles | used for scalability analysis, no memory tile |
| Chapter 4 | 1x1 design with 1 compute tiles | used for scalability analysis, no memory tile |
| Chapter 5 | 4x4 design with 14 compute tiles and 2 memory tiles | 4x4-twin with 2 memory tiles located at grid position (1,1) and (3,3) |
| Chapter 5 | 4x4 design with 14 compute tiles and 1 memory tile | 4x4-single with 1 memory tile at (1,1), the second memory tile is unused |
| Chapter 5 | 2x2 design with 3 compute tiles | 1 memory tile at (1,1) |

### 2.4.2 Operating System and Programming Paradigm

OctoPOS [181] is used as the operating system in this thesis as it is able to exploit the described features of the hardware platform. It is a distributed operating system developed in the context of the Invasive Computing research project. It provides and uses the following existing mechanisms to facilitate inter-process communication between tiles irrespective of the contributions of this thesis. The first mechanism is remote task invocation. A 16 Byte task descriptor consisting of a function pointer, two data words, and OS flags can be sent to remote cores via the network adapter and remote hardware scheduler. Second, asynchronous inter-tile DMA transfers with optional task invocation upon completion on the sender and/or receiver side are provided. The sender may specify two task descriptors when initiating the DMA transfer. The local and remote network adapters take care of inserting them into the corresponding hardware scheduler. This mechanism enables interrupt-less and non-blocking inter-tile data transfers with integrated receiver notification via task invocation. The third existing mechanism for inter-process communication is condition variables used to signal and wait for certain events. The inter-process synchronization and communication accelerators proposed in this thesis will complement these communication mechanisms.

The contributions target different programming models. Chapter 3 extends the state of the art of inter-process synchronization in the distributed shared memory paradigm. The proposed complex remote atomic operations are directly integrated into the native OctoPOS. In contrast, Chapter 4 targets queue-based message-passing communication. The contribution is integrated into the MPI library and showcased with the MPI-based NAS Parallel Benchmarks, which follow a distributed memory paradigm. Finally, Chapter 5 proposes a more efficient implementation of the PGAS inter-process communication mechanism between tiles. The contribution is demonstrated with the X10 IMSuite benchmarks [182] which are written in the PGAS programming language X10. Although these paradigms and languages are used to demonstrate the viability of the approaches, the concepts are applicable beyond them.

### 2.4.3 Benchmarks

Finally, this section presents and analyzes the benchmarks utilized to demonstrate the contributions of this thesis.

**MPI-Based NAS Parallel Benchmark Suite.** Chapter 4 uses the MPI-based NAS Parallel Benchmarks (NPB) [183] written in C and Fortran. It consists of the following eight benchmark kernels: CG (Conjugate Gradient), EP (Embarrassingly Parallel), FT (Fast Fourier-Transform), IS (Integer Sort), MG (Multigrid), BT (Block Tri-diagonal solver), SP (Scalar Penta-diagonal solver), and LU (Lower-Upper Gauss-Seidel solver). Since the evaluation of Chapter 4 primarily uses the 4x4 tile design with 64 application cores and the 2x2 design with 16 application cores, the benchmark kernels are compiled for these configurations. While all kernels exist for the input set running on the 2x2 tile design, unfortunately, only five (i.e., CG, EP, FT, IS, and MG) of them exist for the input set running on the 4x4 tile design. Moreover, due to the limited size of the tile-local memory, the kernels are executed with the problem size class 'S'.

The inter-process communication behavior of the NPB has been analyzed in [184, 185]. The FT, IS, BT, and SP kernels exhibit all-to-all communication predominantly. While the amount of communication is relatively low for the BT and SP benchmark, it is highest for the communication-intensive IS kernel. The other four kernels use mostly point-to-point communication. MG and LU follow a ring pattern while CG forms a nearest-neighbor chain. As expected, the embarrassingly parallel EP kernel exhibits negligible communication.

**PGAS X10 IMSuite Benchmark Suite.** Chapter 5 uses the PGAS-based IMSuite benchmarks [182] written in X10. They are a collection of classical distributed algorithm kernels. A real-world application typically consists of a mixture of the individual kernels. In the following paragraphs, their setup is described, and their inter-process communication behavior is characterized.

The benchmarks are used in the iterative, concurrent, and distributed variant (Iterative/X10-FA). Table 2.4 provides an overview of the benchmarks, including short descriptions of the benchmarks themselves, as well as their input set characteristics and input set sizes. All benchmark kernels have an initialization phase, a computation phase (the *region of interest* RoI), and a result verification phase. Like the IMSuite authors, this thesis focuses on the RoI of the benchmarks, as the distributed and parallel computations, including inter-process communication, happen here.

The benchmark communication behavior is characterized by analyzing the amount and size of the object graph copies performed during inter-process communication. Therefore, a separate run of each benchmark was performed, and all copied object graphs were counted and measured. Table 2.5 shows the results for each benchmark, including the number of copies, the average number of objects per graph, and the average graph size. Further analysis showed that most graphs have a similar composition with one or two typical combinations of graph size and the number of objects per graph. The last column of Table 2.5 contains a classification of the graphs into small (less than 1 000 bytes), medium (between 1 000 and 10 000 bytes), and large (more than 10 000 bytes) graphs. It further quantifies the percentage of the most prevalent types.

**Table 2.4: An overview of the IMSuite benchmarks and the used input sets.** Table taken from [3]. More information about the benchmarks is provided in the IMSuite documentation [182].

| benchmark | abbrev. | description | input | input description |
|---|---|---|---|---|
| bfsBellmanFord | BF | Breadth-first search in a graph using the Bellman-Ford algorithm | 64-spmax | Sparse graph with 64 nodes and $n \log n$ edges |
| bfsDijkstra | DST | Breadth-first search in a graph using Dijkstra's algorithm | 64-rn | Dense graph with 64 nodes and random adjacency |
| byzantine | BY | A solution of the Byzantine generals problem | 16-spar-max | Sparse graph with 16 nodes and $n \log n$ edges |
| dijkstraRouting | DR | Single-source routing through a graph with Dijkstra's algorithm | 32-spar-weq-max | Sparse graph with 32 nodes and $n \log n$ edges of equal weight |
| dominatingSet | DS | Computation of a dominating set | 32-spar-max | Sparse graph with 32 nodes and $n \log n$ edges |
| kcommitte | KC | Partitioning a graph into $k$-committees | 64-rn | Dense graph with 64 nodes and random adjacency |
| leader_elect_lcr | LCR | Leader election in a unidirectional ring network | 64 | Ring of 64 nodes |
| leader_elect_hs | HS | Leader election in a bidirectional ring network | 64 | Ring of 64 nodes |
| leader_elect_dp | DP | Leader election in a general network | 32-spar-max | Sparse graph with 32 nodes and $n \log n$ edges |
| mis | MIS | Finding a maximal independent set in a graph | 64-spmax | Sparse graph with 64 nodes and $n \log n$ edges |
| mst | MST | Computation of a minimal spanning tree | 32-spmax | Sparse graph with 32 nodes and $n \log n$ edges |
| vertexColoring | VC | 3-coloring of a tree | 64-rn | Tree with 64 nodes |

**Table 2.5: Object graph statistics of the IMSuite benchmarks**. The table contains the number of graph copy operations performed by each benchmark and the average number of objects and bytes per copy operation. The last column classifies the graph size into small (less than 1 000 bytes), medium (between 1 000 and 10 000 bytes), and large (more than 10 000 bytes) graphs and quantifies the percentage of the most prevalent types. Table extended from [3].

| benchmark | # copies | avg. objects | avg. size | classification |
|---|---|---|---|---|
| BF | 2 150 | 10.5 | 16 864 | 91% large |
| DST | 8 492 | 11.3 | 16 548 | 91% large |
| BY | 7 362 | 16.5 | 1 875 | 83% medium |
| DR | 13 550 | 6.7 | 2 382 | 36% small / 45% medium |
| DS | 38 778 | 8.7 | 2 571 | 46% small / 41% medium |
| KC | 58 224 | 10.4 | 709 | 98% small |
| LCR | 17 370 | 10.0 | 670 | 98% small |
| HS | 46 336 | 12.0 | 764 | 99% small |
| DP | 8 830 | 15.0 | 4 582 | 77% medium |
| MIS | 6 168 | 13.4 | 16 516 | 34% large |
| MST | 18 486 | 13.3 | 3 618 | 24% small / 53% medium |
| VC | 2 388 | 16.5 | 16 725 | 70% large |

# 3 Near-Memory Accelerated Inter-Process Synchronization

> "Proof of concept is the art of winning a ship-building contract by showing the working model of a paper boat."
>
> — Nipun Varma [186]

Efficient thread synchronization is crucial to the performance of parallel programming since it often is in the critical path of execution. In shared-memory programming, multiple threads communicate and synchronize through concurrent data structures residing in shared memory [22]. The atomic access to these data structures (e.g., semaphores, stacks, queues, or lists) needs to be guaranteed and provided performantly. This is especially challenging on tile-based MPSoCs.

This chapter contributes to alleviating the overhead of inter-process synchronization between tiles via various remote atomic operations on concurrent data structures. This simple yet crucial accelerator helps to mitigate data-to-task locality challenges. It serves as a proof-of-concept for run-time support near-memory acceleration (RTS-NMA), which will be applied to more complex scenarios in the upcoming Chapter 4 and Chapter 5.

The approach is further motivated in Section 3.1 with a refined problem statement. Section 3.2 defines the aspired goals and requirements followed by Section 3.3 which explains the assumptions made and the given boundary conditions. Then, Section 3.4 explores several design alternatives for inter-process synchronization. Section 3.5 presents the concept of the proposed near-memory synchronization accelerator, describes architectural details of the design and implementation, and explains how it satisfies the goals and requirements. Furthermore, the integration of the accelerator into a broader application scenario is described. Next, Section 3.6 covers the evaluation of the approach with a particular focus on inter-tile performance and scalability. Finally, the chapter is concluded in Section 3.7.

## 3.1 Motivation and Problem Statement

Many well-known atomic operations (e.g., `TAS`, `CAS`, `LL/SC`, or `fetch-and-ops`) to handle concurrent data structures exist as hardware ISA extensions in today's MPSoC platforms. The classification of synchronization primitives, as presented in Section 2.3.3, distinguishes between three main categories with different kinds of hardware support:

1) **Lock-based** primitives using hardware lock support for the critical section,
2) **Lock-free** primitives with hardware `CAS` or `LL/SC` to eliminate locking,
3) **Dedicated hardware** implementations of the entire critical section.

All of these have their advantages and downsides, which will be detailed in the upcoming design space exploration. However, on tile-based many-core architectures, all these

primitives are often solely, if at all, present locally inside a tile. Their inter-tile support is missing in most systems with a NoC interconnect since it does not inherently support atomic memory operations. Often, the only possibility is to migrate a task, including all its scheduling overhead, to the remote tile that executes the atomic operation on behalf of the initiating thread. Moreover, the NUMA properties of distributed shared memory architectures give new weight to various aspects of communication and synchronization. For example, the acquisition of a remote spinlock or the repeated execution of a failed remote `CAS` experience much higher memory access latencies and retry penalties than their local pendants. Hence, even though a specific implementation of a given concurrent data structure may benefit a conventional bus-based system, an entirely different behavior might be observed on a tile-based many-core architecture. Nevertheless, it is crucial to enable efficient inter-process synchronization for tile-based MPSoCs.

## 3.2 Goals and Requirements

This chapter's overall goal is to enable efficient, scalable, and lightweight inter-process synchronization on tile-based many-core architectures. The overhead of inter-process synchronization should be alleviated by providing remote atomic operations for crucial run-time support data structures. Therefore, the effects of lock-based, lock-free and dedicated hardware synchronization are investigated on such systems, their inter-tile support is extended, and their conceptual advantages should be combined to tackle their downsides. The following functional and non-functional requirements need to be fulfilled by the approach.

**Inter-Tile Support** (functional) The approach must extend crucial synchronization primitives across tile borders.

**Performance** (non-functional) Since synchronization often resides in the critical path of execution, the intended solution must be efficient and outperform other approaches.

**Scalability** (non-functional) As many-core architectures contain dozens or hundreds of cores and several memories, the concept must be scalable to these system sizes.

**Lightweight Software Integration** (non-functional) Inter-process synchronization should be as lightweight as possible. Thus, direct and synchronous accelerator calls without additional scheduling or notification overhead in software should be possible so that the cores can focus on processing the actual application workload.

**Deadlock-Freedom, Wait-Freeness, No ABA Problem** (functional) It is crucial that the approach is deadlock-free and not suffer from the ABA problem. Furthermore, a wait-free implementation should be provided.

## 3.3 Assumptions and Constraints

After defining the desired goals and requirements of the approach, this section elaborates on the assumptions made and the given boundary conditions for the design. The assumptions and constraints must be considered if one intends to apply the concept to other architectures or software environments.

**Synchronous Implementation** All accelerator calls presented in this chapter are integrated into software via synchronous outsourcing to hardware. The software passively waits

for the completion of the atomic operations so that the operation only returns to software when it is completed in hardware, i.e., similar to waiting until a read request returns. Thus, no scheduler involvement, interrupt, and active polling are required to interact with the accelerator. It is assumed and ensured by hardware that one pending request per core, not per tile, is possible.

**Access to Hardware Registers** It is assumed that direct access to hardware registers to interact with the accelerator is possible from user space. The lightweight software integration enables low overhead as known of existing atomic operations. It needs to be assessed if accelerator access without memory protection is allowed on other systems.

**No MMU or Virtual Memory** The prototype design does not use an MMU or virtual memory. If one desires to incorporate those, it would have to be ensured that the appropriate address translations are available to the accelerator.

**Limited Portfolio of Atomic Operations** This thesis picks and implements representative atomic operations to support a subset of concurrent data structures. An extension of this portfolio requires a dedicated implementation of the specific functionality in the hardware accelerator. Nevertheless, the accelerator interfaces to software and the network adapter are designed generically so that extensions to the actual accelerator can be integrated easily.

**Maintaining Cache Coherence** The design does not assume hardware-managed inter-tile cache coherence. As the near-memory accelerator operates on the respective memory directly, the atomic operations bypass the caches on the initiator tile. Therefore, since the accelerator is used in a shared memory programming environment, the required cache line invalidations and writebacks must be initiated by software explicitly. Nevertheless, although the concept does not rely on hardware-managed inter-tile cache coherence, it is well compatible with it. For instance, the coherency region manager (CRM) proposed by Srivatsa et al. [8] could snoop on the accelerator operations and initiate coherency messages if necessary.

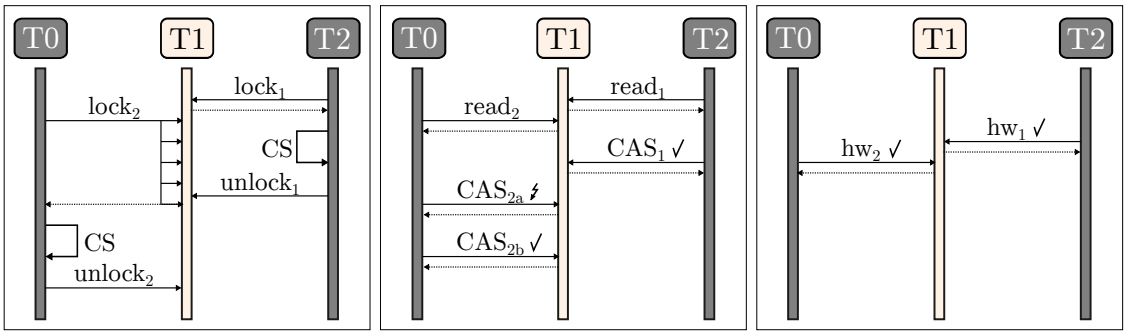
## 3.4 Exploring Design Alternatives

With the goals, requirements, assumptions, and constraints defined in the previous sections in mind, this section explores several design alternatives. The exploration is performed with the help of the concurrent `shared counter` data structure. This example scenario explains the key differences, advantages, and downsides of the different synchronization variants on tile-based many-core systems. The lessons learned from the `shared counter` scenario can then be applied to more complex concurrent data structures and atomic operations. The gained insights lead to this chapter's contribution, presented in the next Section 3.5.

### 3.4.1 Example Scenario

A simple scenario of a concurrent data structure is a shared counter between many cores. It can be used to implement e.g., semaphores. Without loss of generality, this example is limited to counter increments. For correctness, it is required that the following sequence (i.e., the critical section CS) is performed atomically: (1) reading the current counter value from memory, (2) incrementing it, and (3) writing back the updated value to memory. Figure 3.1 depicts various design alternatives for incrementing the shared counter in the

form of message sequence charts. Without loss of generality, the counter resides in the tile-local memory of tile one (T1), whereas all other cores increment the shared counter either locally or remotely. The different variants will be investigated in the following.




**(a)** lock-based **(b)** lock-free **(c)** dedicated hardware

**Figure 3.1: Comparison of inter-process synchronization design alternatives.** Message sequence charts for three types of the shared counter implementation. The counter variable resides on T1, while all cores increment the counter variable (here shown remotely from T0 and T2).

### 3.4.2 Design Alternatives

**Lock-Based Shared Counter.** A lock-based variant of a given function locks the critical section (CS) that must be executed atomically. Applying this variant to the `shared counter` data structure results in the following lock-based implementation (Listing 3.1) which is illustrated in Figure 3.1(a) as a message-sequence chart.

```
1  SC_increment_lock_based(bool *lck, int *cnt){
2      lock(lck);              // remote spinlock (atomic)
3      tmp = *cnt;             // remote read
4      tmp++;                  // local increment
5      *cnt = tmp;             // remote write
6      unlock(lck);
7  }
```

**Listing 3.1:** Pseudo code of a **lock-based** shared counter implementation.

First, the lock is acquired atomically in line of code 2 (loc 2). The lock acquisition may require several retries when another contender holds the lock. An efficient hardware lock implementation, as depicted for $lock_2$, lowers the software interaction for acquiring the lock through polling in hardware at the remote site. Such hardware support for remote spinlocks is present in some state-of-the-art architectures and thus, provided by the evaluation platform of this thesis. In the second step, the critical section (loc 3 – 5) is executed, consisting of the remote read of the current counter value from memory (loc 3), its incrementing (loc 4), and the remote writeback of the new counter value to memory (loc 5). Figure 3.1(a) illustrates that the critical section inside the lock is retry-free, since the read-modify-write cycle (loc 3 – 5) is non-conditional. Finally, the CS is unlocked in loc 6.

**Lock-Free Shared Counter.** A lock-free implementation of any given data structure can be attained with the universal atomic primitives `CAS` or `LL/SC`. Applying this variant to the `shared counter` data structure results in the lock-free and `CAS`-based implementation (Listing 3.2) which is illustrated in Figure 3.1(b).

```
1 SC_increment_lock_free(int *cnt){
2     tmp = *cnt;                        // remote read (non-atomic)
3     do {
4         old = tmp;                     // local store
5         tmp = CAS(cnt, old, old+1);    // remote CAS (atomic)
6     } while(tmp != old);               // local compare
7 }
```

**Listing 3.2:** Pseudo code of a **lock-free** shared counter implementation.

First, the old counter value is read non-atomically in loc 2 without locking. Then, a `CAS` loop is executed in loc 3 – 6. The remote `CAS` is performed on the counter with the old value and the incremented old value as the new value (loc 5). The `CAS` succeeds if the counter value has not been altered in the meantime (loc 6). Otherwise, the loop is reiterated with an updated old value (loc 4) that has been returned by the `CAS` of the previous loop iteration (loc 5). This is illustrated in Figure 3.1(b). The conditional read-modify-write cycle, i.e., the $\text{read}_2$ (loc 2) and the $\texttt{CAS}_{2a}$ (loc 5) from T0 to T1, is interfered by another $\texttt{CAS}_1$ from T2 to T1. The $\texttt{CAS}_1$ succeeds and updates the counter value. Thus, the $\texttt{CAS}_{2a}$ fails and requires a retry ($\texttt{CAS}_{2b}$). Hence, the operation must be repeated until it is successful, i.e., until no concurrent access interfered in the meantime.

**Shared Counter via Task Migration.** The lock-based and lock-free shared counter described above operated remotely on the data structure. This was enabled by hardware support for remote spinlocks and remote `CAS`. If these were not available and only their local pendants exist (as is the case for most tile-based architectures), another design alternative would be the use of `task migration`. This sounds excessive for the implementation of a counter increment. However, first, it is a proof-of-concept scenario. Second, and more importantly, it is the only possibility to ensure inter-process synchronization between tiles on many architectures.

Therefore, the entire Listing 3.1 or Listing 3.2 is scheduled on the remote tile T1 via remote task invocation so that all remote operations become local ones. Only local spinlocks (`TAS`) and local `CAS`, present in nearly every architecture, are used. Although remote memory accesses are saved, the scheduling overhead for the task migration newly arises. Note that the user is free to choose whether to implement the data structure lock-based or lock-free in this case.

**Hardware Accelerated Shared Counter.** Finally, a dedicated hardware implementation of the concurrent `shared counter` promises the best performance. It leverages the atomic fetch-and-increment mechanism illustrated in Figure 3.1(c). The pseudo source code is given in Listing 3.3. This implementation is retry-free by definition since the entire critical section, i.e., the read-modify-write operation of the fetch-and-increment, is executed atomically in hardware. The fetch-and-increment operation is present as an ISA extension in many CPUs. However, most architectures only provide it for local and not for remote operations required for tile-based systems. Moreover, it is one of the few atomic operations in state-of-the-art architectures. This might be due to the fixed functionality, limiting its use case, and requiring a dedicated hardware implementation for every operation.

```
1 SC_increment_hardware(int *cnt){
2     fetch_and_increment(cnt);          // remote atomic operation
3 }
```

**Listing 3.3:** Pseudo code of a **dedicated hardware** shared counter implementation.

### 3.4.3 Discussion

The **lock-based** variant relies on locking the critical section (CS) so that no interfering concurrency by others can occur. Whereas by design, the critical section inside the locked region is retry-free [1], the lock acquisition itself is not, since it is unclear when it will be successfully acquired. Furthermore, it is not wait-free since there is no upper bound for the number of retries until lock acquisition can be given. In contrast, the **lock-free** implementation is not retry-free since its read-modify-write cycle is conditional. Hence, the operation must be repeated until it is successful. In the best case, this can be one try if no concurrent access interfered in the meantime. However, the average case depends heavily on the concurrency, while even starvation cannot be excluded in the worst-case. Thus, wait-freeness is not guaranteed either. The **task migration** approach avoids operating on the data structure via costly remote memory accesses. However, scheduling overhead is introduced. Finally, a **dedicated hardware** implementation promises the best performance. It is retry-free by design since the entire critical section is executed atomically in hardware. However, the `fetch-and-increment` operation is one of few such dedicated implementations present as an ISA extension of CPUs.

Nevertheless, this contribution will demonstrate that the hardware-accelerated shared counter and more complex remote atomic operations to support concurrent data structures are crucial to the performance of inter-process synchronization on tile-based architectures.

## 3.5 Concept and Implementation

As a result of the design space exploration, the concept of a near-memory synchronization accelerator (`CACAO`) is proposed in this section. The approach alleviates inter-process synchronization overhead and complies with the goals and requirements defined in Section 3.2. It fills the void of missing or inefficient inter-tile support for crucial run-time support data structures. The deficiencies of existing lock-based and lock-free implementations are overcome by combining their conceptual advantages, namely retry-freeness and lock-freeness. This is achieved by outsourcing the entire critical section into a dedicated hardware accelerator near the memory where the data structure resides. The following three subsections elaborate on three key attributes of the concept:

1) remote near-memory execution (Section 3.5.1),
2) dedicated hardware acceleration (Section 3.5.2), and
3) support for concurrent data structures (Section 3.5.3).

Besides presenting the concept's main features, several architectural and implementation details of the design are described. Moreover, the hardware-software co-design, i.e., the interplay between the software and the proposed hardware accelerator, is explained in Section 3.5.4. Then, Section 3.5.5 explicates how the functional design requirements are satisfied. Finally, the approach is embedded in a broader application scenario (Section 3.5.6).

[1] Reminder: an operation is retry-free if the read-modify-write cycle is non-conditional, i.e., no interfering concurrency on the data structure can occur that would require to re-execute the operation

### 3.5.1 Remote Near-Memory Execution

The proposed near-memory synchronization accelerator is located near each memory of the tile-based architecture. Figure 3.2 depicts its position within a compute tile, tightly connected to the network adapter (NA) and connected to the local bus. From the local cores' view, the `CACAO` accelerator is not *closer* to the tile-local memory as the local cores themselves. However, since any local or remote core can call the accelerator, it is much closer to memory than all remote cores. This is the interpretation of near-memory acceleration followed in this thesis. The operations are not executed remotely via the NoC but outsourced to the respective accelerator located near the memory in which the respective data structure resides.

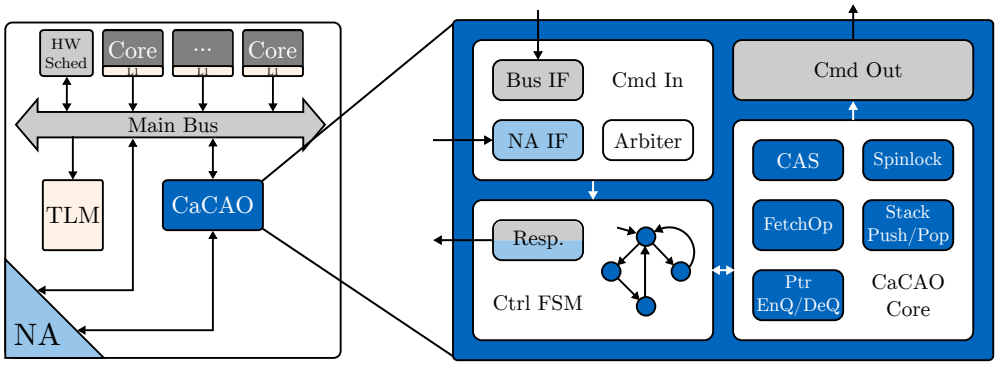



**Figure 3.2: Integration and architecture of the near-memory synchronization accelerator.** Overview of a compute tile including the `CACAO` accelerator (left) and block diagram of the internal structure of `CACAO` (right). `CACAO` receives commands either from a local core or remotely via the NA. The entire critical section is outsourced to the `CACAO` accelerator near the memory where the data structure resides. The accelerator accesses the memory via the bus to provide independence of a particular memory controller protocol implementation.

The near-memory execution is desirable as it reduces remote memory access latencies and increases data-to-task locality. Furthermore, the near-memory execution is required to enable atomic access to the remote memory without locking the entire NoC interconnect since this would unavoidably result in a deadlock.

### 3.5.2 Dedicated Hardware Acceleration

In contrast to lock-based and lock-free implementations where only hardware support for locking or the `CAS` is present, respectively, this contribution outsources the entire critical section into a dedicated hardware accelerator that executes it on behalf of the core. Atomicity of the operations is guaranteed by the hardware module, which is accessing the memory via locked [2] bus accesses. This ensures that no one else can interfere with concurrent operations to the data structure. Furthermore, all atomic operations issued by the cores themselves are consistent with the hardware accelerator's operations.

In contrast to lock-free implementations of concurrent data structures, which might get very complex [143] when transformed from their often more straightforward lock-based pen-

[2] Note that locked bus accesses should not be confused with a lock-based implementation. Locked bus accesses ensure atomic execution of a sequence of bus operations required for e.g., the `TAS`, `CAS`, `fetch-and-Φ` instructions, or the proposed `CACAO` operations.

dants, CACAO implements the critical section as is in hardware. Thus, the implementation is lock-free and retry-free at once, i.e., the advantages of both the lock-free and lock-based variants are combined and guarantee better performance by design. This approach is universally valid with, in theory, arbitrarily complex critical sections. However, the need for a dedicated hardware implementation of each desired functionality is a downside that needs to be pondered over.

Its position within a tile is depicted on the left side, while its internal structure is illustrated on the right side of Figure 3.2. CACAO is tightly connected to the network adapter to receive incoming requests from other tiles, buffered in the NA's existing FIFOs. Moreover, the accelerator is connected to the local bus via its bus interface. The CMD In interface is used to receive incoming requests from local cores (Bus IF), which are arbitrated with the incoming remote requests from the NA IF. The CMD In module decodes the local and remote requests and forwards them to the CACAO core module. Furthermore, the accelerator also contains a CMD Out interface to issue memory reads and writes to the memory controller. The modular design as a bus unit provides independence of a particular memory controller protocol implementation. Thus, it is compatible with the tile-local SRAM and the global DDR-SDRAM memory controller present in every memory tile.

### 3.5.3 Supporting Concurrent Data Structures

The near-memory synchronization accelerator in its current state supports a handful of beneficial atomic operations:

1) test-and-set (TAS),
2) compare-and-swap (CAS),
3) fetch-and-$\Phi$ with $\Phi \in$ {increment, decrement, add, sub, and, or},
4) pointer enqueue and dequeue, and
5) stack push and pop.

They all exist as local (intra-tile) and remote (inter-tile) variants. The remote TAS and CAS are provided for two reasons. First, their inter-tile support is missing on many tile-based architectures. Second, they are used to implement lock-based and lock-free inter-process synchronization between tiles as a reference for evaluation against CACAO, respectively. Some tile-based architectures provide a remote TAS and CAS implementation, and very few provide the fetch-and-$\Phi$ primitives. However, no support exists for the more complex atomic operations on the concurrent queues or stacks.

The presented atomic operations are essential building blocks of many important runtime support data structures. Three of them are implemented in this thesis:

1) concurrent shared counters,
2) concurrent linked-list based queues,
3) concurrent LIFO stacks.

Accelerating them in hardware enables a performant implementation with minimal software involvement. While the concurrent shared counter was presented in detail during the design space exploration, the other two data structures were presented in Section 2.3.3 alongside Figure 2.4 on Page 26.

### 3.5.4 Hardware-Software Co-Design

The interplay between the system-software and the proposed hardware accelerator is described in this section. In a nutshell, the data structures are software-defined and hardware-managed. The contribution of this chapter is limited to synchronizing the atomic access to the data structures. Thus, only pointers to queue or stack elements are handled (i.e., enqueued/dequeued or pushed/popped) by the accelerator. The allocation of new elements and the transfer of the element payload remain a software and decoupled task, respectively.

**Data Structure Creation/Definition.** The concurrent data structures are allocated and defined by software and reside in normal memory. Thus, arbitrary counters, stacks, or queues can be created in any existing memory in the system. No dedicated hardware resources are consumed to store the data structures or additional metadata. Upon creation, the user holds a handle (i.e., a pointer to the memory location of the counter, stack, or queue) passed to the `CACAO` accelerator when initiating a hardware-managed operation on the respective data structure.

**Hardware-Software Interplay.** An instance of the synchronization accelerator is present in every tile and responsible for handling the atomic operations on the data structures residing in this tile's memory. Each accelerator is accessible from any local core via the bus and any remote core via the NoC and network adapter. Whenever an operation is outsourced to the accelerator, the following interplay illustrated in Figure 3.3 is performed:

1) `CACAO` receives the operation from a local (1a) or remote (1b) core,
2) `CACAO` performs the sequence of memory accesses necessary for the operation via locked bus accesses to ensure atomicity,
3) `CACAO` sends a response to the initiating core (3a for local; 3b for remote).

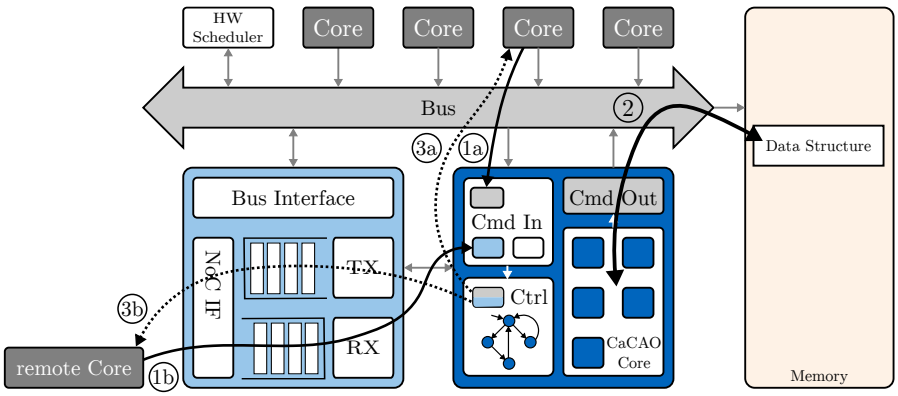



**Figure 3.3: Hardware-software interaction of `CACAO`.** Interaction of local and remote cores with the `CACAO` accelerator. The data structures reside in normal memory. The required atomicity of the operations is ensured by performing the memory accesses in step (2) via bus locking.

Since all supported primitives require a response or acknowledgment, they are implemented synchronously. Thus, each core is limited to one pending request. However, there can be several pending outgoing requests per tile. The maximum number of incoming requests to an accelerator is the total number of cores in the system. These requests are buffered in the virtual channel FIFOs of the network adapter.

**Software API.** Since the software overhead of the hardware-managed operations should be minimal, a thin API wrapper is used. The hardware operations are triggered and completed by writing the commands and reading the response code to and from particular hardware registers. When considering, for example, the `stack push` operation, the software API and driver code boil down to the pseudo-code given in Listing 3.4. These registers (`hw_reg_<>`) on the sender-side can be accessed without locking within the driver code since the concurrent access to them is handled in hardware via bus split transaction. The read operation on register `hw_reg_response` only returns once the accelerator completes its operation and sends back the response code to the initiating tile. This enables lightweight usage with minimal software overhead.

```
1 void stack_push(int *stack_handle, int *element){
2     *hw_reg_stack_push = stack_handle;
3     *hw_reg_operand_1 = element;
4     int done = *hw_reg_response;
5 }
```

**Listing 3.4:** Pseudo code of software API of the `stack push` operation.

### 3.5.5 Satisfying the Functional Design Requirements

The proposed implementation guarantees **deadlock-freedom** by solely and temporarily locking the bus in the tile where the accelerator is operating. In contrast, a deadlock scenario could arise if the bus on the initiating tile is locked for a remote operation. If two tiles mutually performed an atomic operation, both buses would be locked by the initiator. Then, when the accelerator on the respective other tile should perform the operation, the bus would already be locked. The unlock would only occur upon completion of the operation, which never happens. Such a scenario is avoided by the proposed design.

The approach provides **wait-freeness** since an upper bound for the operations' execution time can be given. By design, no interfering concurrency on the data structure can occur during the hardware accelerator's operation. Thus, it succeeds on the first try in contrast to many lock-free implementations. Furthermore, an upper bound for the NoC travel time of the operation's trigger and response messages can be given as the virtual channels are served in a round-robin fashion.

The **ABA problem** is prevalent in many lock-free `CAS`-based implementations of concurrent data structures. Lock-free implementations can avoid it with the LL/SC or the double-word `CAS` operation. However, both are rare on most platforms, especially on tile-based architectures. The proposed contribution does not suffer from the ABA problem since the entire critical sequence of operations is performed atomically by the accelerator via locked bus accesses. Thus, no interfering concurrency by another thread can happen.

### 3.5.6 Application Scenario: Efficient Inter-Tile Memory Allocation

After the conceptual and architectural details have been presented in the previous sections, the synchronization accelerator is embedded in a broader application scenario. The `concurrent stack` is utilized to build and manage an efficient inter-tile memory allocator integrated into the run-time system of the demonstrator platform.

**Motivation.** Memory allocation is one of the most prevalent tasks in nearly every application. Allocating memory requires synchronization since many cores can request new

memory concurrently. In the context of inter-process communication between tiles, also remote memory allocation becomes relevant. One scenario of inter-tile memory allocation occurs in the PGAS inter-process communication, more specifically, in preparation of the metadata transfer from the sender to the receiver (cf. step (2) of the cloning based approach in Figure 2.8b on Page 33). The same metadata transfer is used in Chapter 5. There, the whole inter-process communication mechanism will be analyzed in detail. Nevertheless, in this section, it suffices to understand the metadata transfer described in the sequel.

**Problem Statement.** Ordinarily, the transfer of the metadata $M$ from a sender $S$ to a receiver $D$ would require two round trips to the remote tile with their associated operating system overhead: (1) The sender $S$ allocates a buffer $M$ locally in its tile-local memory, (2) $S$ sets up the content of $M$. (3) $S$ requests a buffer from the receiver $D$, (4) $D$ allocates the buffer $M'$ and returns its address, (5) $S$ sets up a DMA transfer from $M$ to $M'$, and (6) $D$ is notified when the transfer completes. While step (1) is a local memory allocation, steps (3) and (4) denote the remote memory allocation. It requires scheduling a task on the remote tile $D$ that performs the allocation on behalf of the sender $S$.

Note that the metadata transfer resides in the critical path of the inter-process communication and the entire application. An analysis revealed that, in addition to the local allocation in step (1) and the remote allocation in steps (3) and (4), the freeing of the local buffer $M$ lies in the critical path as well. Therefore, it is desirable to accelerate the (remote) memory allocations and de-allocations. The concept of applying the atomic stack push/pop operations to this problem will be presented in the following.

**Implementation.** The additional round trip for buffer allocation (steps (3) and (4)) can be avoided by leveraging the proposed remote atomic stack push and pop operations to build and manage an efficient inter-tile allocator. Note that the stack can also be accessed for the local buffer allocations in step (1).

Therefore, a pool of metadata buffers is pre-allocated, and the corresponding pointers are pushed onto the concurrent stack. Thus, the stack is initialized with available metadata buffers. Whenever a new metadata buffer is required, there is no need for a (remote) allocation with the associated software involvement. Instead, a core can directly obtain a buffer by atomically popping from the stack via the `CACAO` accelerator. When the buffer is no longer in use, a core returns it to the pool using the atomic stack push operation. If the stack is empty, the pop operation blocks until the next push. When pushing, there is no need to check for stack overflows since the stack is used in a push-after-pop manner.

This approach is viable for two reasons. First, metadata buffers typically have the same size, especially in this scenario, so that no problem arises when the stack only manages same-sized buffers. Second, allocations often reside in the critical path of execution and thereby contribute to the overall execution time. However, the combination of pre-allocations, which do not lie in the critical path, with efficient atomic stack push/pop operations (residing in the critical path) mitigate this problem.

**Summary.** The atomic stack push and pop operations were utilized to build and manage an efficient inter-tile memory allocator. The perpetual memory consumption due to buffer pre-allocation is bearable since such metadata buffers for inter-process communication are relatively small and frequently used. The limitation of fixed buffer sizes is counteracted by minimized software involvement. The performance of this application scenario will be analyzed in Section 3.6.6 as part of this chapter's evaluation of `CACAO`.

## 3.6 Evaluation

The viability and efficiency of the proposed near-memory synchronization accelerator are evaluated in this section. `CACAO` is integrated into the FPGA-based prototype platform described in Section 2.4. In the following, the evaluation methodology is defined in Section 3.6.1. The evaluation begins with the analysis of hardware cost and attainable frequency in Section 3.6.2. Then, in Section 3.6.3, the individual operations are analyzed on sub-operation level, before a simple analytical model is derived in Section 3.6.4. Section 3.6.5 presents microbenchmark results with a particular focus on remote performance and scalability compared to the design alternatives. Finally, Section 3.6.6 evaluates the macrobenchmark application scenario that has been presented in the previous section.

### 3.6.1 Methodology

The following four quantitative metrics are used to evaluate the proposed accelerator: hardware cost, attainable frequency, duration of sub-operations, and execution time.

**Hardware cost.** The hardware cost is obtained by analyzing the FPGA resources required for the synthesized accelerator. The entire design was synthesized on the prodesign proFPGA prototyping platform consisting of four connected Virtex-7 2000T FPGAs. On these, one slice contains four LUTs, eight registers, and two muxes. Xilinx Vivado 2018.3 was used as synthesis tool with the implementation mode `performance explore`.

**Attainable frequency.** The second metric, i.e., the attainable maximum frequency of the accelerator, was extracted from a separate synthesis run of the standalone accelerator module with the same options as above. This is required since the entire many-core architecture frequency is limited to 50 MHz due to bottlenecks in other employed components.

**Duration of sub-operations.** The third metric subdivides the atomic operations in the following sub-operations: trigger time $T_{trigger}$, NoC travel time $T_{NoC}$, and actual atomic operation time $T_{atomic}$. $T_{trigger}$ is the elapsed time on the sender side from issuing the command in the CPU core until it is ready to be sent to the remote tile. Furthermore, the time to provide the response back to the initiating core is included. $T_{NoC}$ is the accumulated time for flit generation on the sender side, packet reception on the receiver side, and NoC travel time for both request and response messages. Finally, $T_{atomic}$ is the time required to execute the sequence of operations in the accelerator, including memory access latencies. In the case of local atomic operations via the accelerator, $T_{NoC}$ does not apply.

The duration of the sub-operations provides valuable insights. However, as they are very fine-grained and only last a few clock cycles, it is impossible to measure them via software time measurements without implementing specific hardware counters to extract the values. Therefore, the minimal duration of the sub-operations is obtained by waveform analysis of the synthesizable design simulated with the cycle-accurate Modelsim 10.4c RTL simulator. The simulations are carried out without background load in the system.

**Execution time.** Finally, the execution time of micro- and macrobenchmarks constitutes the fourth metric. It allows comparing the performance of different implementations. The benchmarks are executed on the CPU cores. The execution time is measured by reading a 64-bit timestamp provided by the MPSoC prototype non-intrusively at the beginning and end of the benchmark and calculating the difference between these measurements.

### 3.6.2 Hardware Cost and Frequency Evaluation

**Hardware Cost.** The proposed accelerator comes at the cost of hardware resources. Table 3.1 shows the accelerator's FPGA resource requirements when synthesized onto the prototype platform consisting of four Virtex-7 2000T FPGAs. The module is subdivided into the `CACAO` core module and the logic to receive incoming commands (RX) and send response messages (TX) back to the initiating core. Two reference values for the hardware cost are provided: a LEON 3 core and an entire tile.

**Table 3.1: Resource utilization of `CACAO`.** Synthesized on a Virtex-7 2000T FPGA where one slice contains 4 LUTs, 8 Registers, and 2 Muxes. Each per-tile `CACAO` accelerator requires approximately 4% of the slices of the five LEON3 cores per tile.

| HW Module | Slices | LUT | Register | Mux | BRAM | DSP |
|---|---|---|---|---|---|---|
| `CACAO` core | 285 | 756 | 473 | 0 | 0 | 0 |
| `CACAO` TX & RX | 230 | 503 | 307 | 36 | 0 | 0 |
| $\sum$ `CACAO` | 515 | 1 259 | 780 | 36 | 0 | 0 |
| LEON 3 core | 2 499 | 8 160 | 2 587 | 33 | 18 | 4 |
| `CACAO` % of Tile | 1.7% | 1.3% | 1.6% | 3.3% | 0% | 0% |

The `CACAO` core module contributes to roughly half of the added hardware resources (55% of slices) required for the `CACAO` accelerator. The residual half (45% of the slices) is spent for the RX and TX modules, which receive and decode incoming requests and send response messages to the initiating cores, respectively. The whole synchronization accelerator, which exists once per tile, utilizes approximately one-fifth of the slices of a LEON 3 core. Compared to the five cores incorporated in each tile, `CACAO` amounts to approximately 4% of the cores' slices. Compared to an entire tile, the proposed `CACAO` accelerator, including TX and RX interface, amounts to less than 2% of the tile's slices.

**Frequency Evaluation.** Furthermore, the attainable frequency of `CACAO` was analyzed. The standalone `CACAO` accelerator, including its TX and RX interface, would be able to be operated at 299 MHz. However, since it is integrated into the whole many-core architecture, including CPUs, Buses, the NoC, and other modules, the accelerator is operated at the same frequency as the residual design. The prototype is operated at a clock frequency of 50 MHz due to bottlenecks in other components.

### 3.6.3 Analysis of Sub-Operations

Analyzing the different atomic operations with the third metric defined above results in the cycle-accurate minimal durations provided in Figure 3.4. The stacked bar plot diagram contains a tuple of stacked bars for each hardware-accelerated atomic operation. The left bar of each tuple represents the respective local operation, including only $T_{trigger}$ and $T_{atomics}$. In contrast, each tuple's right bar represents the respective remote operation, including $T_{NoC}$.

As can be seen, $T_{trigger}$ is dependent on the number of operands required by the operations (i.e., three additional cycles per operand). Additionally, the bus arbitration and returning the response code to the CPU are included.

$T_{NoC}$ includes the time for flit generation/packetization on the sender side, NoC travel time (entire round-trip of both request and response messages), and command decoding on

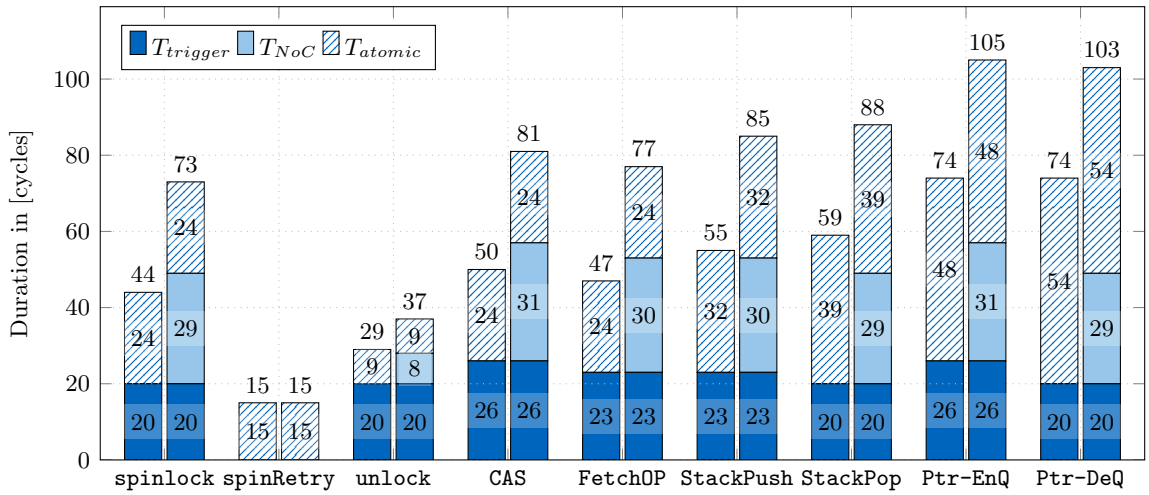



**Figure 3.4: Analysis of simulated `CACAO` sub-operations.** Cycle accurate minimal durations of individual and standalone atomic operations broken down in their integral parts.

the receiver side. This time depends on the number of operands that have to be transferred, the NoC hops, and additionally the load in the system if this analysis would not be carried out in a non-utilized system. However, this section avoids all other system activity to obtain the operations' pure minimal duration.

It can be observed that $T_{NoC}$ contributes roughly one-third to the total cycles of the respective remote operation, with a portion ranging from 22% (e.g., `unlock`) up to 40% (e.g., `spinlock`). As the overhead is relatively low for remote operations, the proposed atomic operations indicate auspicious remote performance. This is achieved by the remote near-memory execution since it minimizes the round-trips and NoC travel time. When dissecting $T_{atomics}$, this becomes even more evident. Even though, the more complex atomic operations, such as `Ptr-EnQ/DeQ`, have a comparably high $T_{\texttt{atomics}}$, $T_{NoC}$ is kept to a minimum. The five required memory accesses of the `Ptr-EnQ` operation create a higher $T_{atomics}$ than the two memory operations of the `CAS`. However, these memory operations are executed locally near the memory instead of remotely via the NoC. Thus, not five but solely one round-trip over the NoC is required, keeping $T_{NoC}$ down.

Furthermore, since the synchronization accelerator performs the operations retry-free, they must be executed only once, even if the data structure is under heavy contention. In contrast, in case of contention, a lock-based or lock-free variant needs to retry the `spinlock` or the entire `CAS` several times. This phenomenon will be analyzed in the following with the help of a simple analytical model.

### 3.6.4 Analytical Model

Based on the cycle-accurate analysis of the individual atomic operations, an analytical model is derived for the required clock cycles of the remote `shared counter increment`. The three different synchronization variants (i.e., lock-based, lock-free, and `CACAO`) are considered.

**Assumptions.** Simplified modeling is used where only the time-consuming remote memory accesses, remote atomic operations, and the number of retries due to interfering concurrency

are accounted for. Furthermore, the necessary cache line invalidation before the remote read and the cache line writeback after the remote write are incorporated into the model. Thus, the remote read including the cache line invalidation requires $T^{\texttt{remote}}_{\texttt{read}} = 130$ cycles. The remote write requires $T^{\texttt{remote}}_{\texttt{write}} = 190$ cycles. This time contains the write itself, the cache line writeback, and a write-barrier mechanism to ensure consistency, i.e., ensure that the write certainly arrived at the remote memory.

Note that the simplified model does not include the duration of local memory operations, the respective function calls, or checking the `CAS`-loop condition (e.g., loc 6 in Listing 3.2 on Page 45). These overheads explain the slight discrepancy between the numbers obtained by this model and the actual measured microbenchmark results provided in the next Section 3.6.5.

**Derivation.** When applying this model to Listing 3.1 (cf. Page 44), the following number of cycles $N_{lock}$ for the lock-based shared counter is obtained:

$$\begin{aligned} N_{lock} &= T_{loc,2} + T_{loc,3} + T_{loc,5} + T_{loc,6} \\ &= T^{\texttt{remote}}_{\texttt{spinlock}} + T^{\texttt{remote}}_{\texttt{read}} + T^{\texttt{remote}}_{\texttt{write}} + T^{\texttt{remote}}_{\texttt{unlock}} \\ &= (73 + R \cdot 15) + 130 + 190 + 37 \end{aligned} \tag{3.1}$$

where $R$ is the number of retries. Similarly, when applied to Listing 3.2 (cf. Page 45), it results in:

$$N_{free} = T_{loc,2} + T_{loc,5} = T^{\texttt{remote}}_{\texttt{read}} + T^{\texttt{remote}}_{\texttt{CAS}} = 130 + 81 \cdot (1 + R)$$

In contrast, the synchronization accelerator (Listing 3.3, Page 45) has a constant

$$N_{hw} = T^{\texttt{remote}}_{\texttt{FetchOp}} = 77. \tag{3.2}$$

**Insights.** Figure 3.5 depicts the analytical model results for the different variants of the remote `shared counter increment` in dependence on the number of retries due to interfering concurrency.

It can be observed that the lock-based variant has a relatively high base duration (413 cycles for $R = 0$) since, in addition to the spinlock, a remote read and remote write are required for the counter increment. When the spinlock experiences contention, only a small overhead of 15 cycles per retry is added as the efficient `CACAO` spinlock polls in hardware at the remote site. In contrast, the base duration with 211 cycles of the lock-free variant is shorter because only an initial remote read is required. However, when the `CAS` fails due to interfering concurrency, it is re-executed, adding 81 cycles per try. Hence, a concurrency-dependent cross-over point exists between the lock-based and lock-free variants. For no and low levels of concurrency (i.e., if the number of retries $R < 3$ in this scenario), the lock-free variant performs better than the lock-based one. Accordingly, for higher concurrency, the re-execution of the critical section of the lock-free variant dominates its duration.

Moreover, as expected, the synchronization accelerator significantly outperforms the lock-based and lock-free variants, especially for higher concurrency levels. Since the operations are retry-free, they have to be executed only once, even if the data structure is under heavy contention.

The analytical model will help to understand better the results obtained from actual microbenchmark measurements in the following sections.

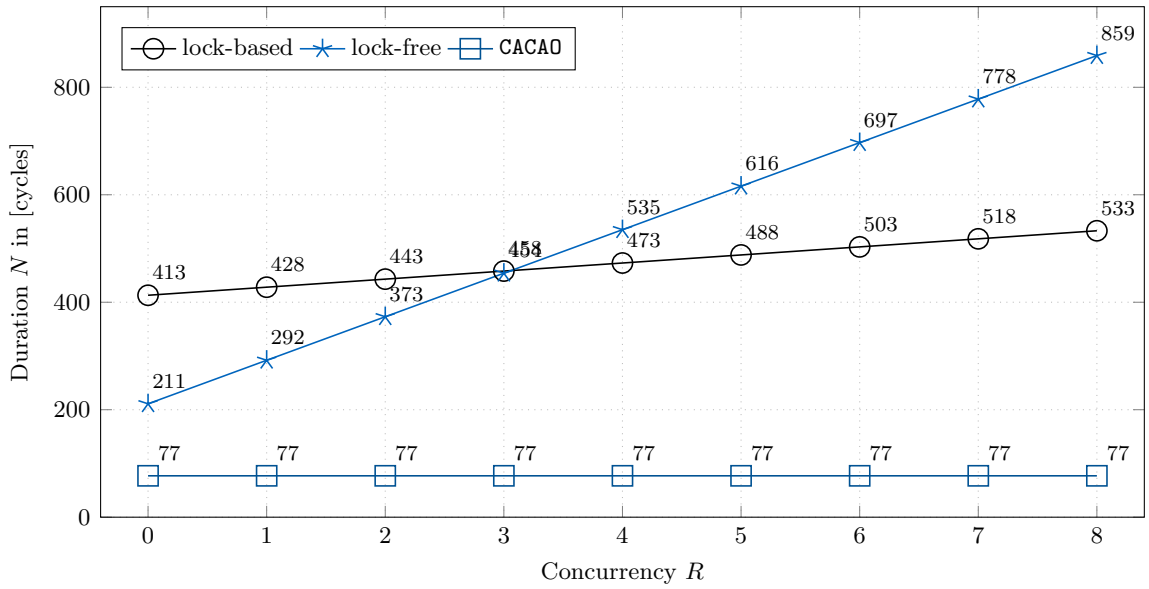



**Figure 3.5: Analytical model for the the shared counter.** Duration in number of clock cycles $N$ in dependence of the number of retries due to interfering concurrency $R$. The model predicts that `CACAO` will outperform lock-based and lock-free synchronization significantly, in particular for higher concurrency.

### 3.6.5 Microbenchmarks

The previous sections provided minimal durations of the individual atomic operations based on cycle-accurate simulations and employed them to derive a simple analytical model for the shared counter. This section investigates the performance of the proposed operations when integrated into the operating system and executed on the proFPGA prototype platform. Two microbenchmarks were developed. The first benchmark analyzes the local versus remote performance of individual operations. In contrast, the second benchmark investigates the approach's scalability when the data structures are under increasing contention. Both microbenchmarks are applied to three scenarios of concurrent data structure operations:

- `increment/decrement` a concurrent `shared counter`,
- `push/pop` to/from a concurrent `stack`, and
- `enqueue/dequeue` to/from concurrent linked-list-based `queue`.

All three scenarios are implemented and evaluated in their lock-based, lock-free, and `CACAO` synchronization variant.

#### 3.6.5.1 Benchmark 1: Analyzing Remote vs. Local Operations

The goal of the first micro-benchmark is to evaluate the performance of remote versus local synchronization. Therefore, the duration of the atomic operations experienced by the initiating CPU core is measured and compared for purely local and remote operations on the concurrent data structures. The benchmark uses one local or remote core, respectively, and avoids contention on the data structures.

**Benchmark Setup.** Without loss of generality, the data structures reside in the tile-local memory of `tile 0`. Thus, the local operations are executed by a core on `tile 0` while the remote operations are performed by a core on `tile 1`. Each atomic operation is performed 10 000 times without concurrency on the data structure to obtain reliable results. The execution time (i.e., the fourth metric) is determined in the following way. First, timestamps are taken before and after the individual operations' execution, and their difference is calculated. Moreover, the measurement error introduced by reading the timestamp is corrected by determining and subtracting the average overhead that the reading of the time stamp takes. Finally, the execution time's mean value and standard deviation are calculated over the 10 000 individual measurements.

**Results.** The resulting execution times in clock cycles are depicted in Figure 3.6 as a bar plot diagram for the three scenarios (counter, stack, and queue) and the three synchronization variants (lock-based, lock-free, and `CACAO`). Each six-tuple of bars for the six atomic

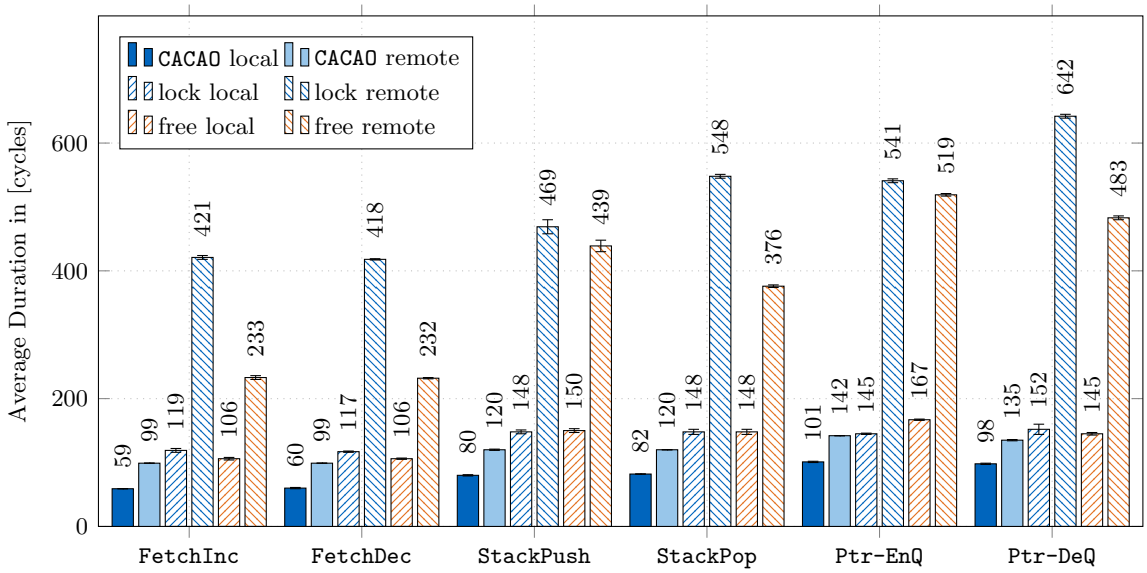



**Figure 3.6: Analysis of remote vs. local atomic operations.** Average execution time of various local and remote atomic operations for the three types of synchronization. `CACAO` is particularly beneficial for remote atomic operations and experiences only small remote overhead compared to the other synchronization variants.

operations (`FetchInc`, `FetchDec`, `StackPush`, `StackPop`, `Ptr-EnQ`, `Ptr-DeQ`) contains the local `CACAO`, the remote `CACAO`, the local lock-based, the remote lock-based, the local lock-free, and the remote lock-free synchronization variant. Additionally, the respective number of clock cycles is annotated on each bar. Four key observations can be derived from Figure 3.6:

1) The local atomic operations outperform the remote atomic operations for all three types of synchronization.
2) The more complex atomic operations on the stack or queue have a longer duration as they require a higher number of memory accesses. While, for the respective remote operations, the lock-based and lock-free variants have to perform them remotely, `CACAO` saves the remote accesses by executing the operation near-memory.

3) For both local and remote operations, `CACAO` outperforms the lock-based and lock-free variants by far. Locally, the respective hardware variant is between 1.43x (`Ptr-EnQ`) and 2.01x (`FetchInc`) better than the lock-based variant and between 1.48x (`Ptr-DeQ`) and 1.88x (`StackPush`) faster than the lock-free variant. In contrast, remotely, these numbers range from 3.81x (`Ptr-EnQ`) up to 4.76x (`Ptr-DeQ`) over the lock-based variant and from 2.34x (`FetchDec`) up to 3.66x (`StackPush`) over the lock-free variant, respectively. Thus, the advantage of hardware acceleration is significantly higher for remote compared to local operations.

4) The measured execution times of the remote `FetchInc` accord with the values obtained from the analytical model for $R = 0$ in the previous section. The reader is referred to the assumptions made for the analytical model to understand the slight deviation.

**Summary.** In conclusion, this first microbenchmark revealed that the `CACAO` accelerator is especially beneficial for remote atomic operations and outperforms the other synchronization variants by far.

#### 3.6.5.2 Benchmark 2: Scalability Analysis

While the first benchmark analyzed individual operations' performance, the goal of the second micro-benchmark is to evaluate the scalability of the proposed synchronization accelerator when the data structures experience higher levels of concurrency and contention.

Thus, the benchmark executes atomic operations on the concurrent data structures originating from varying amounts of cores. The involved cores operate concurrently on the data structures.

**Benchmark Setup.** Without loss of generality, the data structures reside in the tile-local memory of `tile 0`. The core count per tile is seven[3]. The number of involved remote tiles is varied between one and fifteen, resulting in seven up to 105 remote cores operating concurrently on the data structures. Each involved core performs 10 000 successful atomic operations on the respective concurrent data structure. If an operation fails due to concurrency-induced retries, it is repeated. Without loss of generality, each core performs 5 000 counter increment, stack push, or queue enqueue operations, respectively, and 5 000 counter decrement, stack pop, or queue dequeue operations. A proper allocation ensures that no `Ptr-EnQ/DeQ` or `StackPush/Pop` operation fails due to a full or empty queue or stack, respectively. For each number of involved remote tiles (i.e., one to fifteen), the execution time is measured by taking timestamps at the beginning and end of the entire microbenchmark. The execution time is normalized to the total number of successful operations, i.e., divided by 10 000 and by the number of involved cores, to allow for a proper comparison.

Additionally, benchmark runs with seven local cores (i.e., remote tiles equals zero) are performed as a reference.

**Results.** The resulting normalized execution times in clock cycles are depicted in Figure 3.7 for the three scenarios (counter, stack, and queue) in separate diagrams. For each

[3]Section 2.4 defined the many-core architecture with four application cores per tile. Nevertheless, this chapter performs some evaluations with up to seven cores per tile to enable higher concurrency.

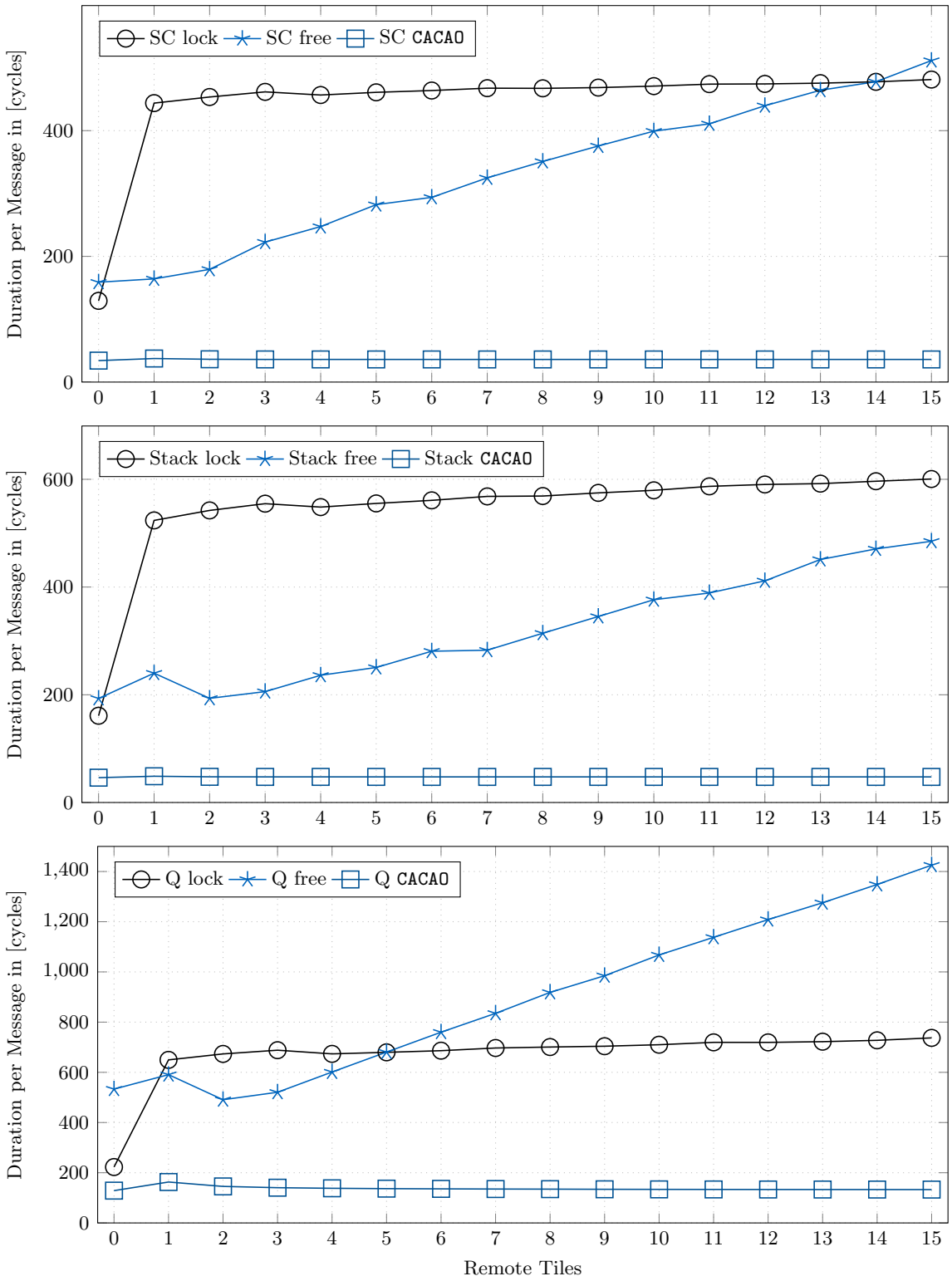



**Figure 3.7: Scalability and concurrency analysis of inter-process synchronization.** Micro-benchmark results of the concurrent counter (SC), stack, and queue (Q) for a varying amount of concurrent remote atomic operations. Zero remote tiles represent a purely local variant as a point of reference. `CACAO` has excellent scalability and enables performant inter-process synchronization between tiles.

scenario, all three synchronization variants (lock-based ⊖, lock-free ✶, and CACAO ⊟) are compared. Several insights can be obtained from Figure 3.7:

1) The three scenarios (counter, stack, and queue) behave similarly. First, CACAO outperforms the lock-based and lock-free variants in all cases. Second, there exists a concurrency-dependent cross-over point between the lock-based and lock-free variants. It is reached at 14 and five remote tiles for the counter and queue, respectively. While the cross-over point is not yet reached for the stack with the given maximum number of fifteen tiles, the trend towards it is visible.
2) CACAO does not suffer from the increased concurrency since it operates retry-free by design. Any increase in normalized execution time is due to the serialization of the requests by the accelerator.
3) The remote lock-based variants of all three scenarios already have a comparably high duration for one remote tile (i.e., seven remote cores). However, a higher number of involved remote tiles creates only minor overhead, resulting in a low slope. This is explained by the small retry overhead of the efficient remote spinlock. Moreover, as expected, the duration per operation is highest for the more complex queue operations.
4) The remote lock-free variants exhibit the behavior as predicted by the analytical model. With higher levels of concurrency, the number of retries increases and, accordingly, the time until the operation succeeds. The queue scenario suffers most from higher concurrency since the critical sections of the queue operations are longer than those of the counter or the stack operations. Thus, it is more susceptible to interfering concurrency and retries, resulting in the steepest slope of the lock-free curve. The analysis of the lock-free counter and stack operations reveals why the less complex counter scenario has an earlier cross-over point. It is dependent on the ratio between the critical and non-critical sections of the operations. The higher this ratio, the higher the risk of interfering concurrency. Both critical sections consist of an increment/decrement. In the case of the stack operations, it is the increment/decrement of the stack index pointing to the next free stack slot. However, the non-critical section of the counter operations is negligible, while the non-critical section of the stack operations consists of the non-atomic write/read of the current stack element to/from the atomically obtained pointer.

**Summary.** In conclusion, CACAO's scalability is very good and enables performant synchronization in large tile-based many-core architectures with at least 100 cores.

### 3.6.6 Marcobenchmark

The previous microbenchmark analysis evaluated the accelerated remote atomic operations under contention in stress test scenarios. Nevertheless, solely the standalone atomic operations were executed on the concurrent data structures. This section embeds and evaluates the atomic operations in the application scenario presented in Section 3.5.6 to evaluate the approach's viability in a broader scenario. The `concurrent stack` is utilized to build and manage an efficient inter-tile memory allocator, which is integrated into the PGAS inter-process communication mechanism required by the X10 IMSuite benchmarks (cf. Section 2.4.3). The evaluation aims to quantify the performance speedup of the whole application when using the efficient inter-tile allocator.

**Evaluation Setup.** The benchmarks are executed on the FPGA-based prototype platform described in Section 2.4. The benchmark execution times of the twelve IMSuite benchmarks in the 4x4 – single configuration (i.e., with one memory tile in use) are measured and compared for both variants of remote memory allocation: the conventional way via task scheduling (`malloc`) versus the synchronization accelerator variant (`CACAO`). The comparison against a lock-based or lock-free variant of the stack-based memory allocator is omitted as the previous microbenchmark analysis revealed `CACAO`'s superiority over them.

**Results.** Figure 3.8 depicts the relative benchmark execution times of the twelve benchmarks for both variants as a bar plot diagram. Each tuple of bars is normalized to the respective `malloc` variant. It can be observed that the benchmarks profit to a different

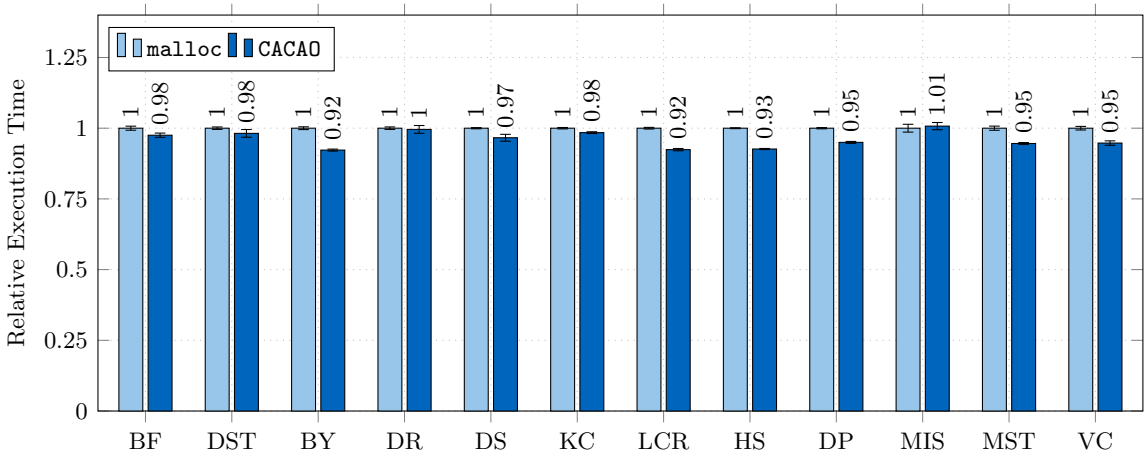



**Figure 3.8: IMSuite benchmark results of the `CACAO` allocator.** Whereas a speedup of 19.3x was achieved for standalone `CACAO` allocations compared to normal remote allocations, the integration into the IMSuite benchmarks revealed an improvement of up to 8% (4% on average). The performance is compelling since the accelerated functionality accounts for only a minor part of the inter-process communication and an even smaller portion of the entire application.

extent from `CACAO`. While the overall benchmark execution times of the BY and LCR benchmarks are reduced by 8%, the MIS benchmark suffers a minor degradation. On average, the twelve benchmarks show a reduction of 4%.

These numbers are very good when put into perspective. When considering the standalone remote memory allocation, the `CACAO`-based variant requires $T^{\texttt{CACAO}}_{\texttt{malloc}} = 220$ clock cycles while the task migration variant is 19.3x slower with $T^{\texttt{task}}_{\texttt{malloc}} = 4240$ clock cycles. It is evident that the standalone speedup is not realizable for the entire application since the allocation and setup of the metadata buffer (steps (1) to (4) in Section 3.5.6) are solely minor sub-tasks of the whole benchmark.

**Summary.** The atomic stack push and pop operations were utilized to build and manage an efficient inter-tile memory allocator. Since the memory allocations account for only a minor portion of the inter-process communication mechanism and an even smaller portion of the entire application, the average performance speedup of 4% of the entire application is compelling.

## 3.7 Summary

Driven by the need for efficient synchronization primitives for tile-based architectures, this chapter presented hardware-accelerated remote atomic operations. The goal was to alleviate the overhead of existing inter-process synchronization mechanisms. The proposed synchronization accelerator is a simple yet crucial run-time support near-memory accelerator (RTS-NMA) that supports important concurrent data structures. It is a hardware-software co-design approach that provides hardware-accelerated remote atomic operations on software-defined concurrent data structures. The evaluation revealed how lightweight, scalable, and performant the approach is.

While the proposed accelerator operates on word granularity (i.e., only pointers to elements are enqueued/dequeued to/from a queue or pushed/popped to/from a stack), the allocation of new elements and the transfer of the element payload remains a software and decoupled task, respectively. The next chapter will extend the approach to a full-fledged software-defined and hardware-accelerated queue management, including data transfers, memory allocation, and receiver notification.

# 4 Dynamic Software-Defined Hardware-Managed Queues

> "Normal people believe that if it ain't broke, don't fix it. Engineers believe that if it ain't broke, it doesn't have enough features yet."
>
> — Scott Adams [187]

The previous chapter focused on supporting and accelerating lightweight synchronization primitives for tile-based architectures. The evaluation revealed the benefit of remote near-memory acceleration of atomic operations to reduce inter-process synchronization overhead. However, since the approach only dealt with pointers to the queue or stack elements, the actual element payload had to be handled independently. Although this is feasible for usual shared-memory architectures, it can create data-to-task locality issues on tile-based architectures with distributed-shared memories.

This chapter presents a contribution that mitigates this challenge and facilitates efficient inter-process communication through dynamic software-defined hardware-accelerated queues. It is an advancement of the previous chapter's approach. The data-to-task locality is not only provided for synchronization primitives but extended to inter-process communication, including data transfer. A smart hardware-software interaction is proposed that combines the performance of hardware acceleration with the flexibility of software queues.

The contribution is further motivated in Section 4.1 with a refined problem statement. Section 4.2 defines the aspired goals and requirements of the approach followed by Section 4.3 which explains the assumptions made and the given boundary conditions. Then, Section 4.4 explores two design alternatives before the concept of the near-memory queue management accelerator is presented. In Section 4.5, the architectural details of the hardware-software co-design approach are provided. Furthermore, the integration of the accelerator into the MPI library is described. Next, Section 4.6 covers the evaluation with micro- and macrobenchmarks before the chapter is concluded in Section 4.7.

While this chapter investigates queue-based communication, the next Chapter 5 targets a more complex scenario where object graphs have to be transferred.

## 4.1 Motivation and Problem Statement

Multi-threaded applications, libraries, and operating systems enable to leverage of the increased scalability and compute performance of many-core architectures. However, parallel programming often experiences a high demand for communication between threads. Typical patterns of communication consist of data transfer and subsequent data processing. A suitable and beneficial data structure to facilitate such communication is a queue since it retains the order of elements and functions as a buffer. Queues are employed in central

locations of many applications, libraries, and operating systems to support, for instance, task scheduling, data distribution, or message-passing. Since queues possess such a crucial role in various scenarios, the need for an efficient queue implementation is inevitable.

As queues, in general, are not a new phenomenon, there exist quite many scalable software queue implementations. However, since their main field of application is standard shared memory systems with a centralized memory, they are optimized for solely handling pointers to queue elements. There is no necessity to copy the actual element payload since all threads operate on the same physical memory.

The attempt to translate these well-established queuing concepts to tile-based many-core architectures poses new and different challenges. At the same time, research efforts in this direction are younger and less mature. First, for inter-process queue operations where data may reside in physically distributed memories, it is not sufficient to only manage pointers to queue elements. Otherwise, the element payload would remain in the sender's memory, and the receiver would need to fetch the payload manually via costly remote memory accesses. Thus, data-to-task locality would be at risk if the payload is not transferred between the respective memories. Second, managing inter-tile queues in software is accompanied by significant software overhead. Especially the scheduling overhead and costly remote round-trips hinder the efficiency of such approaches. A similar observation was made in the previous chapter when handling remote atomic operations in software.

Therefore, tile-based architectures long for an efficient and scalable inter-tile queue management, which tackles these critical challenges while still providing the flexibility known of software queues. The specific goals and requirements will be discussed in the next section.

## 4.2 Goals and Requirements

The overall goal of this chapter's contribution is to facilitate efficient, scalable, and yet flexible inter-process communication on tile-based architectures. The insights obtained from the previous chapter's remote atomic operations are applied and extended to full-fledged queue management. A hardware-software co-design approach is envisioned to enable higher degrees of freedom for the design and make use of them. The following requirements are demanded of the approach. They combine hardware-accelerated performance with software-defined flexibility.

**Performance** Since communication often resides in the critical path of execution, the approach must be efficient, exhibit low latencies, and outperform existing hardware- and software-based solutions.

**Data Locality** Data-to-task locality is crucial for the performance of tile-based architectures. Thus, the approach must procure it for both local (intra-tile) and remote (inter-tile) inter-process communication. Unnecessary local data transfers should be avoided. However, remote inter-tile transfers should be incorporated.

**Scalability** As many-core architectures contain dozens or hundreds of cores and several memories, the proposed concept must be scalable to these system sizes. Furthermore, several queues should be manageable.

**Flexibility** Since queues have various use-cases, the approach must be flexible to comply with them. Software-defined general-purpose queues with user-defined length and ar-

bitrary element size are desired. This guarantees the flexibility and configurability of software queues that can be created and destroyed dynamically at run-time.

**Adaptability to Run-Time Requirement Changes** As run-time requirements are prone to change during an application, the approach should be adaptable to them. Those changes may be application-intrinsic due to different application phases or application-extrinsic, e.g., changing amounts of resources available to the application or varying input data.

**Resource Awareness** As run-time adaptability and resource awareness are essential attributes of applications, the queues should contribute to this goal. The approach should be capable of monitoring and dynamically adjusting the queue's memory footprint at run-time. Thereby, it can react to varying queue fill levels and free unneeded memory, which is beneficial for long queues or large element sizes.

**Minimal Software Overhead** The queue management and monitoring should be outsourced to hardware to achieve flexibility, adaptability, and resource awareness with minimal software overhead. The hardware-software interaction should be lightweight and hardware-controlled during operation to not impede the performance of the approach. Only the queue creation and rare operations that cannot be performed in hardware should remain software tasks.

Fulfilling these functional and non-functional requirements would facilitate efficient and effective inter-process communication with the flexibility known of software queues.

## 4.3 Assumptions and Constraints

After defining the desired goals and requirements of the approach, this section elaborates on the assumptions made and the given boundary conditions for the design. If one intends to apply the concept to other architectures or software environments, the assumptions and constraints must be considered.

**Access to Hardware Registers** As for `CACAO`, it is assumed that direct hardware register access to interact with the accelerator in a lean manner is possible. It needs to be assessed if accelerator access without memory protection is allowed on other systems.

**No MMU or Virtual Memory** As for `CACAO`, the prototype design does not use an MMU or virtual memory. If one desires to incorporate those, it would have to be ensured that the appropriate address translations are available to the accelerator.

**Synchronous and Asynchronous Calls** In contrast to `CACAO`, the accelerator calls can additionally also be asynchronous so that a notification of the completion becomes necessary. It is assumed that the accelerator can interact with software in such an asynchronous manner. While other platforms might be limited to sending interrupts from the accelerator to the cores, the contribution relies on direct accelerator interaction with the scheduler. Although the concept is also conceivable without a hardware scheduler, it enables seamless hardware-software interaction, as will be explained next.

**Interruptless Hardware Scheduling** The design leverages the existing hardware scheduler, which receives task descriptors from cores, network adapters, or accelerators and inserts them into the cores' job queues. Thus, up-calls to software can be performed without

interrupts. In collaboration with the network adapter, task descriptors can also be inserted into remote hardware schedulers without software involvement. A further assumption or constraint is that the used operating system has to comply with and allow the usage of the hardware scheduler. Additionally, the software engineers need to consent to use a hardware scheduler instead of e.g., the standard Linux software scheduler. Although the hardware scheduler and the used OctoPOS operating system (cf. Section 2.4.2) are no conceptual requirements, they enable optimizations that would not be possible without them.

**Access to other Hardware Units** It is assumed that other existing hardware units like the DMA unit can be reused to minimize the resource and other overhead. In the case of the network adapter's inter-tile DMA unit, this implies exchanging necessary information (e.g., the completion notification or the destination address of a data transfer) between the DMA unit and the proposed accelerator. Thus, minor modifications of the NA need to be allowed, and possible to forward such information directly. Otherwise, undesired additional round-trips over the NoC may become necessary.

**Queue Representation** It is assumed that the queue representation is compatible with hardware acceleration, i.e., the accelerator has to be able to retrieve all necessary information about a particular queue from memory without up-calls to software. This includes e.g., the queues' internal state, their memory management, or the interaction with the hardware scheduler.

**No Concurrent Queue Access by Software** It is assumed that access to the queues usually is only allowed via the accelerator. Thus, the software is not allowed to operate concurrently on the queues. If software access to a queue becomes necessary, a dedicated locking mechanism provided via the accelerator must be used. Through this exclusiveness, it is ensured that the accelerator can operate undisturbed.

**Bounded FIFO Queue** The design is limited to bounded FIFO queues with software-defined but limited queue and element size. The boundedness simplifies the queue's memory management which can be handled by the accelerator entirely. Otherwise, undesired and costly up-calls to software would be required for dynamic memory allocations and deallocations on every queue operation. Furthermore, the current design only supports FIFO queues and e.g., no priority queues.

**Message-Passing Communication** As the design targets message-passing communication, it does not assume or require inter-tile cache coherence. The near-memory accelerator operates on the respective memory directly so that the queue operations and remote data transfers bypass the caches.

## 4.4 Exploration and Concept

With the goals, requirements, assumptions, and constraints defined in the previous two sections in mind, this section first explores two design alternatives in Section 4.4.1 before the concept of the actual contribution is presented in Section 4.4.2.

### 4.4.1 Exploring Design Alternatives

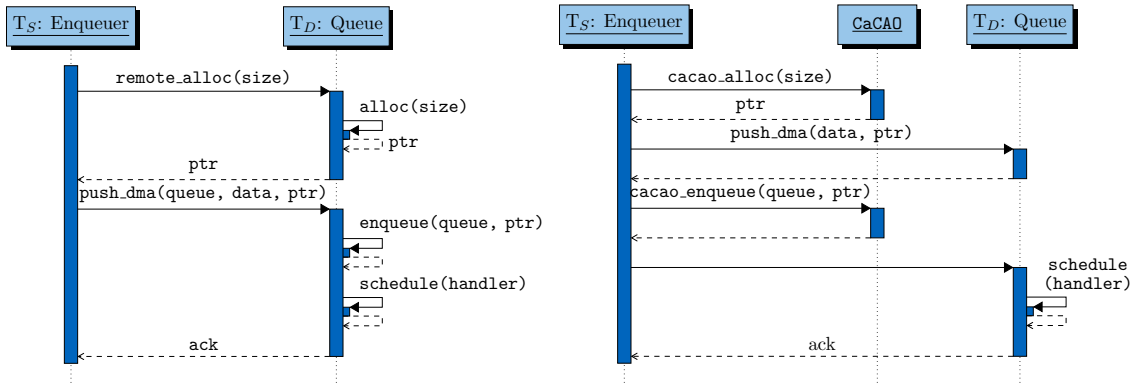
Even if today's tile-based architectures are not equipped with dedicated queue managers, they provide certain hardware support for inter-process communication between tiles, such as remote procedure calls (RPCs), remote task invocation/scheduling, and hardware-assisted DMA. These enable a software-managed queue implementation (Section 4.4.1.1) based on remote DMA and remote task invocation. Moreover, when extending the portfolio with the remote atomic operations presented in the Chapter 3, another design alternative arises (Section 4.4.1.2).

#### 4.4.1.1 Software-Managed Queue

A software-managed queue can be realized when employing the existing hardware support for inter-process communication (i.e., remote task invocation and DMA). Figure 4.1a illustrates the necessary sequence of operations to perform a remote software-managed enqueue from a core on the sender tile $T_S$ to a queue residing in the memory on a remote tile $T_D$:

1) Allocation of a destination buffer where the element should be transferred to achieve data locality. This step requires a remote task invocation. Thus, CPU time is consumed both on the source tile $T_S$ and destination tile $T_D$.
2) DMA transfer of the queue element from $T_S$ to $T_D$ with subsequent task invocation on $T_D$. Note that without the advanced direct task invocation feature of existing DMA engines [176], another round-trip would be required.
3) A core on $T_D$ performs the atomic enqueue operation of the element pointer.
4) A handler task is scheduled on $T_D$ to process the enqueued element.
5) A response or acknowledgment is signaled back to the sender.

Although this software-centric approach achieves the required data-to-task locality through the hardware-assisted DMA transfer, several drawbacks still arise. First, at least two round trips with the accompanied dispatching and scheduling overhead are required. Second, CPU time is consumed on both tiles even without considering the advanced requirements such as run-time monitoring, resource awareness, and run-time adaptability.




**(a)** Software-Managed Queue. **(b)** CACAO-Managed Queue.

**Figure 4.1: Comparison of remote queue design alternatives.** Message sequence charts showing a remote enqueue operation of both design alternatives from a core on tile $T_S$ to a queue in the memory of a remote tile $T_D$. Both variants have plenty of software overhead.

#### 4.4.1.2 CaCAO-Managed Queue

When additionally including the remote atomic operations into the software-managed queue approach, certain software overhead can be reduced on the remote tile $T_D$. Then, the sequence of operation depicted in Figure 4.1b is required:

1) The destination buffer allocation could be replaced by the efficient inter-tile allocator proposed in the previous Chapter 3 (cf. Section 3.5.6).
2) DMA transfer of the queue element from $T_S$ to $T_D$.
3) Remote atomic enqueue of the pointer to the queue element into the queue.
4) Remote task invocation of a handler task on $T_D$ to process the enqueued element.
5) A response or acknowledgment is signaled back to the sender.

Although the remote atomic operations reduce the overall software overhead on the remote tile $T_D$, a remote task invocation of the handler task is still required since the sender and receiver are not decoupled. Furthermore, the advanced requirements are not yet supported and would lead to CPU time on the remote tile.

**Summary.** In conclusion, the two design alternatives use existing hardware support for inter-process communication between tiles. However, since they still face several drawbacks (i.e., software overhead and round-trips), more efficient support for inter-tile queue management would be a precious extension to the portfolio of inter-process communication primitives.

### 4.4.2 Concept of the Near-Memory Queue Management Accelerator

Therefore, this section presents the concept of the proposed near-memory queue management accelerator `SHARQ` (software-defined hardware-managed queues), which facilitates efficient inter-process communication. The contribution adheres to the goals and requirements defined in the previous Section 4.2. While this section provides a brief conceptual overview of the approach, Section 4.5 describes the respective details.

`SHARQ` builds upon the insights gained from the Chapter 3 (outsourcing the performance-critical atomic operations to a remote near-memory accelerator mitigates the software overhead of inter-process synchronization). `SHARQ` extends the approach to full-fledged queue management based on a hardware-software co-design with minimal software overhead.

#### 4.4.2.1 Dynamic, Software-Defined, and Hardware-Managed

`SHARQ` features dynamic, software-defined, and hardware-managed queues. To fulfill the design requirements, `SHARQ` is

- **software-defined** to enable configurability and flexibility during queue creation,
- **hardware-managed** to achieve performance, scalability, and minimal software overhead, and
- **dynamic** to adapt to run-time requirement changes and be resource-aware.

The incidence and performance requirements of specific operations determine whether they are hardware or software tasks. Complex, rare, and performance-uncritical operations

(e.g., queue creation) remain software tasks, while performance-critical operations and monitoring tasks are outsourced to the SHARQ hardware accelerator.

The flexibility known of software queue approaches is achieved by software queue creation. SHARQ queues of user-defined length and element size are dynamically allocated and defined by software in any existing memory of the architecture. Thus, no dedicated hardware resources are utilized to store queue metadata or elements. Thereby, the approach is scalable since it is independent of the number and layout of the queues. Each so-called **queue descriptor** resides in normal memory and contains all information characterizing the particular queue. Since the queue descriptor's sole existence in memory is sufficient, and no registration with the SHARQ hardware accelerator is required, an indefinite number of individually software-defined queues can be managed.

Hence, the queue descriptor is the interface of the proposed hardware-software co-design approach. The following subsections elaborate on three key attributes of the concept and how they satisfy the design requirements:

- achieving **data-to-task locality** (Section 4.4.2.2),
- leveraging **hardware acceleration** (Section 4.4.2.3), and
- providing **smart hardware-software interaction** (Section 4.4.2.4).

#### 4.4.2.2 Remote Near-Memory Execution / Data-Locality

As discussed initially in this chapter's problem statement, it is insufficient to handle pointers to queue elements on tile-based architectures with distributed-shared memory. Data-to-task locality will be at risk if the sender and receiver do not operate on the same physical memory. To conquer this challenge of distributed architectures, SHARQ inherently couples the queue operations with element payload data transfers via the existing DMA engine. In contrast, both design alternatives' mechanisms have the DMA decoupled from the queue operation as they are missing the proposed hardware support.

While the data transfer is mandatory for remote inter-tile queue operations to guarantee data-to-task locality, data transfers inside the same memory should be avoided. Thus, for local intra-tile operations, it is apparent to manage pointers to queue elements solely.

Although this hybrid approach is desired, especially for medium and large element sizes, it creates additional complexity for the hardware-software interaction. The memory management during queue operations is entirely and atomically handled by the SHARQ accelerator, which obtains the necessary information about free queue slots from the queue descriptor. Initially, all queue buffer slots are owned by hardware. However, pointer-only queue operations change the number of hardware-owned buffers since e.g., a pointer-only enqueue operations adds a previously software-owned buffer to the queue. This may lead to an inconsistent state of the queue. Therefore, additional lightweight operations and the ownership concept are introduced, enabling the hybrid, data locality-aware approach of remote queue operations, including data transfers and local pointer-only queue operations.

Furthermore, the concept of remote near-memory execution already known from the remote atomic operations is applied. Instead of operating on a queue remotely, the queue management is outsourced to the respective SHARQ hardware accelerator located in the tile where the queue resides in memory.

#### 4.4.2.3 Queue Management Acceleration

This chapter's problem statement further pointed out that software-managed inter-tile queues impose significant software overhead. Therefore, performance-critical operations are outsourced to the SHARQ hardware accelerator. As mentioned above, each tile contains one SHARQ accelerator. It can be triggered by any local or remote core in the system and is responsible for managing every queue resident in the memory of the respective tile. The SHARQ queue management comprises the following hardware-performed tasks. They are the basis for the smart hardware-software interaction described in the next section.

**Queue operation management** The SHARQ accelerator entirely handles the atomic enqueue and dequeue operations. It supports multi-producer multi-consumer (MPMC) queues that can be accessed locally intra-tile and remotely inter-tile.

**Queue memory management** The SHARQ accelerator is also responsible for queue memory management. All hardware-owned element buffers are pre-allocated by software at the time of queue creation, and a reference to them is inserted into the queue descriptor. Thus, SHARQ can obtain and return element buffers from and to the queue descriptor without requiring performance-degrading up-calls to the software allocator on every queue operation. Thereby, the software overhead is reduced significantly.

**Optional data transfer** As explained in the previous subsection, inter-tile data transfers are intrinsically coupled with the existing DMA engine of the network adapter. The intra-tile data transfer via a local-DMA engine is optionally possible.

**Handler task invocation** The SHARQ accelerator is capable of conditionally invoking an optional handler task. It is introduced to enable user-defined processing of queue elements in a decoupled manner, i.e., the SHARQ accelerator, and not the sending core is responsible for triggering the receiver. It is scheduled by the SHARQ accelerator following a particular contract between hardware and software that will be explained later.

**Run-time monitoring** To enable memory resource awareness and adaptability to run-time requirement changes, the SHARQ accelerator is capable of run-time monitoring the current queue state on the fly.

**Memory management task invocation** Depending on the run-time monitoring, the SHARQ accelerator is capable of invoking another task via the hardware scheduler, the user-defined memory management task. The corresponding task descriptor is deposited in the queue descriptor. The hardware-software interaction will be explained next.

#### 4.4.2.4 Smart Hardware-Software Interaction

As mentioned before, the hardware-software co-design approach features handler tasks, dynamic queue memory management support, and an ownership concept for element buffers. These aspects will be discussed in the sequel.

**The Handler Task.** The concept of the handler task is proposed to enable decoupled processing of queued elements. At the time of queue creation, the optional handler task is software-defined, and its task descriptor, as well as its maximum concurrency (i.e., how many instances of the task may run in parallel), are stored in the queue descriptor. The handler task is conditionally scheduled on the queue's tile and intended for dequeuing and

processing the elements of the particular queue. In contrast to both design alternatives, where the sender was responsible for invoking the handler task on the remote tile, the responsibility is put on the SHARQ accelerator. It is a hardware-controlled smart wake-up mechanism. Hardware and software adhere to a specific contract which will be explained later in Section 4.5.3.1.

**Dynamic Memory Management Support.** The hardware-software co-design approach supports the desired resource awareness and adaptability to run-time requirement changes. Therefore, the SHARQ accelerator is equipped with a run-time monitoring capability of the queues' memory footprint. Upon user-defined conditions, the accelerator is able to schedule a software-defined memory management task. The whole interaction has minimal software involvement since the monitoring and initiation are hardware duties. Solely, the infrequent execution of the memory management task, which is allowed to run concurrently to normal queue operations, is a software responsibility. The idea behind this task is to dynamically allocate additional or free unused queue element buffers if certain user-defined thresholds of the queue fill level are reached. Thereby, the queue's memory footprint can be adapted to changing run-time requirements. It can be ensured that the queue can be operated with safe fill levels so that, on the one hand, no memory is wasted, and on the other hand, enough free queue slots are available that a full queue is avoided.

**Ownership Concept.** The ownership concept for the queue element buffers is required to enable dynamic memory management and pointer-only queue operations. If solely regular (i.e., not pointer-only) enqueue and dequeue operations are available, on each enqueue operation, the SHARQ accelerator would get a free hardware-owned buffer slot from the queue descriptor, copy the payload into it, and insert its reference into the queue. Thus, the queue memory management and the queue operations are coupled and fully hardware-controlled. However, in case of the dynamic memory management, the memory management task must be able to insert software-owned or retrieve hardware-owned buffer slots to and from the queue descriptor independent of enqueue or dequeue operations. Likewise, the pointer-only operations solely transfer the ownership of the software-owned buffer to hardware on an enqueue and vice-versa for a dequeue operation.

#### 4.4.2.5 Summary

The SHARQ hardware-software co-design approach facilitates efficient, scalable, and data locality-aware inter-process communication. The SHARQ accelerator is responsible for queue operations and memory management, data transfers, task invocation, and run-time monitoring. The flexibility and configurability known of software queues are provided as SHARQ queues are software-defined. Moreover, through smart hardware-software interaction, minimal software overhead is achieved. Hardware monitoring and task invocation enable resource awareness and adaptability to run-time requirements changes.

## 4.5 Architecture and Implementation

The previous section gave a brief conceptual overview of the SHARQ concept. This section presents several architectural and implementation details of the hardware-software co-design approach. First, the software-defined aspects are expounded in Section 4.5.1. Sec-

tion 4.5.2 explains the hardware-software interplay, before the advanced hardware-software interaction is covered in Section 4.5.3. Then, the hardware architecture of the SHARQ accelerator is described in Section 4.5.4. Moreover, Section 4.5.5 presents a software-emulated variant of SHARQ comprising the same features to provide a reference for evaluation. Finally, Section 4.5.6 explains how SHARQ is applied to and integrated into the MPI library.

### 4.5.1 Software-Defined Queue Creation

The desired flexibility known of software queues is achieved by software-defined queue creation. The operating system dynamically allocates SHARQ queues in any existing memory of the system via the standard `malloc()` API. The resulting `queue descriptor` contains all information characterizing the particular queue and acts as the interface between hardware and software. Since no information about the queues (neither queue metadata nor queue elements) is stored in the SHARQ accelerator, the approach is scalable as it is independent of the number and layout of the queues. Thus, an indefinite number of individually configured queues can be created by software and managed by hardware. Upon creation, the user receives a unique queue handle (i.e., a pointer to a queue descriptor), identifying the particular queue. When initiating queue operations, it is sufficient to pass the queue handle to the SHARQ accelerator, which obtains all required information from the queue descriptor.

**The Queue Descriptor.** The layout of the proposed `queue descriptor` is depicted in Figure 4.2. It consists of four major parts (queue metadata, allocator stack, bounded buffer, and payload buffers) and is highly configurable. As illustrated, except for the payload buffers, the individual parts form contiguous memory blocks. However, the four parts can be allocated at different positions in the same memory. In the sequel, their roles and configurability options will be explained.

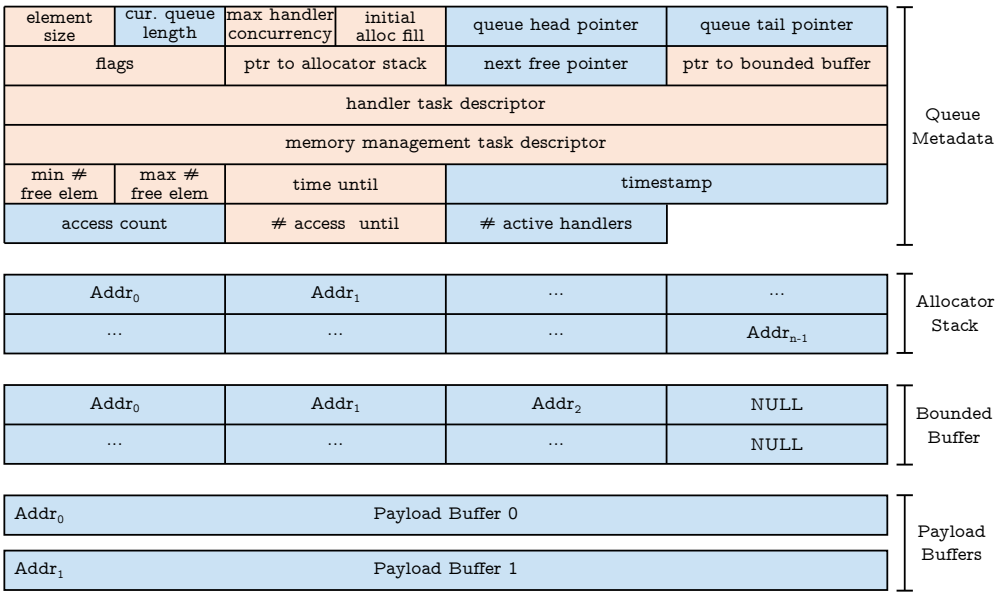



**Figure 4.2: SHARQ queue descriptor.** Layout of the queue descriptor consisting of queue metadata, allocator stack, bounded buffer, and payload buffers. The queue descriptor serves as interface for the hardware-software interplay with the SHARQ accelerator. Software-configurable fields are highlighted in orange. Hardware-managed parts are highlighted in blue.

**Queue Metadata** The software-defined queue properties, highlighted in orange, are stored in the queue metadata section. These encompass the initial queue length (i.e., the initial fill level (`initial alloc fill`) of the allocator stack), the maximum `element size`, the maximum `concurrency of the handler task`, the `handler task descriptor`, the `memory management task descriptor`, and various thresholds for the dynamic memory management (`min # free elem`, `max # free elem`, `# access until`, `time until`). All these fields are user-configurable at the time of queue creation. Moreover, the `allocator stack ptr` and the `bounded buffer ptr` are stored in the metadata section by software during queue creation, referencing the memory address of the allocator stack and bounded buffer, respectively. Furthermore, the queue metadata section stores the queue's internal state so that no queue information needs to be stored in the accelerator itself. The respective fields are highlighted in blue and are fully controlled by the `SHARQ` accelerator. These are the `current queue length`, the `head pointer` and `tail pointer` of the bounded buffer, the `next free ptr` on the allocator stack, the `# active handlers`, a `timestamp` and `access count`.

**Allocator Stack** The allocator stack is required for the hardware-controlled queue memory management. It holds the pointers to currently unused queue element buffers. At the time of queue creation, the software pre-allocates these element buffers and inserts a reference to them into the allocator stack. Thereby, they become hardware-owned. During enqueue and dequeue operations, the `SHARQ` accelerator fetches or returns element buffers from and to the allocator stack. Thereby, no up-calls to the software are required for the queue's regular memory management.
Through `SHARQ`'s dynamic memory management capability, the fill level of the allocator stack can be changed during run-time and thereby adjusted to the queue's memory footprint. Thus, it is possible to begin with an initially only partly-filled allocator stack. This is particularly beneficial for large queue element sizes. Note that the length of the allocator stack and not its initial fill level determines the maximum queue length.

**Bounded Buffer** As `SHARQ` queues are bounded FIFO queues, a ring buffer that stores the order of queue elements is required. Note that the bounder buffer solely contains pointers to the queue elements, while the element payload resides in the payload buffers. The bounded buffer is allocated by software during queue creation and demands no initialization. Its length is identical to the length of the allocator stack. The bounded buffer is fully hardware-controlled by the `SHARQ` accelerator.

**Payload Buffers** The payload buffers contain the queue element payloads. These can be allocated at arbitrary locations, however, inside the same memory. The hardware-owned payload buffers are also pre-allocated during queue creation, and their reference is inserted into the allocator stack. They are only used for regular enqueue or dequeue operations, including data transfers. In contrast, when pointer-only enqueue or dequeue operations are performed, the element buffers are inserted or retrieved to or from the bounded buffer, respectively. However, no interaction with the allocator stack is required since the payload is not copied into or from a payload buffer.

**Queue Re-Configuration.** The queue descriptor is allocated and configured by software during queue creation. During operation, it is fully hardware-controlled. Nevertheless, when required due to unforeseen requirements changes, it is still possible to re-configure the fundamental software-defined properties of a particular queue at run-time. However, as this

is a significant queue change, it is not allowed concurrently with normal queue operations. Therefore, a queue locking mechanism is provided, which blocks the SHARQ accelerator from accessing a specific queue while the software re-configures the queue properties. Note that the SHARQ accelerator can operate on other queues without constraint during this time. Moreover, the re-configuration of a queue should not be confused with the dynamic memory management support of SHARQ where free element buffers are inserted or retrieved to and from the allocator stack. The latter are performed concurrently via the additional atomic sharqAllocatorPush/Pop operations available to the software API.

### 4.5.2 Hardware-Software Interplay

Upon the software-defined queue creation, each SHARQ queue is uniquely and completely characterized by its queue handle pointing to its queue descriptor. It serves as the interface for the hardware-software interplay explained in this section.

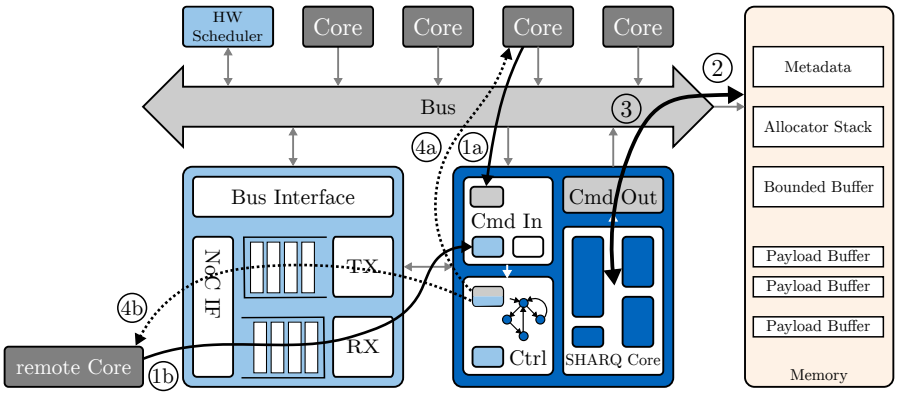



**Figure 4.3: Hardware-software interaction of SHARQ.** Interaction of local and remote cores with the SHARQ accelerator. The SHARQ queues reside in normal memory and are accessed via the bus. SHARQ obtains all required queue metadata from the queue descriptor in step (2) and performs the sequence of steps necessary for a particular queue operation in step (3).

An instance of the SHARQ accelerator is present in every tile and responsible for handling all operations on the queues residing in this tile's memory. As illustrated in Figure 4.3, the SHARQ accelerator is connected to the bus and network adapter. It can receive requests from local cores, access the tile-local memory, and interact with the hardware scheduler through the bus. Moreover, via the NoC and network adapter, the SHARQ accelerator is accessible from any remote core in the system and is able to receive and initiate remote data transfers coupled to remote enqueue and dequeue operations, respectively.

Figure 4.3 further illustrates the interplay between the cores and the SHARQ accelerator during various queue operations. They will be described in more detail for the individual queue operations in Section 4.5.4.3. Whenever an operation is outsourced to the accelerator, the following basic steps are performed:

1) SHARQ receives a request including the queue handle from a local (1a) or remote (1b) core. Thus, the software involvement is minimal as cores only initiate the operations.
2) SHARQ accesses the queue descriptor (residing in the memory) corresponding to the queue handle to retrieve the required queue metadata for the particular operation.

3) SHARQ performs the necessary sequence of steps for the particular queue operations (e.g., accessing the allocator stack or bounded buffer, updating the queue metadata, interacting with the DMA engine for data transfers, performing the run-time monitoring, and interacting with the hardware scheduler).
4) SHARQ sends a response to the initiating core (4a for local; 4b for remote).

Most of the supported queue operations are implemented synchronously and receive a response code in step 4). Exceptions will be covered in Section 4.5.4.3. As each core is limited to one pending request, the maximum number of incoming requests to one SHARQ accelerator is given by the number of cores in the system. No additional buffer for incoming requests is required since they are buffered in the NA's existing virtual channel FIFOs.

**Software API.** Since the software overhead of the SHARQ operations should be minimal, a thin API wrapper is used. The hardware operations are initiated by writing the respective commands and completed by reading the response code to and from particular hardware registers, respectively. These registers are accessible without locking within the driver code since concurrent access to them is ensured by hardware via bus split transaction. The read operation on the result register only returns once the SHARQ accelerator sends back the response code to the initiating core. This mechanism enables lightweight usage with minimal software overhead.

During operation, SHARQ queues are under full hardware control, and solely the SHARQ accelerator is allowed to access them. Figure 4.4 lists all SHARQ operations that are exposed to the user via the software API. Whenever the software needs to access a queue (e.g., to re-configure or destroy a queue), it first needs to initiate the `sharqLock` hardware operation on the particular queue. Optionally, the enqueue and dequeue ports can be locked independently. When a certain queue is locked, the SHARQ accelerator is temporarily denied its access so that the software can safely operate on the queue. During this time, any SHARQ operation would return with the status `queue locked` to the initiating core. Finally, the queue can be unlocked via the `sharqUnlock` hardware operation.

While this subsection described the fundamental interplay between hardware and software, the next Section 4.5.3 focuses on the advanced features that enable a smart hardware-software interaction. Moreover, Section 4.5.4 explains the functionality of the hardware-accelerated operations listed in the API.

### 4.5.3 Smart Hardware-Software Interaction

The handler task's underlying idea, the ownership concept, and the dynamic memory management support have been briefly sketched in the SHARQ concept. This section expounds on them and accentuates the benefits of such hardware-software interaction.

#### 4.5.3.1 Handler Task

The idea behind the handler task is to enable decoupled processing of queued elements. Instead of putting the responsibility for invoking the handler task on the sending core, as is the case for the design alternatives and necessary without appropriate hardware support, the SHARQ accelerator takes charge. It can conditionally schedule the handler task adhering to a specific contract between hardware and software. At the queue creation time, the optional and application-specific handler task is user-defined for the particular queue. Its

```
/* Software-Defined (all local) */
sharq_t sharqCreate(int elemSize, int maxLength, int allocFill);
int sharqDestroy(sharq_t Q);
int sharqReconfigure(sharq_t Q, int maxLength, int allocFill);
int sharqRegisterHandlerTask(sharq_t Q, task_t *handlerTask, int maxHandlers);
int sharqRegisterMemoryTask(sharq_t Q, task_t *memoryTask, int minFree, int maxFree,
     int timeDelay, int accessDelay);

/* Hardware-Managed */
int sharqLock(sharq_t Q, bool enqueue, bool dequeue); // local
int sharqUnlock(sharq_t Q, bool enqueue, bool dequeue); // local

int sharqEnqueue(sharq_t Q, void *element); // local or remote
int sharqEnqueueAsync(sharq_t Q, void *element, task_t *completionTask); // remote
int sharqEnqueuePtr(sharq_t Q, void *element); // local
int sharqDequeue(sharq_t Q, void *buffer, bool fromHandler=false); // local
int sharqDequeueAsync(sharq_t Q, void *buffer, task_t *completionTask); // remote
void *sharqDequeuePtr(sharq_t Q, bool fromHandler=false); // local

int sharqAllocatorPush(sharq_t Q, void *buffer); // local
void *sharqAllocatorPop(sharq_t Q); // local
```

**Figure 4.4: Software API of SHARQ.** The list contains functions for queue creation/configuration via software and hardware-accelerated operations handled by the SHARQ accelerator.

task descriptor and its maximum concurrency (i.e., how many instances of the task may run in parallel) are registered in the queue descriptor via the `sharqRegisterHandlerTask` API call. Note that each queue may have a different handler task and concurrency level.

During queue operations, the hardware-software interaction works as follows. The SHARQ accelerator keeps track of the amount of actively running handlers. Upon a successful enqueue operation, an instance of the handler task is invoked (or so to speak, woken up) by the SHARQ accelerator if and only if the number of actively running handler tasks is less than the user-defined maximum concurrency level. To schedule the handler task, SHARQ solely retrieves its task descriptor from the queue metadata section and inserts it into the hardware scheduler [1] via the bus. The core on the queue's tile, which executes the handler task, adheres to its part of the protocol. Whenever the handler task encounters an empty queue (i.e., a dequeue operation returns no element), it will terminate. The hardware is informed that the current operation was performed by a handler task in that a special bit is set in the dequeue command (`bool fromHandler=true`). Consequentially, the SHARQ accelerator decrements the count of actively running handlers in the respective queue descriptor.

In summary, an instance of the handler task goes to sleep when encountering an empty queue and is woken up by hardware once the queue is not empty anymore and if less than the maximum number of active handlers are running. This hardware-monitored mechanism relieves the cores of this duty and enables a decoupled processing between enqueueing and dequeuing cores.

[1] Although the prototype platform uses a hardware scheduler [165] which enables interrupt-less scheduling, it is not a conceptual requirement. Nevertheless, it enables seamless hardware-software interaction.

#### 4.5.3.2 Ownership Concept

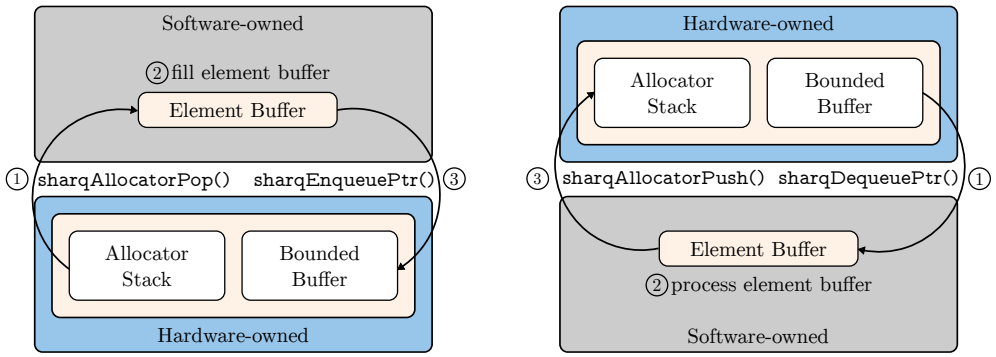
As mentioned earlier, the ownership concept is required to enable the proposed dynamic memory management of SHARQ as well as the pointer-only sharqEnqueuePtr and sharqDequeuePtr operations.

**Motivation.** Since the SHARQ queue operation and memory management are fully hardware-controlled, the queue's allocator stack and bounded buffer in the queue descriptor can only be accessed by the SHARQ accelerator. For regular queue operations including data transfer (i.e., sharqEnqueue, sharqEnqeueAsync, sharqDequeue, and sharqDequeueAsync), the access to the allocator stack and bounded buffer are coupled. On every regular enqueue operation, the software interacts with the SHARQ accelerator as follows: (1) The *source buffer* is created and filled by the user. Either it already existed, or it has to be allocated by software. (2) During the sharqEnqueue operation, the SHARQ accelerator pops a pre-allocated *element buffer* from the queue's allocator stack, copies the payload from the *source buffer* into the *element buffer*, and inserts the reference to the *element buffer* into the bounded buffer. (3) Upon the completed sharqEnqueue operation, the *source buffer* is unused and may be freed by software. Likewise, for dequeue operations, the SHARQ accelerator gets an *element buffer* reference from the bounded buffer, copies the payload from it into the *destination buffer*, and hands back the *element buffer* to the allocator stack.

In both cases, the *element buffers* are fully hardware-owned while the *source* and *destination buffers* are fully software-owned. So far, this hinders pointer-only sharqEnqueuePtr and sharqDequeuePtr operations since they do not require the separation of the buffers.

**Concept.** To circumvent the problem, the ownership cycle of the *element buffer* is introduced by leveraging the hardware-managed sharqAllocatorPush and sharqAllocatorPush operations in conjunction with the pointer-only sharqEnqueuePtr and sharqDequeuePtr operations. The smart hardware-software interaction, available for local queue operations, is depicted in Figure 4.5.

A local enqueue operation works as follows: (1) The software obtains a hardware-owned *element buffer* via the sharqAllocatorPop operation instead of allocating a new buffer (only if the queue's allocator stack is empty, a new buffer needs to be allocated by software).




**(a)** Ownership Cycle for Enqueue. **(b)** Ownership Cycle for Dequeue.

**Figure 4.5: Ownership concept.** Illustration of the hardware-software interaction of the ownership concept. These purely local operations do not incorporate data transfers but solely transfer the element buffer ownership between hardware and software.

(2) The software fills the temporarily software-owned *element buffer*. (3) The *element buffer* is inserted into the queue via the `sharqEnqueuePtr` operation. This also transfers the ownership of the *element buffer* back to hardware.

Analogously, for a local dequeue operation: (1) The software dequeues the pointer of the hardware-owned *element buffer* from the queue via the `sharqDequeuePtr` operation. (2) The software processes the temporarily software-owned *element buffer*. (3) The *element buffer* is pushed back onto the queue's allocator stack via the `sharqAllocatorPush` operation. This also transfers the ownership of the *element buffer* back to hardware. Only if the queue's allocator stack is full the software needs to free the buffer itself.

The benefit of solely transferring the ownership of element buffers is that the source and destination buffers become obsolete for local operations. Thus, potential software memory allocations, de-allocations, and superfluous data transfers can be saved. However, the pointer-only operations lead to a varying amount of hardware-owned element buffers. Although the local pointer-only operations harmonize with the regular enqueue and dequeue operations, the user needs to be aware of it. It may lead to an empty bounded buffer and an empty allocator stack at once, preventing regular queue operations from succeeding.

**Summary.** The proposed ownership concept enables pointer-only queue operations. Moreover, when encountering a steady-state of the queue's fill level, costly software allocations of the source and destination buffers can be avoided for local queue operations. Last but not least, the `sharqAllocatorPush` and `sharqAllocatorPop` operations also form the basis for the dynamic memory management support.

#### 4.5.3.3 Dynamic Memory Management Support

The dynamic memory management of `SHARQ` enables adaptability to run-time requirements changes in the queue's memory demands.

**Motivation.** With the introduction of the pointer-only queue operations and the ownership concept, `SHARQ` no longer has the exclusive ownership of the queue's element buffers. The ownership of individual buffers can be transferred between hardware and software. This is unproblematic as long as the queue is operated in a steady-state with a safe fill level. However, it cannot be assured that the user neither overloads the queue with buffers nor forgets to return buffers to the queue fast enough. Thus, it is desirable to have a management procedure that enforces enough but not too many hardware-owned element buffers. Such a mechanism comprises two parts: monitoring the queue state and adding or removing hardware-owned buffers. Theoretically, the application itself could take responsibility for the monitoring and enforcement. However, this would create undesired software overhead and have a performance-degrading impact on queue operations. It is desirable to decouple queue operations from taking care of the queue's memory management.

**Concept.** Therefore, the `SHARQ` accelerator monitors the queue state at run-time, i.e., the available number of element buffers on the allocator stack is checked. Upon user-specified conditions, `SHARQ` enforces the dynamic memory management by scheduling the memory management task. To do so, `SHARQ` retrieves the memory management task descriptor from the queue descriptor and, similar to the handler task invocation, inserts the task descriptor into the hardware scheduler. The memory management task and the enforcement conditions are deposited in the respective queue descriptor at the time of queue creation.

The conditions comprise the software-defined thresholds (`min/max # free elem`) so that the memory management task can already be scheduled before the queue runs out of free buffers or waists too many unused ones. Once these thresholds are hit, `SHARQ` initiates the dynamic memory management. Optionally, the user-defined `time until` or `accesses until` values can be used to delay the enforcement until the fill level is beyond a threshold either for a given amount of time or accesses, respectively. This may prevent too frequent scheduling of the memory task if the fill level fluctuates around one of the thresholds. Moreover, `SHARQ` only schedules one memory management task per queue at a time. It sets a flag in the queue descriptor, which the software unsets once the task terminates.

The software is only involved in this hardware-controlled mechanism when the memory management task gets triggered. When executed, the memory management task first analyzes the current queue state as it might have altered since the memory management task was triggered by `SHARQ`. The software fetches the required information from the queue descriptor read-only. Suppose the decision to allocate additional or free unused element buffers stands. In that case, the software triggers the `SHARQ` accelerator via the `sharqAllocatorPush/Pop` operations to insert or retrieve the element buffers to or from the queue's allocator stack, respectively. Thus, the memory management task can be executed concurrently with normal queue operations, requiring no queue locking.

**Summary.** The dynamic memory management support of `SHARQ` allows adapting the queue size to the actual need of the application. Varying requirements are monitored and enforced by the `SHARQ` accelerator, and, if necessary, a user-defined memory management task is scheduled. Thus, `SHARQ` queues are flexible, resource-aware, and adaptive.

### 4.5.4 Hardware-Accelerated Queue Management

The previous sections dealt with the software-defined queue creation, the hardware-software interplay, and the advanced hardware-software interaction, including the handler task, the ownership concept, and the dynamic memory management. This section presents the functionality of the `SHARQ` accelerator and its interaction with other hardware components.

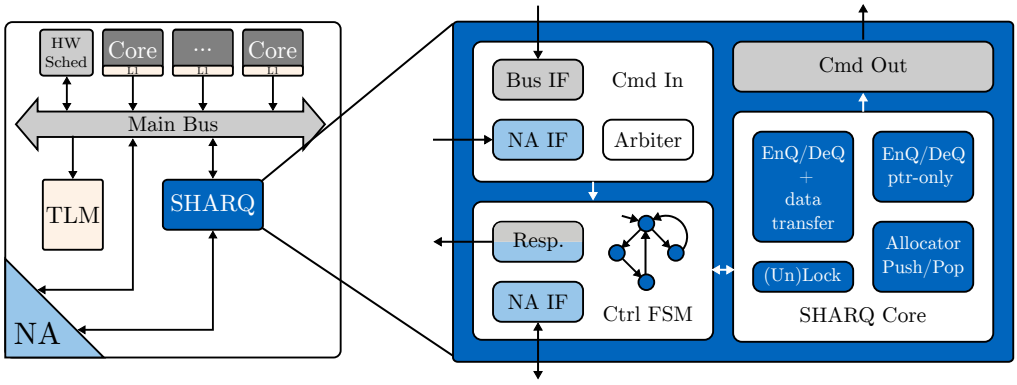



**Figure 4.6: Integration and architecture of the `SHARQ` accelerator.** Overview of a compute tile including the `SHARQ` accelerator (left) and block diagram of the internal structure of `SHARQ` (right). `SHARQ` receives commands either from a local core or remotely via the NA. The queue operations are outsourced to the `SHARQ` accelerator near the memory where the queue resides. `SHARQ` cooperates closely with the NA to initiate and receive remote data transfers and with the hardware scheduler to invoke the handler and memory management tasks.

#### 4.5.4.1 Architecture Overview

The position of the SHARQ accelerator within a tile is depicted on the left side of Figure 4.6 while its internal structure is illustrated on the right side. SHARQ is tightly connected to the network adapter (NA). Thus, it can receive incoming requests from other tiles, buffered in the NA's existing FIFOs, and receive or initiate payload transfers from and to remote memories. Moreover, the accelerator is connected to the local bus via its bus interface. The CMD In interface is used to receive incoming requests from local cores (Bus IF), which are arbitrated with the incoming remote requests from the NA IF. The CMD In module decodes the local and remote requests and forwards them to the SHARQ core module. Furthermore, the accelerator contains a CMD Out interface to interact with the hardware scheduler, and issue memory reads and writes to the memory controller. The modular design as a bus unit provides independence of a particular memory controller protocol implementation. Thus, it is compatible with the tile-local SRAM and the global DDR-SDRAM memory controller.

#### 4.5.4.2 Queue and Memory Management

All performance-degrading up-calls to the software are avoided during regular operation to ensure efficient queue operations. The queue's memory management is entirely handled by the SHARQ accelerator with the help of the allocator stack present in the queue descriptor. It is not obvious that the queue's memory management via the allocator stack needs to be decoupled from the bounded buffer. If the entire queue operation, including the data transfer, would be atomic (i.e., not interleaved by other queue operations), it would be possible to implement the queue with the bounded buffer alone. However, on a packet-based NoC architecture [178], a queue operation can comprise several packets that may as well be interleaved with packets of other (queue) operations. This requires that the SHARQ accelerator can process requests on packet granularity, leading to potentially multiple ongoing operations. To ensure that each queue remains consistent, the decoupling of the bounded buffer and the allocator stack becomes a requirement. For example, when processing a multi-packet remote enqueue operation, a free element buffer is popped from the allocator stack when the first packet of the operation arrives. Then the payload, which resides in the remaining packets, is received. Note that all packets may be interleaved by packets of other (queue) operations. Only upon the completion of the entire data transfers (i.e., once the last packet of this enqueue operation arrived) SHARQ is allowed to insert the element into the bounded buffer, becoming visible for dequeuers.

Furthermore, this decoupling allows asynchronous enqueue and dequeue operations.

#### 4.5.4.3 Operations and Data Transfer

This section explains the functionality of the hardware-accelerated SHARQ operation:

- remote sharqEnqueue with synchronous data transfer,
- remote sharqEnqueueAsync with asynchronous data transfer,
- local sharqEnqueue and sharqDequeue with data transfer,
- local pointer-only sharqEnqueuePtr and sharqDequeuePtr,
- remote sharqDequeueAsync solely with asynchronous data transfer, as well as
- local sharqAllocatorPush and sharqAllocatorPop operations.

**Remote Enqueue with Synchronous Data Transfer.** The `sharqEnqueue` operation combines the local and remote enqueue with synchronous data transfer. `SHARQ` is able to differentiate them based on the address of the queue handle. As the name suggests, the data transfer of the element payload occurs synchronously to the enqueue request. Through tight interaction with the network adapter's remote DMA engine, the synchronous remote enqueue basically is a DMA transfer with a special header flit containing the queue handle instead of the destination address. Upon reception on the remote tile, the network adapter forwards the request to the `SHARQ` accelerator, which executes the steps illustrated in Figure 4.7:[2] (1b) A remote core initiates the `sharqEnqueue(queue_handle)` opera-

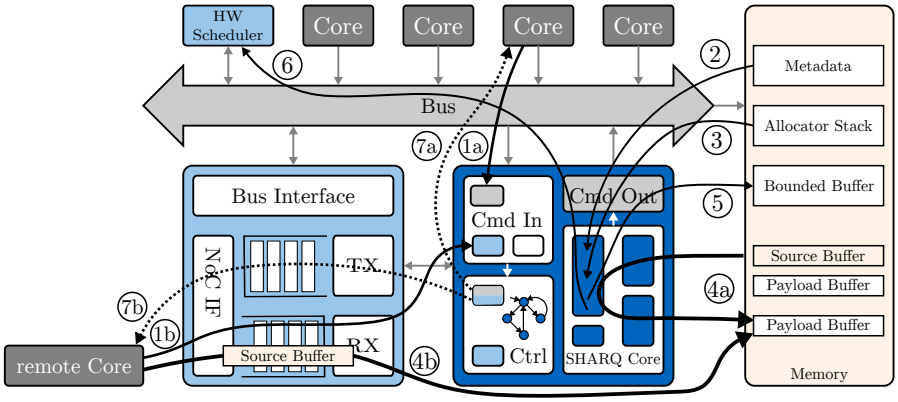



**Figure 4.7: Enqueue operation sequence.** Steps 1a, 4a, and 7a are performed in case of a local enqueue; steps 1b, 4b, and 7b are performed in case of a synchronous remote enqueue; asynchronous remote enqueue is not shown.

tion, (2) `SHARQ` decodes the request and retrieves the queue metadata from the respective queue descriptor corresponding to the `queue_handle`. `SHARQ` analyzes the queue state, e.g., whether the queue is full. If it is full, the remaining packets of the incoming payload transfer via DMA must be dropped. In this case, steps 3 to 6 are skipped, and an error response code is generated. Otherwise, (3) the `SHARQ` accelerator accesses the queue's allocator stack in memory and pops a pointer to a free element buffer from it. This pointer to the element buffer is communicated to the DMA engine as the destination address of the payload transfer, i.e., where the remaining packets of the DMA should be stored. (4b) the DMA engine, which paused until steps 2 and 3 were completed, receives the element payload into the element buffer. (5) Once the data transfer is completed, `SHARQ` inserts the element reference into the queue's bounded buffer. (6) If the condition to schedule a handler task is fulfilled (i.e., the queue was empty, and the number of actively running handler tasks is below its maximum concurrency) `SHARQ` inserts the handler task descriptor, which is retrieved from the queue metadata, into the hardware scheduler via the bus, (7b) Finally, `SHARQ` sends the response code to the initiating core. The benefit of the synchronous data transfer is the minimized latency in case the operation succeeds. However, if the queue is full, the payload must be dropped upon reception. If this happens frequently, it might lead to performance degradation. Furthermore, the initiating core is blocked until the entire operation, including synchronous data transfer, is completed. Especially for larger payload sizes, this may be too long. Thus, the remote enqueue also exists in an asynchronous variant.

[2] Parts of this figure also apply to the asynchronous remote and the local enqueue with data transfer.

**Remote Enqueue with Asynchronous Data Transfer.** The `sharqEnqueueAsync` operation still handles the data transfer in hardware. However, the payload is not directly sent with the enqueue requests but initiated by `SHARQ` in a later stage. In general, most of the steps depicted in Figure 4.7 apply. Nevertheless, the initiating core receives the response code directly at the end of step 2 (before the asynchronous transfer) instead of step 7 (after the synchronous transfer). Thus, the initiating core is informed if the operation failed due to a full queue or will be successful at this early stage. Thereby, latency hiding is enabled, and the core may resume processing earlier. Hence, this variant does not require payload dropping if the queue is full. However, if the analysis of the queue state in step 2 is successful, `SHARQ` proceeds with steps 3 to 6. Instead of receiving the payload in step 4b directly, `SHARQ` initiates a pull-DMA request of the element buffer from the initiating tile into the queue. The initiating core, which has already resumed other work, is notified that the asynchronous data transfer is completed. Therefore, a user-defined completion task is scheduled by `SHARQ` on the initiating core via remote task invocation into the hardware scheduler on the initiating tile. This task should not be confused with the handler task conditionally scheduled on the queue's tile in step 6. The task descriptor of the specific completion task is passed to `SHARQ` together with each `sharqEnqueueAsync` request.

**Local Enqueue with Data Transfer.** The `sharqEnqueue` also exists in a local variant with data transfer to allow for symmetric behavior of both local and remote operations with data transfers. The same steps as for the synchronous remote enqueue apply with the following slight modifications in steps 1, 4, and 7. They are also illustrated in Figure 4.7 as step 1a, 4a, and 7a. A local core triggers the operation (1a) and receives the response code (7a). Moreover, the data transfer in step 4a is outsourced from `SHARQ` to the local-DMA engine.[3] The pointer-only `sharqEnqueuePtr` operation will be described later.

**Local Dequeue with Data Transfer.** The local `sharqDequeue` operation works complementary to the local enqueue operation. The necessary steps are illustrated in Figure 4.8. The figure contains the sequence of steps for both the local dequeue (variant a) and the remote dequeue (variant b). (1a) A local core initiates the `sharqDequeue(queue_handle)` operation, (2) `SHARQ` decodes the request, and retrieves the queue metadata from the respective queue descriptor corresponding to the `queue_handle`. `SHARQ` analyzes the queue state, e.g., whether the queue is empty. If it is empty, steps 3 to 6 are skipped and an error response code is generated. Otherwise, (3) `SHARQ` dequeues the next element pointer from the bounded buffer, (4) *only performed for remote dequeues* (5a) `SHARQ` copies the element buffer to the destination buffer, which was passed to `SHARQ` together with the dequeue request, (6) After the data transfer, `SHARQ` pushes the pointer of the now unused element buffer back onto the allocator stack. (7a) `SHARQ` signals the completion of the operation to the initiating core by sending the response code.

**Remote Dequeue with Asynchronous Data Transfer.** The remote dequeue only exists in an asynchronous variant as it is obviously not possible to send the payload together with the requests. For the `sharqDequeueAsync`, analogous to the asynchronous remote enqueue operation, `SHARQ` initiates the remote data transfer only after checking the queue state. The necessary steps are basically the same as for the local dequeue operation. They are also depicted in Figure 4.8, and differences are marked as variant b. In step 1b, the request is

[3] The local-DMA engine is hanging on the bus. It is not depicted due to abstraction.

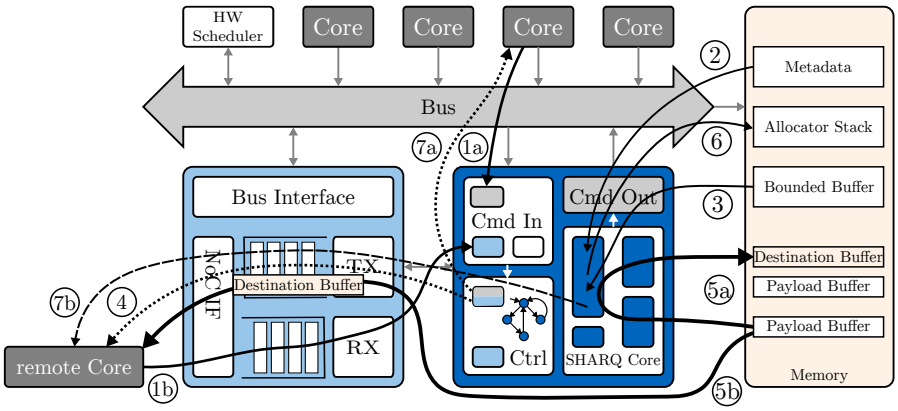
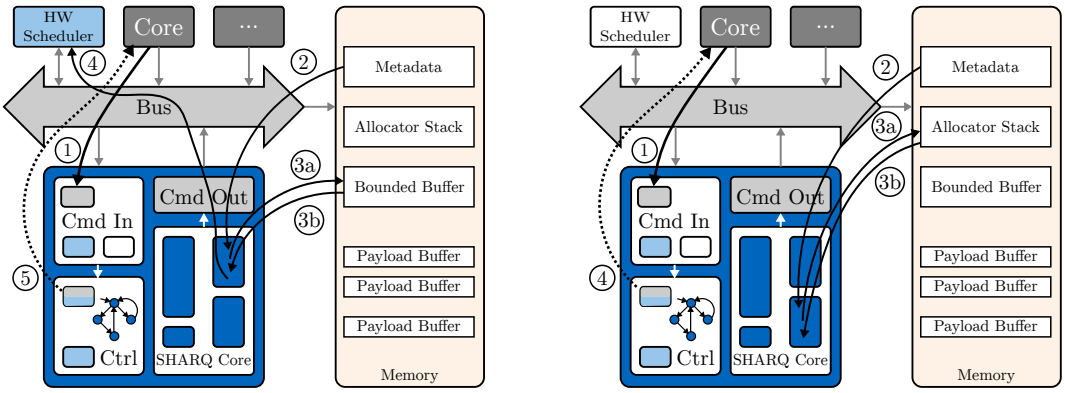



**Figure 4.8: Dequeue operation sequence.** Steps 1a, 5a, and 7a are performed in case of a local dequeue; steps 1b, 5b, and 7b are performed in case of remote dequeue which is always asynchronous; step 4 is only performed for remote dequeues.

received via the NoC from a remote core instead of a local core. Step 4, which does not exist for the local variant, sends the response to the initiating core instead of step 7a. Moreover, in step 5b, `SHARQ` initiates a remote DMA of the element buffer into the destination buffer in the memory of the initiating tile. Finally, step 7b schedules the completion task, already known of the asynchronous remote enqueue, to a core on the initiating tile.

**Local Pointer-Only Enqueue and Dequeue.** The pointer-only `sharqEnqueuePtr` and `sharqDequeuePtr` operations avoid the data transfer inside the same physical memory. As explained earlier, they solely transfer the ownership of the element buffer from software to hardware for an enqueue and vice versa for a dequeue operation. Thus, no interaction with the queue's allocator stack is required. Solely the bounded buffer is involved, and the number of hardware-owned buffers is increased or decreased, respectively. Figure 4.9a illustrates the necessary steps performed by the `SHARQ` accelerator: (1) `SHARQ` is triggered by a local core, (2) `SHARQ` decodes the request, reads the queue metadata from the respective queue descriptor corresponding to the `queue_handle`, and analyzes the queue state. (3) For an enqueue operation (3a), `SHARQ` enqueues the pointer to the element buffer into the bounded buffer if the queue is not full. For a dequeue (3b), and if the queue is not empty, `SHARQ` dequeues the pointer to the next element buffer from the bounded buffer. (4) In case of an enqueue, and if the condition is fulfilled, `SHARQ` schedules the handler task via the hardware scheduler. (5) Finally, on a dequeue, `SHARQ` sends back the element buffer pointer to the initiating core, or simply the response code for an enqueue.

**Local Allocator Push and Pop.** Finally, the `sharqAllocatorPush` and `sharqAllocatorPop` operations are the remaining complementary building blocks of the ownership concept. They do not touch the queue's bounded buffer but only interact with the allocator stack. Figure 4.9b illustrates the necessary steps performed by the `SHARQ` accelerator: (1) `SHARQ` is triggered by a local core, (2) `SHARQ` decodes the request, reads the queue metadata from the respective queue descriptor corresponding to the `queue_handle`, and analyzes the queue state. (3) For an allocator push operation (3a), `SHARQ` pushes the pointer onto the allocator stack if it is not full. For an allocator pop operation (3b), and if it is not empty, `SHARQ` retrieves the pointer of the next free buffer slot from the allocator stack. (4) `SHARQ` sends back the response code or the buffer pointer to the initiating core, respectively.




**(a)** Pointer-only enqueue/dequeue operation.

**(b)** Allocator push/pop operation.

**Figure 4.9: `SHARQ` operations for the ownership concept.** Operation sequence of the pointer-only enqueue/dequeue and allocator push/pop operations. These operations only exist locally.

### 4.5.5 Software-Managed SHARQ Emulation

The previous subsections expounded the hardware-software co-design approach of `SHARQ`. This section describes a software-managed queue approach that supports the same features as `SHARQ` to evaluate `SHARQ` against such an approach. The API of the software-managed alternative is identical to `SHARQ`'s API so that the programmer does not see a difference in the application or library code. In the operating system, the `SHARQ` operations are then either forwarded to the `SHARQ` accelerator via the OS hardware driver code, or the operations are emulated using the procedure explained in this section. The queue is defined by the same queue descriptor. However, all queue operations are software-managed, and atomic access is enforced via lock-based synchronization. Local queue operations are straightforward and can directly be performed by the initiating core. If they include payload copies larger than 32 Bytes, they may benefit from the existing local-DMA engine.

As indicated during the analysis of the design alternatives in Section 4.4.1, the software-managed remote queue operations are more complicated than the hardware-managed ones. Although the software-emulated `SHARQ` variant has an advantage over the design alternative, still a queuing task has to be scheduled on the remote tile, which performs the queue operation on behalf of the initiating core. Nevertheless, the remote allocation is avoided as the element buffers are pre-allocated in the queue descriptor's allocator stack.

In summary, the following steps are necessary for, e.g., the remote enqueue with synchronous data transfer: (1) The initiating core calls the `sharqEnqueue` operation, which schedules the enqueuing task on the remote tile. (2) The task is scheduled on a core on the remote tile, decoding the request and retrieving the queue state from the queue descriptor. (3) A free buffer slot is popped from the allocator stack. (4) The core initiates a pull-DMA request to fetch the payload from the source buffer on the initiating tile into the free element buffer obtained from the allocator stack. (5) Upon completion of the data transfer, the core on the remote tile enqueues the element buffer into the bounded buffer. (6) If the condition is fulfilled, the handler task is scheduled on the remote tile. (7) Finally, the response code is sent back to the initiating core via remote task invocation.

The other `SHARQ` operations are emulated accordingly. Thus, a software-managed mechanism with the same features is provided as a reference for evaluation.

### 4.5.6 Application Scenario: MPI Library Integration

SHARQ is integrated into the MPI library as one of many possible application scenarios to showcase its benefit. In the already existing MPI library implementation for the prototype platform, the communication between MPI processes is performed synchronously (e.g., MPI_send() or MPI_receive()). Before the communication between local and/or remote processes can be established, the registration between sender and receiver is necessary, which requires the transmission of 68 Bytes of metadata from the sender to the receiver. A queue of outstanding registrations exists per MPI process. However, without the SHARQ concept, this registration would require a remote memory allocation, a DMA transfer, and the remote task invocation of a handler task that performs the registration.

Therefore, SHARQ is applied to enable a more efficient implementation of the MPI process registration. Each MPI process has its SHARQ queue to receive pending registration requests from other MPI processes to themselves. Thus, initiating a registration results in a simple local or remote enqueue operation to the respective SHARQ queue of the receiver. The respective handler task of each queue is responsible for dequeuing the requests and performing the registration between sender and receiver. The necessary information is retrieved from the dequeued element payload.

In summary, the mitigation of the registration overhead between communicating MPI processes is achieved by buffering the incoming requests in hardware-accelerated SHARQ queues. Even more complex MPI operations such as MPI_broadcast() profit from the optimization as they are based on MPI_send() and MPI_receive(). Thus, the user of the MPI library does not see any changes in the MPI API. Solely the internal implementation of the individual MPI functions is modified and maps the registration request to SHARQ operations as explained above. In other words, the MPI function calls are not replaced by SHARQ calls, but the particular MPI function performs SHARQ calls internally. The resume of the responsible software engineer of the operating system team of the Invasive Computing research center was: "The adjustments required to utilize [SHARQ] in our MPI library were, for the most part straight forward since using [SHARQ] queues typically decreases implementation complexity drastically in comparison to custom solutions in software." [6]

The benefit of this approach will be evaluated with the MPI-based NAS benchmarks in Section 4.6.4.

## 4.6 Evaluation

The previous section presented the concept and architecture of the proposed dynamic, software-defined hardware-accelerated SHARQ queues. This section evaluates the viability and efficiency of the hardware-software co-design approach on a tile-based many-core architecture. SHARQ is integrated into each tile of the FPGA-based prototype platform described back in Section 2.4. The evaluation methodology is defined in Section 4.6.1. The evaluation begins with the analysis of hardware cost and attainable frequency in Section 4.6.2. Then, in Section 4.6.3, several microbenchmarks are employed to analyze the efficiency of individual SHARQ operations when compared to a software-managed queue implementation. A special focus is set on remote versus local performance. Furthermore, Section 4.6.4 analyzes SHARQ with the MPI-based NAS parallel benchmarks. In particular, performance, scalability, and the benefit of SHARQ's dynamic memory management are under evaluation.

### 4.6.1 Methodology

The evaluation methodology resembles the previous Chapter 3's approach. Three of the four metrics, described in Section 3.6.1, are used again: hardware cost, attainable frequency, and benchmark execution time. Additionally, since incorporating data transfers to achieve data locality is one of the key differences between the previous and this chapter, the portion of time spent on data transfers is analyzed.

### 4.6.2 Hardware Cost and Frequency Evaluation

Table 4.1 shows the FPGA resource requirements and the attainable frequency of the `SHARQ` accelerator when synthesized onto the prototype platform consisting of four Virtex-7 2000T FPGAs. Xilinx Vivado 2018.3 was used as the synthesis tool in the default synthesis mode and the implementation strategy `area explore`. The hardware cost and frequency analysis are based upon the same synthesis run.

The first part of the table distinguishes the cost of the `SHARQ`-base variant from the additional cost of the logic required to provide the ownership concept and the dynamic memory management support. Then, in the second part of the table, the cost and frequency of the `SHARQ` accelerator for different system sizes are provided, which enables to analyze the scalability. Finally, the third part shows two reference values of a 4x4 tile design running at 50 MHz: a LEON 3 core and an entire tile.

**Hardware Cost.** The hardware cost of `SHARQ` is obtained by analyzing the FPGA resources required for the synthesized accelerator. By design, it is independent of the number of `SHARQ` queues or the software-defined queue properties like queue length or element size.

Each per-tile `SHARQ` accelerator consumes 1752 slices in the base variant. A more detailed breakdown into LUTs, Registers, MUXes, BRAMs, and DSPs is also found in the table. When including the ownership concept (i.e., pointer-only queue operations and allocator push/pop) and the dynamic memory management support, 115 and 295 slices are added, respectively. Thus, each full-fledged `SHARQ` accelerator consumes 2162 slices in the smallest possible multi-tile 2x1 design. In a 4x4 design, it requires 2116 slices. This decrease in resources can be explained by potentially fewer optimizations in this synthesis run, as also a lower frequency is attained.

**Table 4.1: Resource utilization and attainable frequency of `SHARQ`.** Synthesized on a Virtex-7 2000T FPGA where one slice contains 4 LUTs, 8 Registers, and 2 Muxes. Each per-tile `SHARQ` accelerator requires approximately 16.9% of the slices of the five LEON3 cores per tile.

| HW Module | Slices | LUT | Register | Mux | BRAM | DSP | f [MHz] |
|---|---|---|---|---|---|---|---|
| `SHARQ`-base | 1 752 | 5 170 | 2 379 | 12 | 0 | 0 | 218 |
| + ownership | 115 | 413 | 208 | 22 | 0 | 0 | 215 |
| + mem-task | 295 | 784 | 409 | 0 | 0 | 0 | 218 |
| `SHARQ` 2x1 | 2 162 | 6 367 | 2 996 | 34 | 0 | 0 | 218 |
| `SHARQ` 2x2 | 2 142 | 6 264 | 2 998 | 7 | 0 | 0 | 203 |
| `SHARQ` 4x4 | 2 116 | 6 261 | 3 021 | 5 | 0 | 0 | 209 |
| `SHARQ` 32x32 | 2 107 | 6 120 | 3 001 | 21 | 7 | 0 | 206 |
| 1 `SHARQ` 4x4 | 2 116 | 6 261 | 3 021 | 5 | 0 | 0 | 50 |
| 1 Tile in 4x4 | 33 524 | 107 267 | 53 653 | 1 095 | 249.5 | 20 | 50 |
| 5 LEON3 cores | 12 495 | 40 800 | 12 935 | 165 | 90 | 20 | 50 |

When comparing the different design sizes, it can be observed that only a small overhead is introduced for larger system sizes. The SHARQ accelerator requires 121 Bits of internal metadata per remote tile to manage concurrent queue operations from several source tiles. Thus, for a 32x32 tile design (i.e., 1024 tiles with a total of 5120 cores), the cost of SHARQ only grows by seven FPGA block rams (7x 36 kBit BRAM) to maintain the internal metadata. However, the logic itself remains roughly constant. Hence, SHARQ is scalable to large systems and an arbitrary number and layout of queues.

When comparing the SHARQ accelerator to the other components existing in the 4x4 design, it can be observed that it consumes roughly 6.4% of each tile's slices. The resource requirements are lower than for a single LEON3 core. Compared to the five LEON3 cores existing per tile, each per-tile SHARQ accelerator requires approximately 16.9% of the resources (even without BRAMs and DSPs). In total, the 16 SHARQ accelerators of a 4x4 tile design accumulate to roughly 5.8% of the required resources of the entire 16 tile design.

**Frequency Evaluation.** Table 4.1 further contains the timing analysis results. The standalone SHARQ accelerator could be operated at roughly 203-218 MHz, depending on the synthesis run. The advanced features (i.e., the ownership concept and the dynamic memory management) marginally influence the results. However, since SHARQ is integrated into the whole many-core architecture, including CPUs, Buses, the NoC, and other modules, the accelerator is operated at the same 50 MHz frequency as the residual design due to bottlenecks in other components.

### 4.6.3 Microbenchmarks

As a first step of the performance evaluation, three microbenchmarks are employed to compare SHARQ against a software-managed queue approach (SMQ). The first microbenchmark investigates the duration of the individual queue operations, both local and remote. In contrast, the second benchmark analyzes the approach's scalability when the queue is experiencing more concurrent accesses. Finally, the third benchmark highlights the benefit of pointer-only queue operations in a stress test scenario. All three microbenchmarks use the operating system and are executed on the prototype platform.

**Naming Convention of Queue Operations.** Table 4.2 defines the naming convention used in this section. The shorter names allow for a better in-text referencing. Note that for the asynchronous remote enqueue `sharqEnqueueAsync` and dequeue `sharqDequeueAsync` operations, two time measurements are differentiated. The payload-independent time until the operation returns (`r-*Q-AR`) and unblocks the initiating CPU indicating whether the operation will be successful. Moreover, the time until the entire operation, including the asynchronous payload transfer, is finished and notified by the completion task (`r-*Q-AC`).

#### 4.6.3.1 Benchmark 1: Individual Operation Performance

The goal of the first microbenchmark is to evaluate the efficiency of the individual SHARQ operations. Moreover, it analyzes the portion contributed by the payload data transfer compared to the actual queue operation management. Therefore, the duration of the SHARQ operations experienced by the initiating CPU core is measured and compared for element payload sizes of 4 Byte (pointer-only) and 2048 Byte. The benchmark uses one local or remote core, respectively, and avoids any contention on the queue.

**Table 4.2: Naming convention and abbreviations of SHARQ operations.** The shorter names allow for a better in-text referencing.

| operation | location | data transfer | API call | abbreviation |
|---|---|---|---|---|
| enqueue | local | ✓ | `sharqEnqueue()` | `l-EQ` |
| dequeue | local | ✓ | `sharqDequeue()` | `l-DQ` |
| pointer-only enqueue | local | ✗ | `sharqEnqueuePtr()` | `l-EQ-P` |
| pointer-only dequeue | local | ✗ | `sharqDequeuePtr()` | `l-DQ-P` |
| allocator push | local | ✗ | `sharqAllocatorPush()` | `l-Apush` |
| allocator pop | local | ✗ | `sharqAllocatorPop()` | `l-Apop` |
| sync. enqueue | remote | ✓sync | `sharqEnqueue()` | `r-EQ-S` |
| async. enqueue return | remote | ✗ | `sharqEnqueueAsync()` | `r-EQ-AR` |
| complete | remote | ✓async | | `r-EQ-AC` |
| async. dequeue return | remote | ✗ | `sharqDequeueAsync()` | `r-DQ-AR` |
| complete | remote | ✓async | | `r-DQ-AC` |

**Benchmark Setup.** Without loss of generality, the queue resides in the tile-local memory of `tile 0`. Thus, the local operations are executed by a core on `tile 0` while the remote operations are performed by a core on `tile 1`. Each SHARQ operation is performed 512 times without concurrency on the queue to obtain reliable results The execution time is determined in the following way. First, timestamps are taken before and after the individual operations' execution, and their difference is calculated. Moreover, the measurement error introduced by reading the timestamp is corrected by determining and subtracting the average overhead that the reading of the time stamp takes. Finally, the execution time's mean value and standard deviation are calculated over the 512 individual measurements.

**Results.** The resulting execution times in clock cycles are depicted in Figure 4.10 as a stacked bar plot diagram for the different SHARQ operations when performed by SHARQ compared to the software-managed queue SMQ. The respective left bar of each tuple corresponds to SHARQ while the respective right bar belongs to the SMQ. Moreover, each stacked bar is vertically split into the duration of the actual queue operation management ( for SHARQ, and for SMQ), and the respective payload transfer of 2048 Byte ( for SHARQ, and for SMQ). The stacked bars are generated as follows. Each operation is performed with a pointer-only 4 Byte and a 2048 Byte payload. The execution time of the 4 Byte operation denotes the queue operation management overhead, while the difference between both measurements represents the data transfer overhead. Five key observations can be derived from Figure 4.10:

1) While SHARQ already creates a benefit for local operations compared to the SMQ, its remote performance is significantly better. For local operations the overhead of the SMQ's queue management without the actual payload transfer is approximately 32% (`l-DQ`) - 62% (`l-EQ`) higher than for SHARQ . However, remotely, this number is much higher, e.g., a factor of 10.0x and 2.85x for the synchronous (`r-EQ-S`) and asynchronous remote enqueue operation (`r-EQ-AC`), respectively.

2) When considering queue operations with 2048 Byte payload, the same additional duration for the data transfer occurs for the respective SHARQ and SMQ variants. For local queue operations, the data transfer, if existing, dominates the total duration while the necessary queue management becomes a minor part. Also for the syn-

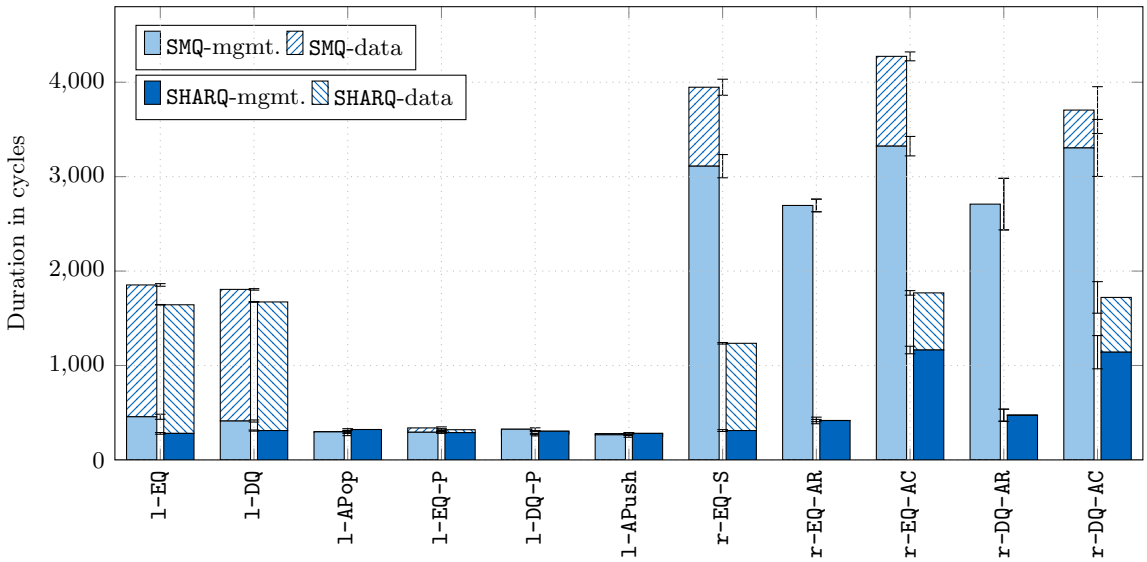



**Figure 4.10: Analysis of individual SHARQ operations.** Duration of individual operations broken down into SHARQ management and data transfer of 2048 Byte. Abbreviations are listed in Table 4.2. SHARQ is especially beneficial for remote operations since its remote management overhead is reduced to a minimum and is almost as small as for local SHARQ operations.

chronous remote enqueue operations `r-EQ-S`, the payload transfer of SHARQ takes longer than the actual queue management. The asynchronous remote SHARQ operations `r-*Q-A*` have a higher management overhead due to the additional scheduling of the completion task to notify the completion. Nevertheless, all remote SMQ operations have a significantly higher overhead originating from the software queue management compared to the payload transfer.

3) Interestingly, the remote enqueue with synchronous data transfers is even faster than the local enqueue. This is explained by the twice as high effective memory bandwidth for remote DMAs since the source and destination buffers are in different memories and can be streamed instead of copied by the local DMA engine with alternating read and write bursts. Furthermore, the queue management of SHARQ of the remote enqueue with 338 cycles takes solely 30 cycles longer than for the local enqueue.

4) The pointer-only operations belonging to the ownership concept (`l-EQ-P`, `l-DQ-P`, `l-Apush`, and `l-Apop`) have a comparable performance in hardware and software since they are purely local operations. Except for the `l-Apop` operation (-8%), SHARQ is between 4% (`l-DE-P`) and 12% (`l-Apush`) faster than the SMQ variant.

5) The standard deviation of all SHARQ operations, except the ownership operations, is significantly lower than for the SMQ.

**Summary.** In conclusion, this first microbenchmark revealed that SHARQ is especially beneficial for remote queue operations. The management overhead of remote queue operations is reduced significantly compared to the SMQ. For the synchronous remote enqueue, the management overhead is even comparable to local queue operations.

#### 4.6.3.2 Benchmark 2: Scalability Analysis

While the first benchmark analyzed individual operations' performance, the goal of the second microbenchmark is to evaluate the scalability of `SHARQ` when the queue experiences higher levels of concurrency and contention. Thus, cores from a varying amount of tiles perform concurrent queue operations on a single queue. Various runs of the benchmark are executed for different queue element sizes.

**Benchmark Setup.** Without loss of generality, the queue resides in the tile-local memory of `tile 0`. The core count per tile is four, and the number of involved remote tiles is varied between one and fifteen. Thus, four up to 60 remote cores operate concurrently on the queue. Each involved remote core performs 1 000 remote enqueue operations with synchronous data transfer (`sharqEnqueue`, `r-EQ-S`) on the queue. Concurrently, all four local cores on `tile 0` run the handler task (i.e., the maximum concurrency of the handler task is set to four), which is defined to dequeue all elements via the `sharqDequeuePtr` operation, discard them and push the element buffer back on the allocator stack via `sharqAllocatorPush`. No other tasks are running on `tile 0`. Thus, this second microbenchmark performs remote enqueue operations including data transfer to `tile 0`, and local pointer-only dequeue operations on `tile 0`.

For each number of involved remote tiles (i.e., one to fifteen) and each queue element size, the execution time is measured by taking timestamps at the beginning and end of the entire microbenchmark. The execution time in microseconds is normalized to the total number of remote enqueue operations, i.e., divided by 1 000 and by the number of involved cores, to obtain the average duration per enqueue operation.

**Results.** The resulting normalized execution times in microseconds are depicted in Figure 4.11 for different element sizes and a varying amount of concurrent enqueuers. Several insights can be obtained from Figure 4.11:

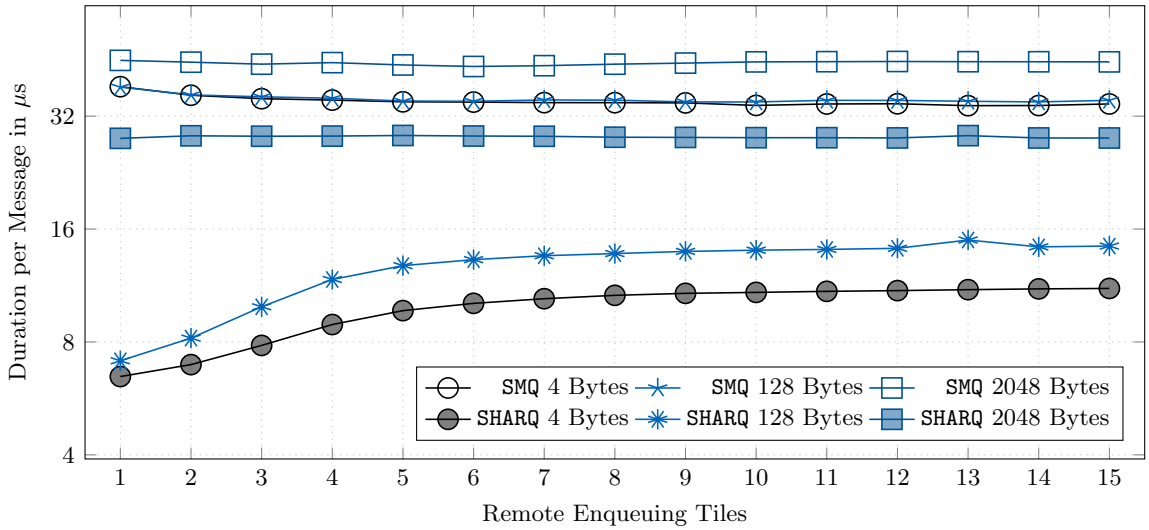



**Figure 4.11: Scalability and concurrency analysis of `SHARQ`.** Comparison of the average duration of concurrent remote enqueue operations (`SHARQ` vs. `SMQ`) for a varying amount of remote tiles and for different queue element sizes. `SHARQ` unfolds its full potential if the queue operations are not dominated by the payload transfer.

1) SHARQ outperforms the SMQ approach, especially for small element sizes and few enqueuing tiles. For an element size of 4 Byte a speedup of up to 5.92x is achieved for only one enqueuing tile. For eight enqueuing tiles, the performance of SHARQ is still roughly 3.1x faster than for the SMQ. Similarly, for the 128 Byte variants, the speedup is 5.36x and 2.45x, respectively. Thus, having eight times as many concurrent enqueuers only leads to a performance drop of SHARQ by 1.9x (4 Byte) and 2.19x (128 Byte), respectively. This is explained by the serialized execution of the queue management in the SHARQ accelerator and the increased number of concurrent dequeues by the handler task on tile 0 to avoid the queue from running full.
2) Between approximately 8 and 15 enqueuing tiles, the speedup of the 4 Byte and 128 Byte variants saturates to roughly 3.1x and 2.45x, respectively. In contrast, for the 2048 Byte variant, the saturation occurs from the beginning with a benefit of 1.61x. Two factors explain the saturation. First, when incorporating larger element sizes, the actual data transfer dominates, and the benefit of SHARQ is diminished by the increasing portion of the data transfer. Second, the remote enqueue operations are slowed down by the concurrent local dequeue operations since both operate on the same queue in the same memory. Thus, they share the available bus and memory bandwidth inside tile 0. Besides this bottleneck, the remote enqueue operations from different tiles have to share the NoC bandwidth on shared links.

**Summary.** The second microbenchmark revealed that SHARQ unfolds its full potential if the remote queue operations are not dominated by the payload transfer or slowed down by concurrent local dequeue operations of the receiver.

#### 4.6.3.3 Benchmark 3: Pointer-Only vs. Data Transfer

While the first and the second microbenchmark analyzed the performance of queue operations when performed individually or concurrently, the third microbenchmark analyzes the benefit of local pointer-only queue operations compared to normal dequeue operations.

**Benchmark Setup.** Therefore, remote enqueue operations including data transfer (r-EQ-S) to tile 0 are performed concurrently by all cores on all 15 tiles. All cores on tile 0 either perform normal local dequeues (l-DQ) or the combination of pointer-only dequeue (l-DQ-P) with l-APush. The remaining benchmark setup is the same as the previous one.

**Results.** The resulting normalized execution times in microseconds are depicted in Figure 4.12 for different element sizes. Figure 4.12 reveals that:

1) For element sizes smaller than 256 Bytes, SHARQ performs better with the normal local dequeue l-DQ than with the combination of l-DQ-P and l-APush because the small data transfer is faster than accessing the SHARQ accelerator twice. The SMQ does not experience this behavior since it does not require the sequentialized hardware-software interaction with a single SHARQ accelerator but can operate on the queue with four cores concurrently.
2) For larger element sizes, the benefit of local pointer-only operations without local payload transfer becomes evident for both variants. While the average duration per remote enqueue rises drastically for the scenario including the local data transfer for both SHARQ and the SMQ, the incline of the pointer-only variants is less steep. Ideally,

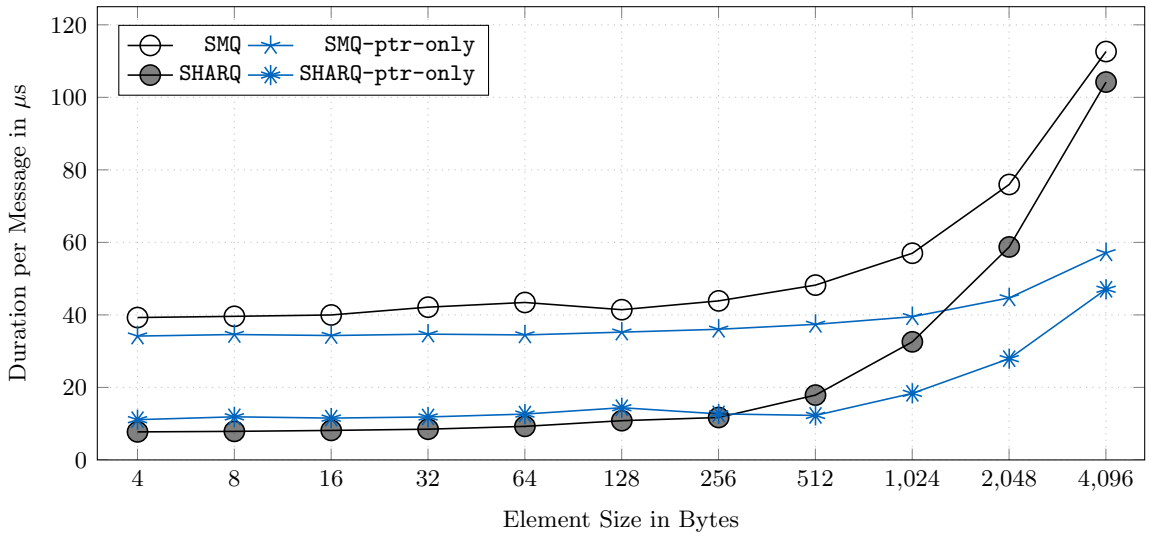



**Figure 4.12: Analysis of pointer-only queue operations.** Comparison of the average duration of local dequeue operations on tile 0 (pointer-only vs. data transfer) for different queue element sizes. Concurrently, all cores on all 15 tiles perform remote enqueue operations with payload transfer to tile 0. Since all operations share the bus bandwidth in tile 0, the rise in duration with larger element sizes is explained. Nevertheless, the overall performance is better since the pointer-only queue operations avoid the superfluous local data transfer.

the pointer-only scenarios should be independent of the element size. However, the bus and memory in `tile 0`, which are also occupied by the remote enqueue with data transfer, remain the bottleneck. Nevertheless, each pointer-only dequeue is faster so that the effective bandwidth of the enqueue operations becomes higher.

**Summary.** In conclusion, the pointer-only queue operations avoid the superfluous local data transfer. Thereby, they enable `SHARQ` to achieve its full potential, even under heavy contention of the queue.

### 4.6.4 Marcobenchmarks

While the previous section evaluated the `SHARQ` operations with and without concurrency on the queue, this section analyzes the impact of `SHARQ` when integrated into entire applications. As described in Section 4.5.6, `SHARQ` is employed to optimize the MPI inter-process communication. Thus, the evaluation in this section is based on the MPI-based NAS Parallel Benchmarks (NPB) [183], which were characterized in Section 2.4.3.

Based on these benchmarks, the following analysis is threefold. Section 4.6.4.1 and Section 4.6.4.2 evaluate the performance and scalability of `SHARQ`, respectively. Two points of reference are used in these sections: `SHARQ` is compared against the `SMQ` approach, as well as the previously existing MPI library implementation `BASE` of the prototype, which did not use queues to manage the registration of sender and receiver. `SHARQ` uses remote enqueue operations with synchronous data transfer `r-EQ-S`, and normal local dequeue operations `l-DQ` also with data transfer. This is because of the small queue element size of 68 Bytes required for the use case. With the results of Figure 4.12 in mind, the `l-DQ` performs better than the combination of `l-DQ-P`) with `l-APush` for element sizes smaller than 256 Bytes. Performance comparison of these two variants resulted in minor degradation of -3% to 1%

of the pointer-only variant. Thus, in the remainder of the evaluation, solely the SHARQ variant with normal local dequeue operations is used. Finally, Section 4.6.4.3 investigates the benefit of SHARQ's dynamic memory management when applied to the NAS benchmarks.

### 4.6.4.1 Performance Analysis

The first analysis evaluates the performance speedup, or in other words, the reduction in execution time when using SHARQ. Therefore, the benchmark execution times of the five [4] NAS benchmark kernels, available on the 4x4 tile design, are measured for the SHARQ, SMQ, and BASE variants. Figure 4.13 depicts the relative benchmark execution times of

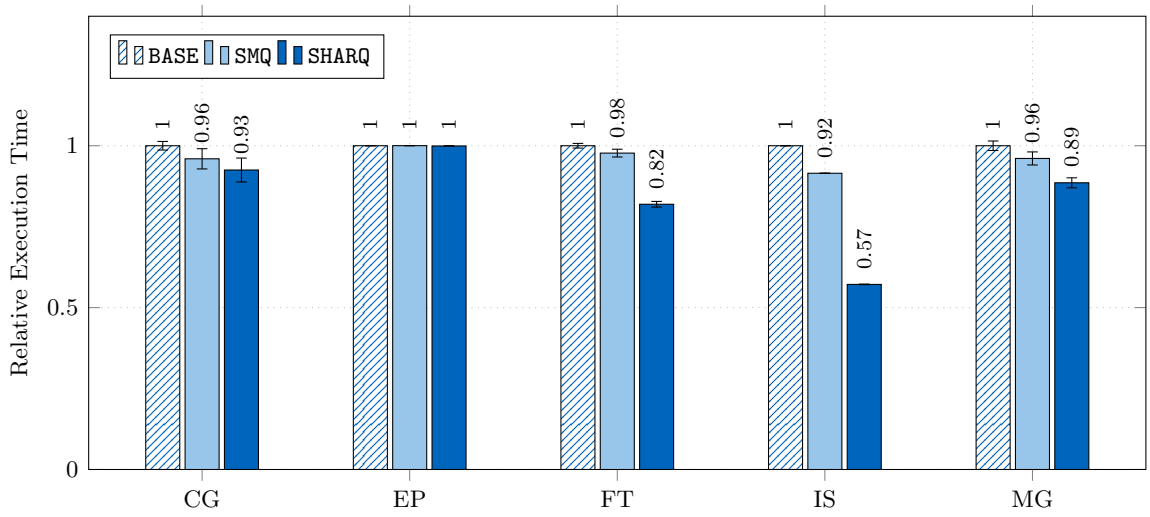



**Figure 4.13: NAS benchmark performance of SHARQ.** SHARQ integrated in the MPI-library and evaluated with the MPI-based NAS benchmarks in the 4x4 tile configuration. SHARQ accomplishes a reduction in execution time of up to 43% for the communication-intensive IS kernel.

the five benchmarks for the three variants as a bar plot diagram. Each triple of bars is normalized to the respective BASE variant. It can be observed that most benchmarks benefit from a queue-based implementation of the MPI inter-process communication and from SHARQ in particular. Solely, the EP benchmark is indifferent to this optimization since it is embarrassingly parallel and essentially does not communicate. For the remaining kernels, the SMQ achieves a reduction in execution time between 2% (FT) and 8% (IS). In contrast, SHARQ even accomplishes a reduction of 7% (CG) up to remarkable 43% (IS). In both cases, the speedup of the IS benchmark is most considerable since it is the most communication-intensive kernel [184]. Thus, SHARQ facilitates a more efficient inter-process communication of the MPI library. The higher the communication needs of the kernel are, the faster the performance speedup of the entire benchmark.

### 4.6.4.2 Scalability Analysis

The previous conclusion becomes even more evident when performing the second analysis. The scalability of SHARQ with higher communication demands is evaluated. Therefore, the

[4] Note that of the eight NAS benchmark kernels (i.e., CG, EP, FT, IS, MG, BT, SP, and LU), only five (i.e., CG, EP, FT, IS, and MG) exist for the input set running on 64 cores on the 4x4 tile design.

most communication-intensive IS kernel is executed on different system sizes, which results in varying amounts of communication. In the 1x1 design, only tile-local communication between four local cores is made. In contrast, in the 4x4 design, all 64 cores communicate with one another, obviously including much remote communication.

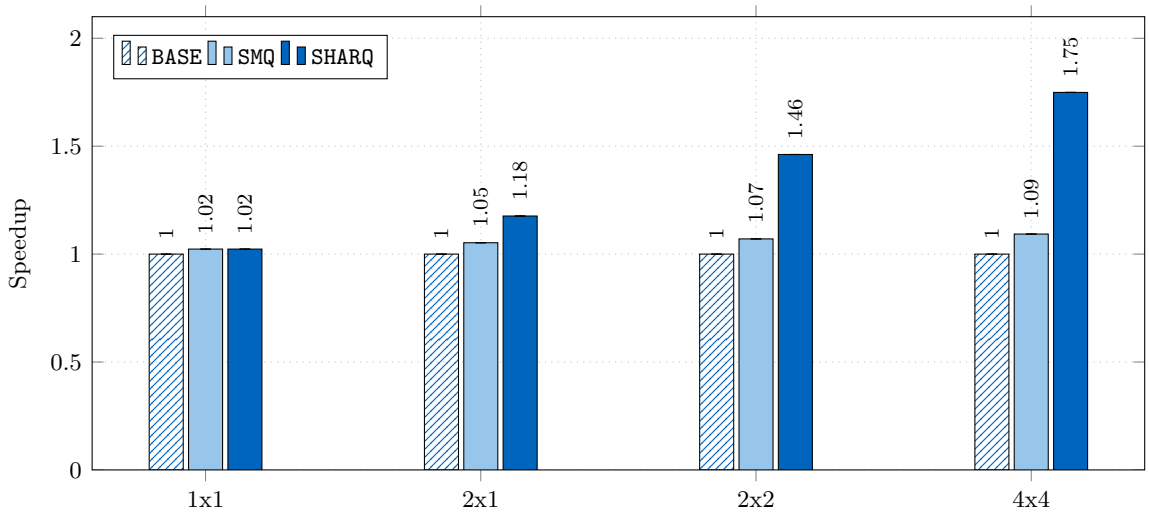



**Figure 4.14: NAS-IS benchmark scalability analysis of `SHARQ`.** Relative speedup of the communication-intensive IS kernel compared for different system sizes. Especially bigger systems with increased communication demand profit from `SHARQ`'s efficient inter-process communication.

Figure 4.14 depicts the attained speedup of the IS benchmark for the different system sizes as a bar plot diagram. Each triple of bars is normalized to the respective `BASE` variant. As expected, the 1x1 variant has no significant advantage of `SHARQ` since no remote communication occurs. However, when increasing the design size, the benchmark exhibits increased communication. Thus, the larger the system, the more `SHARQ` demonstrates its potential. Whereas the speedup achieved by the `SMQ` is between 5% (2x1) and 9% (4x4), `SHARQ` reaches between 18% (2x1) up to 75% (4x4) improvement in performance. Thus, especially bigger systems with increased communication demand profit from `SHARQ`'s efficient inter-process communication.

#### 4.6.4.3 Dynamic Memory Management Evaluation

Finally, this section evaluates the dynamic memory management feature of `SHARQ`. Besides analyzing the required memory footprint, the potential impact on the benchmark performance is quantified.

**Previous Static Queue Size.** Initially, all `SHARQ` queues in the system had a static queue size without the dynamic memory management feature enabled. Each queue needs to be able to receive an enqueue request by all other cores in the system since the communication between sender and receiver is synchronous. Thus, each queue is dimensioned as follows in the static variant on a 2x2 and 4x4 tile design, respectively:

$$\begin{aligned} \texttt{\#buffers}^{\texttt{static-2x2}}_{\texttt{per-queue}} &= \texttt{\#cores}^{\texttt{2x2}}_{\texttt{total}} - 1 = 16 - 1 = 15 \\ \texttt{\#buffers}^{\texttt{static-4x4}}_{\texttt{per-queue}} &= \texttt{\#cores}^{\texttt{4x4}}_{\texttt{total}} - 1 = 64 - 1 = 63 \end{aligned} \tag{4.1}$$

The total static size of all queues in the system accumulates to:

$$\begin{aligned} \texttt{\#buffers}_{\texttt{total}}^{\texttt{static-2x2}} &= \texttt{\#cores}_{\texttt{total}}^{\texttt{2x2}} \cdot \texttt{\#buffers}_{\texttt{per-queue}}^{\texttt{static-2x2}} = 16 \cdot 15 = 240 \\ \texttt{\#buffers}_{\texttt{total}}^{\texttt{static-4x4}} &= \texttt{\#cores}_{\texttt{total}}^{\texttt{4x4}} \cdot \texttt{\#buffers}_{\texttt{per-queue}}^{\texttt{static-4x4}} = 64 \cdot 63 = 4032 \end{aligned} \tag{4.2}$$

The queues with static size were allocated at the beginning of the benchmark, irrespective of potentially varying queue size requirements at run-time. This worst-case dimensioning could be improved by e.g., a static analysis of the application. Thereby, application-specific static queue sizes could be obtained that might be lower than the worst-case.

Nevertheless, to adapt to changing needs or varying communication patterns at run-time, SHARQ's dynamic memory management feature is required and employed in the sequel.

**SHARQ Configuration.** Therefore, at the time of queue creation, SHARQ's memory management task is registered in the queue descriptor of each queue. Moreover, each queue descriptor is allocated with only one initial element buffer in its partly-filled allocator stack. The memory management parameters in the queue descriptor are configured as follows. The lower and upper threshold for the minimum and maximum number of hardware-owned element buffers are `min # free elem = 1` and `max # free elem = 5`, respectively. Furthermore, SHARQ should only schedule the memory management task if these thresholds are exceeded for at least `# access until = 10` queue operations.

The memory management task's goal in this specific use case is to keep the number of hardware-owned element buffers in this window between 1 and 5 elements. Thus, when triggered and executed, it adjusts the fill level of the allocator stack to 3. If fewer buffers are in the allocator stack, the memory management task allocates additional buffers in software and performs a `sharqAllocatorPush` to hand them over to hardware. Accordingly, if too many buffers are available, they are popped from the allocator stack (`sharqAllocatorPop`), and their memory is freed.

**Benchmark Setup.** For evaluation purposes, the number per queue $\texttt{\#buffers}_{\texttt{per-queue}}^{\texttt{dynamic}}$ and the total number $\texttt{\#buffers}_{\texttt{total}}^{\texttt{dynamic}}$ of circulating buffers in hardware and software is tracked by the memory management task. These comprise the hardware-owned buffers in the allocator stack and bound buffer and the currently software-owned buffers. Thus, the memory consumption of active buffers in the MPI layer is tracked and compared to the previously static memory consumption.

In the following, two aspects are analyzed. First, the dynamic memory consumption $\texttt{\#buffers}_{\texttt{queue-0}}^{\texttt{dynamic-4x4}}$ of queue 0 (i.e., the queue corresponding to tile 0 core 0, which is the entry point of the application) is evaluated in the 4x4 design. Second, the accumulated memory consumption $\texttt{\#buffers}_{\texttt{total}}^{\texttt{dynamic-2x2}}$ of all queues in a 2x2 design are analyzed.

**Results.** Figure 4.15 depicts the dynamic memory consumption $\texttt{\#buffers}_{\texttt{queue-0}}^{\texttt{dynamic-4x4}}$ of queue 0 over time for the five NAS benchmarks available on the 4x4 design. The execution time is normalized for each kernel to be able to plot them into the same diagram. Additionally, the static queue size $\texttt{\#buffers}_{\texttt{per-queue}}^{\texttt{static-2x2}} = 63$ is plotted as a reference.
Figure 4.15 reveals the following insights:

1) Most of the dynamic memory management occurs in the early stages of the application since the `initial fill = 1` of the allocator stack is low. This is most notably for the FT kernel, which peaks at the maximum of 63 buffers at the beginning.

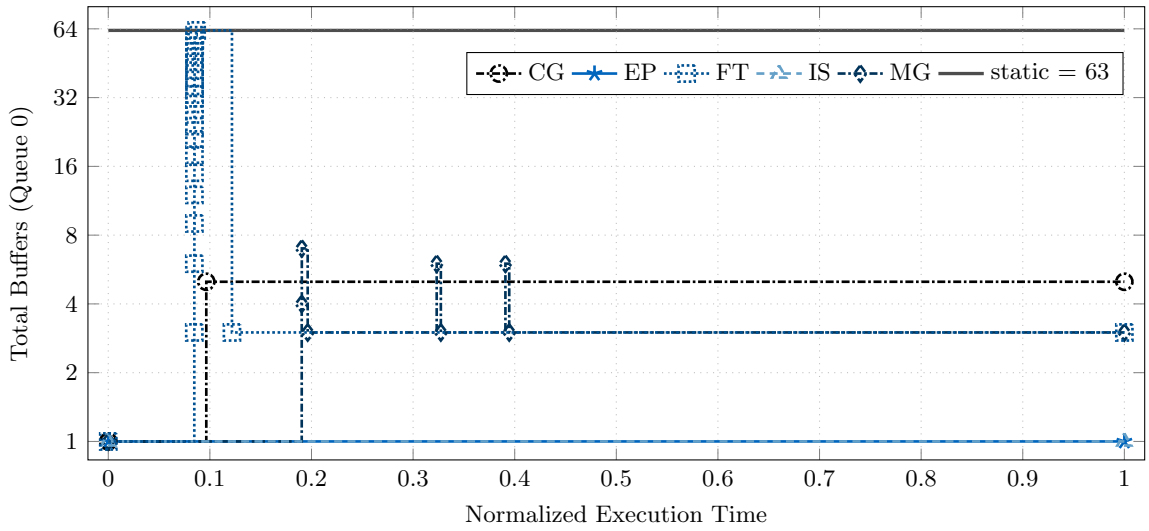



**Figure 4.15: Dynamic memory footprint of NAS benchmarks (queue 0 only).** NAS benchmarks executed in the 4x4 design. `initial fill = 1`, `min # free elem = 1`, `max # free elem = 5`, `# access until = 10`. The dynamic memory consumption is reduced significantly while the performance is impacted only marginally in most cases.

2) Subsequently, for all kernels, $\texttt{\#buffers}^{\texttt{dynamic-4x4}}_{\texttt{queue-0}}$ more or less finds a steady-state. Only minor and short variations appear for the MG kernel in the middle of the execution. While the CG kernel rises and stays at the upper threshold of 5 buffers, the EP kernel is satisfied with the single initial buffer as the EP (embarrassingly parallel) basically does not communicate.

3) While static analysis of the individual kernels would improve the worst-case static queue size of CG to 5 or MG to 7, it would not reduce the static queue size for the FT kernel. Thus, the memory consumption of FT profits significantly from dynamic memory management. In more than 96.3% of the execution time, FT suffices with three buffers. The peak of 63 buffers is only for a short period of 3.7%. Thus, over the entire execution time, FT would save 95.2% of the memory footprint for queue 0.

4) Quantifying the impact on the performance leads to a slight degradation of 5% for the FT kernel in the 2x2 design. This needs to be put in perspective to the 95.2% of reduced memory footprint for queue 0. Moreover, the CG, EP, and MG benchmarks experience no performance drop.

   Solely, the IS benchmark suffers from a 10% performance reduction. Interestingly, the $\texttt{\#buffers}^{\texttt{dynamic-4x4}}_{\texttt{queue-0}} = 1$ for IS over the entire execution time. It seems like no dynamic memory management was performed, resulting in unintelligible performance degradation. However, first, the behavior can be explained by the delayed triggering of the memory management task. `SHARQ` only schedules the task if the thresholds are exceeded for longer than `# access until = 10` queue operations. Until then, IS may encounter a full or empty queue. Second, solely queue 0 is shown. An analysis of all queues will be performed in the sequel.

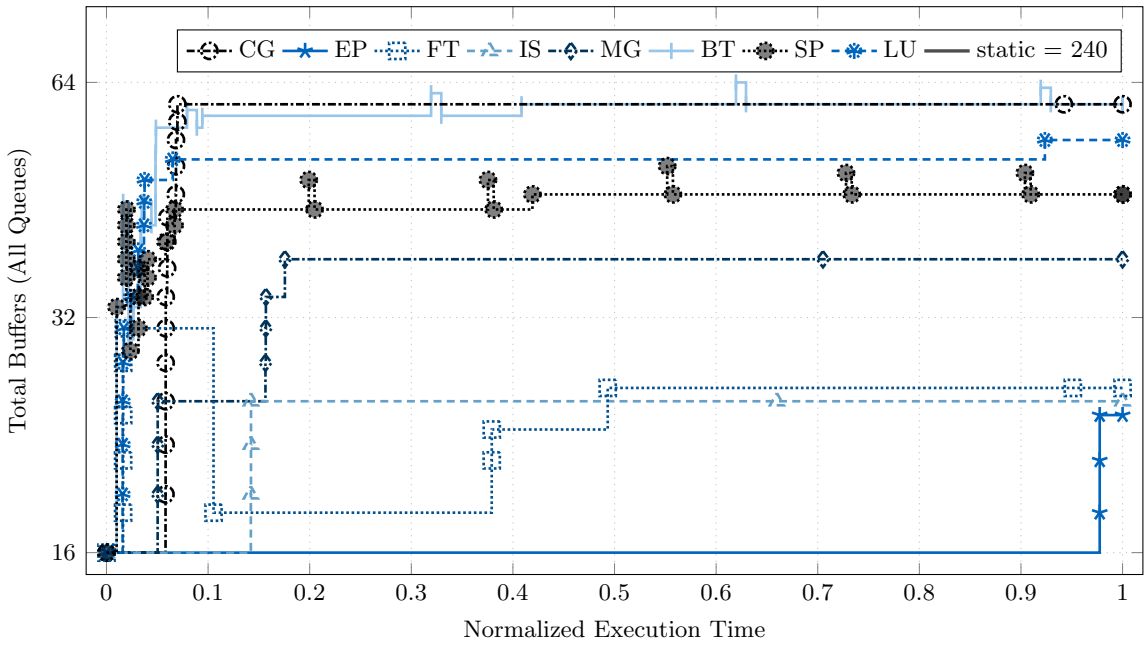



**Figure 4.16: Dynamic memory footprint of NAS benchmarks (all queues accumulated).** NAS benchmarks executed in the 2x2 design. `initial fill = 1`, `min # free elem = 1`, `max # free elem = 5`, `# access until = 10`. The dynamic memory consumption is reduced significantly while the performance is impacted only marginally in most cases.

The evaluation of all queues' accumulated memory footprint is depicted in Figure 4.16 for a 2x2 design. Here, all eight NAS benchmark kernels exist. When taking all queues into account, more action is observed:

1) As before, much dynamic memory management occurs in the early stage (i.e., normalized execution time below 0.1). All kernels start with 16 buffers (i.e., one buffer for each of the 16 cores in the 2x2 design) and dynamically adjust until they reach their steady-state.
2) Once a steady-state is reached, most benchmarks experience only minor fluctuations around it. However, the FT kernel drops at around 0.1 normalized time and stepwisely rises again at around 0.37 and 0.5, respectively. It undergoes different phases of the application.
3) The BT kernel obtained the least savings in memory footprint compared to the static variant with 240 buffers, which peaks at 64 buffers. However, FT will profit since its queue sizes are dynamically adjusted to the different application phases. Also, EP exhibits only a late rise at 0.97 from 16 to 24 buffers.
4) Analyzing the impact on the performance reveals no degradation for EP, FT, MG, BT, and LU and only minor degradation of 1% for SP and 3% for CG. Again, IS suffers most with 8%.

**Summary.** Since `SHARQ` monitors and dynamically adapts to run-time requirement changes, its memory footprint can be optimized. In most cases, the performance of the benchmark is only marginally impacted. However, the memory consumption was significantly reduced.

## 4.7 Summary

The chapter presented SHARQ to facilitate efficient inter-process communication. The lessons learned from the remote atomic operations of Chapter 3 were extended to full-fledged hardware-accelerated queue management. SHARQ provides the flexibility and configurability known of software queue approaches since SHARQ queues are software-defined in normal memory. In contrast, performance and scalability are achieved by outsourcing the entire queue and memory management, including data transfers to the hardware. Furthermore, a smart hardware-software interaction is employed. SHARQ is able to schedule a handler task (responsible for processing queued elements) on-demand, monitor the queues memory footprint, and dynamically adjust to run-time requirement changes of the queue sizes.

The evaluation revealed that SHARQ unfolds its full potential for remote queue operations and in bigger systems. A software-managed queue alternative SMQ is outperformed significantly. Thus, SHARQ is a powerful extension of the portfolio of inter-process communication primitives. It tackles the recurring problems of remote inter-process communication by caring about data-to-task locality and minimal software involvement.

While this chapter focused on queue-based communication, Chapter 5 addresses a more complex scenario where inter-process communication requires object graph transfers.

# 5 Near-Memory Graph Copy Accelerated Inter-Process Communication

> "Don't spend time beating on a wall, hoping to transform it into a door."
>
> — Coco Chanel [188]

The previous chapters presented near-memory accelerators to optimize the management of concurrent data structures, such as queues, stacks, or semaphores, which are extensively leveraged by essential library and OS routines. Chapter 3 focused on lightweight atomic operations on word granularity (such as CAS, semaphore increment/decrement, stack push/pop, pointer enqueue/dequeue), which are mostly missing in tile-based architectures. However, the remote near-memory acceleration proved crucial to alleviating the overhead software-managed inter-process synchronization. Chapter 4 extended this approach to a full-fledged near-memory hardware queue management, including data transfers and receiver notification to enable performant and flexible inter-process communication. Both approaches revealed that data-to-task locality and minimized software involvement are crucial in systems with physically distributed memories.

So far, the handled data was flat, i.e., it resided at contiguous address ranges. Thus, either references to data elements were managed, or it was possible to directly and efficiently copy data with a DMA engine. However, today's high-level programming languages have evolved from simple and flat data structures like strings and arrays to more complex data structures like object graphs. In parallel applications, libraries, and operating systems, the transfer of data and its subsequent processing on remote tiles is a typical communication pattern. Since a DMA engine is not capable of adjusting the copied pointers, unfortunately, it cannot be leveraged to copy object graphs. Nevertheless, it is crucial for the performance of object-oriented applications on tile-based architectures that efficient transfers of object graphs are supported.

Therefore, this chapter presents a near-memory graph copy hardware accelerator that achieves this goal. Section 5.1 further motivates the approach and refines the problem statement. Then, Section 5.2 defines the aspired goals and requirements for the hardware-software co-design approach followed by Section 5.3 which explains the assumptions made and the given boundary conditions. To identify and quantify the potential for improvement, Section 5.4 investigates the inter-process communication mechanism using several qualitative and quantitative metrics. Based on this investigation's findings, Section 5.5 presents the concept of the proposed near-memory graph copy accelerator and explains how it satisfies the goals and requirements. Furthermore, Section 5.6 describes the details of the design and implementation. The in-depth evaluation of the approach is covered in Section 5.7 before the chapter is concluded in Section 5.8.

## 5.1 Motivation and Problem Statement

Modern high-level programming languages face a manifold of challenges on tile-based many-core architectures. First, many languages are missing architecture awareness for the physically distributed memory and processing nodes. Second, the architectures miss hardware support for object-oriented data structures like graphs, in contrast to e.g., a DMA engine for copying flat data. These challenges are further complicated when tile-based architectures do not provide hardware support for inter-tile cache coherence and memory consistency.

The PGAS programming model is a well-known attempt to tackle these challenges. As introduced in Section 2.3.5, PGAS associates specific memory partitions to each processor or tile and requires a message-passing-like protocol for its inter-process communication beyond coherency borders. This thesis considers the object-oriented PGAS programming language X10 to showcase its contribution. In X10, any inter-process communication between tiles/places is exclusively possible via the so-called `AT statement`.

In Section 2.3.5, two related approaches of this PGAS inter-process communication mechanism between tiles were presented. Both have their benefits and drawbacks. The serialization-based approach utilizes a DMA engine for inter-tile data transfers. However, the (de-)serialization overhead is costly. The cloning-based `PEGASUS` approach reveals complementary attributes. While it avoids the (de-)serialization overhead, it performs the graph copy operation in step (3) via remote load-store operations through the L2 cache and over the NoC. These remote memory operations will limit the performance since memory access latencies are much higher, and data-to-task locality is missing. In addition to performing the graph copy *far from memory*, no hardware accelerator for the graph copy operation (similar to the DMA engine used in the serialization-based approach) is leveraged or available to relieve the cores of this duty.

Because of the high relevance of efficient inter-process communication and the paramountcy of data-to-task locality, a solution to mitigate these problems is required.

## 5.2 Goals and Requirements

This section defines essential goals and requirements for optimizing the inter-process communication on tile-based many-core architectures following the PGAS programming paradigm. Although the programming language X10 is used to showcase the contribution, the goals and requirements of the concept are defined in a way so that they apply to other languages and architectures as well.

The overall goal is to develop a more efficient implementation of the PGAS inter-process communication procedure to mitigate its data locality challenges. A hardware-software co-design approach is envisioned that inherently integrates near-memory accelerators into the run-time system of the tile-based architecture. Thus, no changes to the API or application code are required while the benefits of higher performance and power efficiency of hardware accelerators are exploited. The synergy between the hardware accelerators and the architecture-aware system software should provide a performant and scalable mitigation approach for the inter-process communication overhead. The following specific functional and non-functional requirements are demanded of the approach.

**Performance** Although the approach targets inter-process communication, it must affect the performance of the entire application to mitigate the locality wall.

**Scalability** As many-core architectures contain dozens or hundreds of cores, the proposed concept must be scalable to these system sizes.

**Memory and Interconnect Traffic Reduction** The near-memory concept must reduce the interconnect and memory traffic of the inter-process communication as these heavily contribute to the energy footprint and data access latencies of the application.

**Reduced Cache Polution** The approach should produce less unnecessary cache pollution, which can arise from the graph copy operations. Since today's data sets may easily outgrow the cache capacity, this is especially important for the cache hit rate experienced by other cores sharing the cache.

**Applicable to Multiple Memories** In order to avoid access hot spots, tile-based architectures often comprise several physically distributed memory tiles. Therefore, the proposed concept must also provide a solution to efficiently copy object graphs between physically separated memory partitions located in different tiles.

**Minimal Software Involvement** Since inter-process communication should be as lean as possible, the software involvement should be minimal so that the cores can focus on processing the actual application workload.

**OS Integration** Ease of programmability and widespread compatibility should be maintained by replacing the existing PGAS inter-process communication mechanism between tiles with the proposed hardware-software co-design approach at the operating system level, i.e., opaque to the application programmer.

## 5.3 Assumptions and Constraints

After defining the desired goals and requirements of the approach, this section elaborates on the assumptions made and the given boundary conditions for the design. If one intends to apply the concept to other architectures or software environments, the assumptions and constraints have to be considered.

**PGAS IPC** The contribution presented in this chapter targets the inter-process communication between tiles of the PGAS programming paradigm. PGAS associates specific memory partitions to each processor or tile and requires a special message-passing-like protocol for inter-process communication between tiles.

**PGAS programming language X10** In this work, the object-oriented PGAS programming language X10 is considered. It divides the global address space into separate memory partitions assigned to the individual tiles. In X10, a so-called place (i.e., a tile and its corresponding memory partition) must be cache-coherent, which is in line with the provided intra-tile hardware cache coherence of the used demonstrator platform. However, X10 does not assume or require inter-tile cache coherence between tiles/places. Thus, a thread running on a specific tile is only allowed to freely access the memory partition associated with this tile (i.e., tile local memories and caches and its memory partition in the global shared memory). Whenever IPC access to another tile's memory partition is required, a special RPC-like programming construct (X10's so-called `AT statement`) needs to be performed, which requires object graph copies between memory partitions.

**No MMU or Virtual Memory** As for CACAO and SHARQ, the prototype design does not use an MMU or virtual memory. If one desires to incorporate those, it would have to be ensured that the accelerators are able to look up the appropriate address translations.

**Access to Hardware Registers** As for CACAO and SHARQ, it is assumed that direct access to hardware registers is available to interact with the accelerators in a lightweight manner. It needs to be discussed if accelerator access without memory protection is allowed on other systems.

**Asynchronous Offloading, Completion Notification, and Hardware Scheduling** It is assumed that the near-memory accelerator calls can be offloaded asynchronously by software. Thus, a notification of the completion becomes necessary. While other platforms might be limited to sending interrupts from the accelerator to the cores, this contribution, similar to SHARQ, relies on direct accelerator interaction with the hardware scheduler. Although the hardware scheduler and the used OctoPOS operating system (cf. Section 2.4.2) are no conceptual requirements, they enable optimizations that would not be possible without them. For more information about the hardware scheduling, the reader is referred to the paragraph "Interruptless Hardware Scheduling" in the "Assumptions and Constraints" Section 4.3 of SHARQ (cf. Page 65) which applies as is.

**Triggering of other Hardware Units** It is assumed that the proposed accelerators are able and allowed to trigger other hardware units without further software involvement. For instance, the near-cache accelerator may initiate a data transfer and task invocation to the destination tile via the network adapter or trigger the near-memory accelerator directly. Thus, the calls to the near-cache accelerator can also be offloaded from software asynchronously.

**Graph Representation, Object Model, and Auxiliary Data Structures** It is assumed that the object graph representation is compatible with hardware acceleration, i.e., the accelerator has to be able to retrieve all necessary information about a particular graph from memory without up-calls to software. This includes e.g., the graph's object types, graph copy methods, graph copy memory management, and the interaction with the hardware scheduler. Necessary adaptions to the object model, as described in Section 5.6.2.1 need to be allowed. The design supports arbitrary graphs following the Java-like X10 object model with graph sizes up to 65535 objects. Beyond that, adaptations to the hardware accelerators would be required. Furthermore, it is assumed that required auxiliary data structures, as explained in Section 5.6.2.2, will be pre-allocated and provided by software but managed by the accelerators.

**No Concurrent Graph Access by Software** It is assumed that the software is not allowed to modify the source or destination graph concurrently during an ongoing graph copy which is ensured by X10.

**Garbage Collection Support Missing** In the current design, the support for garbage collection is limited as no additional allocator information is added before the individual objects by the accelerator. However, this can be easily extended in future work. Two possible solutions will be laid out in Section 5.6.2.2.

## 5.4 Pre-Investigation

After describing the goals, requirements, assumptions, and constraints in the previous two sections, this section performs an in-depth pre-investigation of the PGAS inter-process communication to identify and quantify the potential for improvement.

The goal of the investigation is to show that the inter-process communication mechanism

1) contributes a significant portion to the overall execution time,
2) is a very memory intensive task with a lot of remote memory accesses, and
3) has a degrading impact on the entire application.

Moreover, the sub-tasks with the highest potential for improvement through near-memory and potentially near-cache acceleration will be identified. The evaluation methodology builds upon one qualitative and several quantitative metrics, which are described in Section 5.4.1 and analyzed in Section 5.4.3 and Section 5.4.4, respectively. Section 5.4.2 explains why the in-depth analysis is only carried out for `PEGASUS`.

### 5.4.1 Methodology

One qualitative and four quantitative metrics are used to analyze the inter-process communication mechanism utilized between tiles/places.

#### 5.4.1.1 Qualitative Metric

The qualitative metric analyzes the necessary sub-tasks of the `AT statement` in more detail than in Section 2.3.5. Alongside the graph writeback and graph copy operations, memory allocations, transfer of metadata information, notifications, or wait statements are also considered. A particular focus is on costly remote memory accesses as they have a much higher latency and lower bandwidth. This metric will reveal insights into which sub-tasks exist so that they can be further analyzed quantitatively.

#### 5.4.1.2 Quantitative Metrics

Four useful metrics are defined to evaluate the concept and identify and quantify the bottlenecks and weak spots of the related approaches. The measurements to extract these metrics are integrated into the hardware and run-time system of the evaluation platform. Both hardware performance counters and software time measurements were employed.

In the following, several different times are used to define the metrics. These are the benchmark execution time $t_{exec}$, the pure application processing time $t_{proc}$, the time for inter-process communication between tiles $t_{ipc}$, the active time of the cores $t_{active}$, and the idle or sleeping time of the cores $t_{idle}$. Their meaning is illustrated in Figure 5.1 for a parallel application running on $N_{cpu}$ cores.
While $t_{exec}$ is the elapsed wall clock time from the start until the end of the application's region of interest (RoI), $t_{proc}$, $t_{ipc}$, $t_{active}$, $t_{idle}$ are the accumulated times over all application cores, respectively. Thus the following equations hold true:

$$t_{active} = t_{proc} + t_{ipc} \tag{5.1}$$

$$t_{active} + t_{idle} = N_{cpu} \cdot t_{exec} \tag{5.2}$$

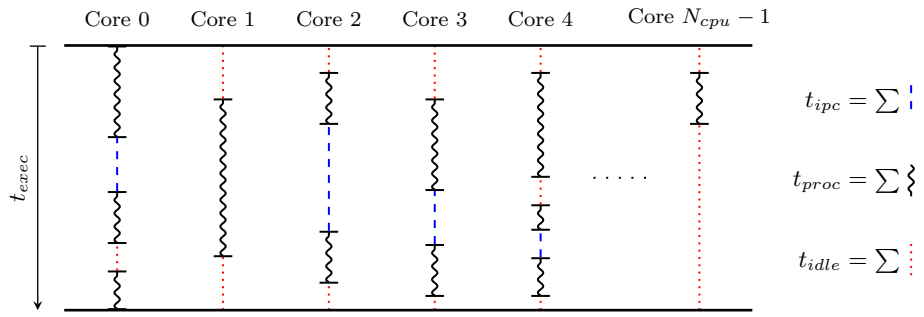



**Figure 5.1: Illustration of benchmark times.** Example application scenario including the definitions of $t_{exec}$, $t_{proc}$, $t_{ipc}$, and $t_{idle}$.

**Benchmark Execution Time.** The first metric is the overall benchmark execution time $t_{exec}$, i.e., the duration of the RoI. This metric allows comparing the performance of different implementations. It is measured by reading a 64-bit timestamp counter provided by the MPSoC prototype before entering the RoI and after leaving the RoI and calculating the difference between them. This metric is non-intrusive as the measurements do not influence the execution time.

**Relative Communication Time.** The second metric is the relation of the communication time $t_{ipc}$ compared to the total run-time, i.e., active time, of the cores:

$$R_{ipc} = \frac{t_{ipc}}{t_{active}} = \frac{t_{ipc}}{t_{proc} + t_{ipc}} \tag{5.3}$$

This metric quantifies how significant the portion of the inter-process communication between tiles is during the benchmark execution, which is the first goal of this pre-investigation. It helps decide whether an improvement of the inter-process communication mechanism can have a noticeable effect on the overall benchmark.

The active time $t_{active}$ is read non-intrusively from the cores' hardware performance counter registers after the RoI. The counters are only activated during the RoI. The time for inter-process communication $t_{ipc}$ is measured differently, as it needs to be more fine-grained during the RoI. Every time the `AT statement` is called, the timer is started, and it ends whenever the `AT statement` function returns. The `AT statement` time $t_{ipc}$ will further be split into the sub-tasks graph writeback $t_{wb}$, graph copy $t_{gc}$, and the remaining parts (e.g., allocations or notifications) of the `AT statement` $t_{ipc,rest}$. These fine-grained measurements of the `AT statement` are slightly intrusive and influence the benchmark. This side-effect will be coped with by measuring this metric in separate benchmark runs, than e.g., the runs for the overall performance measurements.

**Memory Intensiveness – Relative Remote Memory Bytes.** The third metric characterizes the inter-process communication's memory intensiveness, which is the second goal of this pre-investigation. This metric quantifies the portion of remote memory accesses to the main memory during the inter-process communication between tiles in relation to the entire application. It helps to decide whether a near-memory integration of the mechanism is worthwhile.

Only remote load and store memory accesses, i.e., L2 cache misses, contribute to this metric as they suffer from much higher memory access latencies and reduced memory

bandwidth than tile-local memory accesses or L1 and L2 cache hits. This metric is measured by reading out the hardware performance counters at the back-end of the L2 cache, separate for the different application phases.

As these counters are only available once per tile and not once per core, it is unfortunately not possible to extract the information which core is responsible for the specific L2 cache miss. Thus, the individual cache misses cannot be associated with either the benchmark's processing or inter-process communication. Therefore, since this metric only delivers meaningful results when running the benchmark with one core per tile instead of four cores per tile, only one core per tile is used for these measurements. Moreover, this metric is also intrusive.

**Compute Performance.** The fourth metric quantifies the compute performance in operations per second for the different application phases, which is the third goal of this pre-investigation. This metric is useful to identify whether the inter-process communication between tiles has a degrading effect on the compute performance of the entire application.

It is measured by reading the appropriate performance counters of the cores for the number of executed processing operations $Op_{proc}$ and the elapsed time. As this requires fine-grained measurements during the RoI for the individual parts, this metric is also intrusive. In addition to the processing operations in relation to the processing time, also the processing operations in relation to the total active time, as well as to the total wall clock time, are analyzed:

$$OpS_{proc} = \frac{Op_{proc}}{t_{proc}}, \qquad OpS_{active} = \frac{Op_{proc}}{t_{proc} + t_{ipc}}, \qquad OpS_{bench} = \frac{Op_{proc}}{t_{proc} + t_{ipc} + t_{idle}} \tag{5.4}$$

### 5.4.2 Rationale to Choose Pegasus for Comparison

After defining the metrics in the previous section, this section provides the rationale behind solely analyzing the cloning-based `PEGASUS` approach in the following qualitative and quantitative analysis. It is not only the closest related work to the contribution of this thesis but is also outperforming the serialization-based approach.

The benchmark execution times (i.e., the first quantitative metric) of all twelve IMSuite benchmarks were compared for the serialization-based approach against the cloning-based `PEGASUS` approach. In most cases, `PEGASUS` is equally fast or faster (8.8% on average) than the serialization-based approach, especially when both memory tiles are used to avoid access hot-spots (16.3% speedup on average). Therefore, all further analyses are solely performed on the `PEGASUS` variant.

### 5.4.3 Qualitative Analysis

After narrowing down the related approaches to the `PEGASUS` approach, the latter will be analyzed qualitatively first before also the quantitative metrics will be applied in the next section. The PGAS inter-process communication based on the `AT statement`, as introduced in Section 2.3.5, is now presented in more detail to identify performance bottlenecks and potential for improvement.

**Mechansim.** The five steps previously introduced in Figure 2.8b (Page 33) boil down to the interplay between the cores, caches, and memory that is illustrated as a message sequence chart (MSC) in Figure 5.2. Note that when executing the `AT statement` between the source

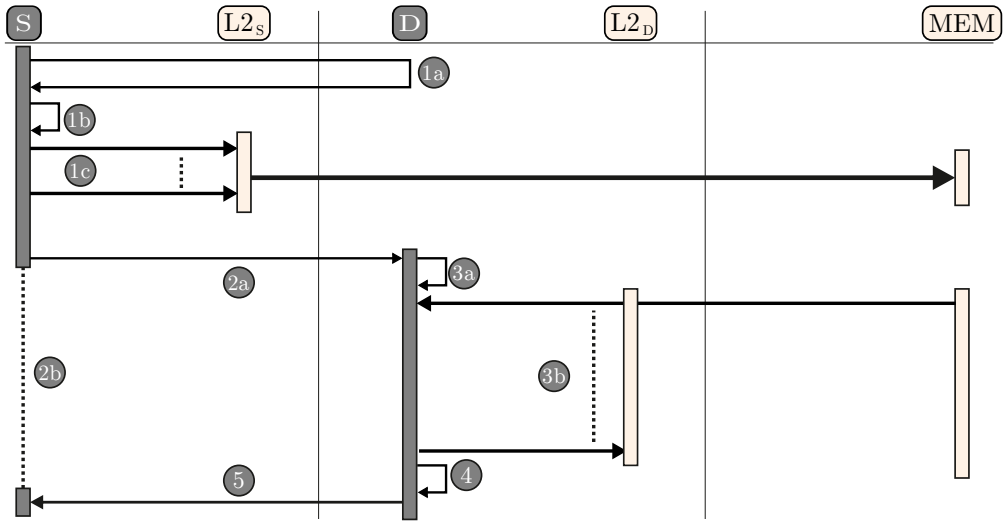



**Figure 5.2: Message sequence chart of the PEGASUS inter-process communication.** All steps are performed by software. Thick arrows indicate the high-latency remote memory accesses of the graph writeback in step (1c) and the graph copy in step (3b) via the NoC and caches.

place $S$ and the destination place $D$, additional metadata information $M$ needs to be passed between them. It contains, e.g., the pointer $G$ to the source graph, the function $func$ to be executed on the remote place $D$, or synchronization objects to notify the completion.

When the `AT statement` is called on a core on place $S$, the following steps are executed, which correspond to the steps depicted in the MSC:

(1a) The sender $S$ performs a remote allocation of the metadata buffer $M'$ on $D$.

(1b) The sender $S$ then allocates and sets up a metadata structure $M$ that needs to be transferred via DMA to the receiver $D$ (into $M'$).

(1c) $S$ synchronously writes back the source graph $G$ into its memory partition,

(2a) $S$ initiates a DMA of the metadata $M$ to $M'$ with subsequent invocation of the `remote_task` on $D$,

(2b) $S$ waits for the completion of the execution of the remote part of the `AT statement` on $D$.

Then, the function `remote_task` is scheduled and executed on a core on tile $D$:

(3a) A core on $D$ reads the metadata information from $M'$,

(3b) $D$ copies/clones the graph $G$ to $G'$ remotely via the NoC and cache hierarchy,

(4) $D$ executes the function `func` on the copied graph $G'$ (*Although this step belongs to the `AT statement` it is not attributed to the inter-process communication but to the processing part of the application.*)

(5) Either the termination of the remote procedure call is signaled back to $S$, or the resulting graph of (4) is copied back to $S$ applying the same mechanism (while reusing the metadata structures $M$ and $M'$).

Finally, the `AT statement` is also completed on the sender side and returns the result to its caller.

**Analysis.** In the underlying implementation, the metadata structure $M$ has a relatively small and constant size when compared to the varying graph sizes of $G$ for the different benchmarks (Table 2.5 on Page 39 analyzed the graph sizes of the employed IMSuite benchmarks). Thus, it can be expected that the allocations in steps (1a) and (1b) and the metadata transfer via DMA in step (2a) have a minor contribution to the duration of the entire `AT statement`. Similarly, the function call on $D$, and the notification in step (5), which is awaited by step (2b), produce scheduling overhead but no costly remote data transfers.

In contrast, the graph writeback of $G$ in step (1c) requires a complete graph traversal to reach every object of $G$ to issue writeback operations of the affected L2 caches lines of $S$. In the case of L2 cache misses during the graph traversal, remote load operations to memory partition $S$ are required. Similarly, the writeback commands result in remote stores of L2 cache lines to memory partition $S$ if the cache line was modified. Otherwise, the L2 cache performs a nop. In summary, the best case of a graph writeback occurs when the traversal only experiences L2 cache hits, and the entire graph resides in the L2 cache un-modified. This would result in no remote load or store operations at all. However, the worst case either has to read the graph from memory via remote loads completely, or the entire graph resides in the L2 cache modified, resulting in remote stores.

Furthermore, the graph copy procedure in step (3b) appears to be the most complex task. It contains the graph traversal of $G$ in the memory partition of $S$, the allocation of the new object of $G'$ in the memory partition of $D$, and the copying of each object of $G$ into $G'$ with proper pointer adjustments. The copy process produces remote load operations to the memory partition of $S$ for every object since this memory range must be invalidated in the L2 cache of place $D$ before copying to ensure inter-tile coherence. Fortunately, at least the resulting graph $G'$ may remain in the L2 cache of $D$ and is only evicted to main memory if cache line conflicts occur.

In conclusion, the graph copy operation via high-latency remote load/stores over the NoC is expected to be the largest contributor to the inter-process communication overhead, especially for large object graphs. This will be investigated quantitatively in the following.

### 5.4.4 Quantitative Analysis

Based on three of the quantitative metrics, defined in Section 5.4.1, the qualitative analysis of Section 5.4.3 will be complemented. All following results of the quantitative analysis, depicted in Figure 5.3, Figure 5.4, and Figure 5.5, have been measured for the `PEGASUS` approach on a 4x4 tile design with one core per tile and a single memory tile. The measurements will corroborate the goals of this pre-investigation that have been defined at the beginning of Section 5.4.

#### 5.4.4.1 Relative Communication Time

First of all, the relative inter-process communication time is measured and analyzed. The portion of the individual steps of the `AT statement` has been measured. Figure 5.3 depicts the results as stacked bar plots for the twelve IMSuite benchmarks. Each stacked bar contains the relative times for graph copy, graph writeback, the remaining minor parts (ipc rest) of the `AT statement` (e.g., allocations, notifications), and the actual processing time

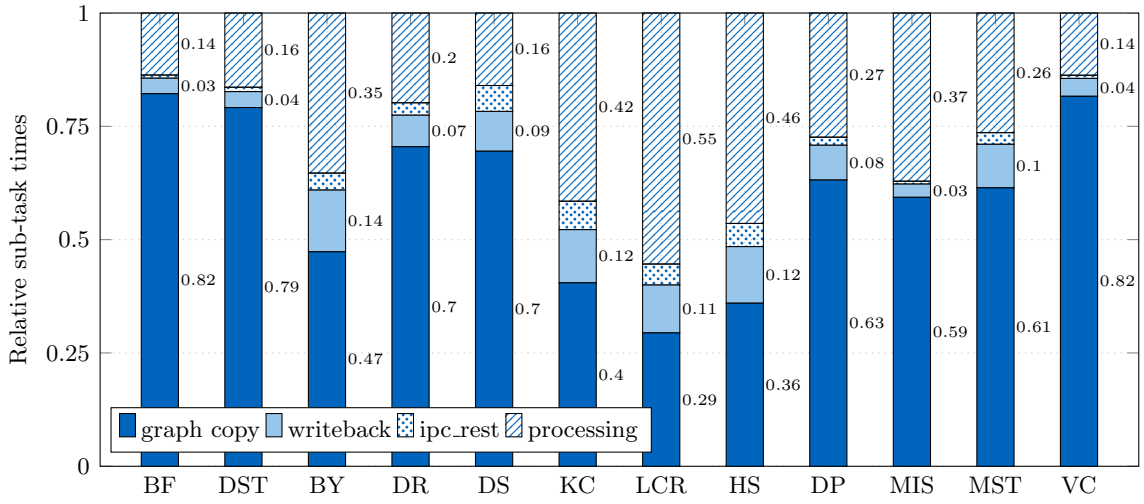



**Figure 5.3: Relative communication time of PEGASUS.** The inter-process communication, particularly the graph copy but also the graph writeback sub-tasks, contributes significantly to the IMSuite benchmarks' overall execution time. Improving these operations is crucial to performance.

of the application apart from inter-process communication. The bar for each benchmark is normalized to the respective active run-time $t_{active}$ of the cores.

Figure 5.3 shows that the portion of inter-process communication among tiles lies between 45% (for the LCR benchmark containing mostly small graphs) and a remarkable 86% (for the BF and VC benchmarks containing a majority of large graphs) with an average of 71%. The graph copy itself contributes between 29% and 82% with an average of 60%. The general observation is that the graph sizes are linked to the graph copy duration. However, it also depends on the individual benchmarks' amount and kind of processing and their inherent parallelism. Furthermore, the graph writeback operations account for 3% to 14% (8% on average) of the run-time, while the remaining parts of the `AT statement` have a small portion between 1% and 6% (3% on average). Although the relative graph copy times for the BY, KC, LCR, and HS are the lowest, their relative graph writeback times are the highest. This can be explained with a glance at Figure 5.4 which presents the highest memory intensiveness of the graph writeback operation for these four benchmarks.

In conclusion, the inter-process communication between tiles, especially the graph copy operation, contributes significantly to the benchmarks' overall execution time.

#### 5.4.4.2 Memory Intensiveness – Relative Remote Memory Bytes

While Section 5.4.4.1 analyzed the relative communication time, this section investigates the memory intensiveness of the inter-process communication mechanism. The remote memory bytes, i.e., the memory accesses caused by L2 cache misses, have been measured for the different parts of the `AT statement` (graph copy, graph writeback, remaining parts) and the actual processing part of the application. The results are presented in Figure 5.4 as stacked bar plots for the individual benchmarks.

As expected, the graph copy operation is very memory intensive as its portion of remote memory bytes contributes between 38% (LCR) and 88% (BF) (70% on average) of the total main memory accesses for the different benchmarks. While the actual processing may profit from L2 cache hits, the graph copy operation inevitably creates high-latency L2 cache misses. Thus, the inter-process communication mechanism is very memory intensive.

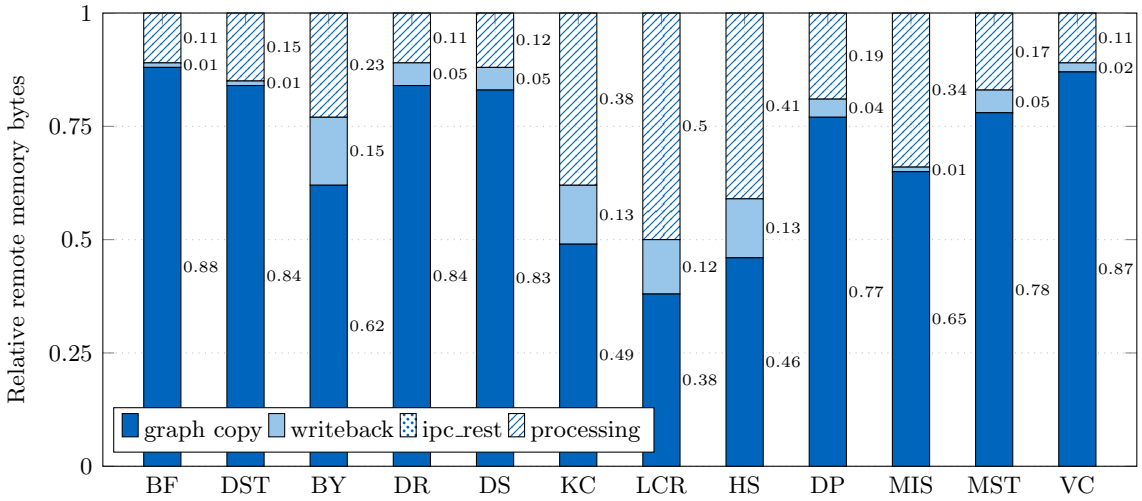



**Figure 5.4: Memory intensiveness of PEGASUS.** The inter-process communication mechanism during the IMSuite benchmarks is very memory intensive. Particularly the graph copy but also the graph writeback sub-tasks require many high-latency remote memory accesses.

#### 5.4.4.3 Compute Performance

So far, we have seen that the inter-process communication between tiles contributes significantly to the overall run-time of the benchmarks and is very memory intensive. Finally, the impact of this mechanism on the processing part of the application is analyzed.

Therefore, the compute performance of the processing operations per second has been measured for the actual processing $OpS_{proc}$ and the entire application $OpS_{active}$. Figure 5.5 depicts the results as a bar plot diagram for the twelve benchmarks, respectively. The results clearly show that the $OpS_{proc}$ for the actual processing is significantly higher than for the complete application $OpS_{active}$ (processing plus communication). It can be deduced that inter-process communication has a degrading effect on the entire application.

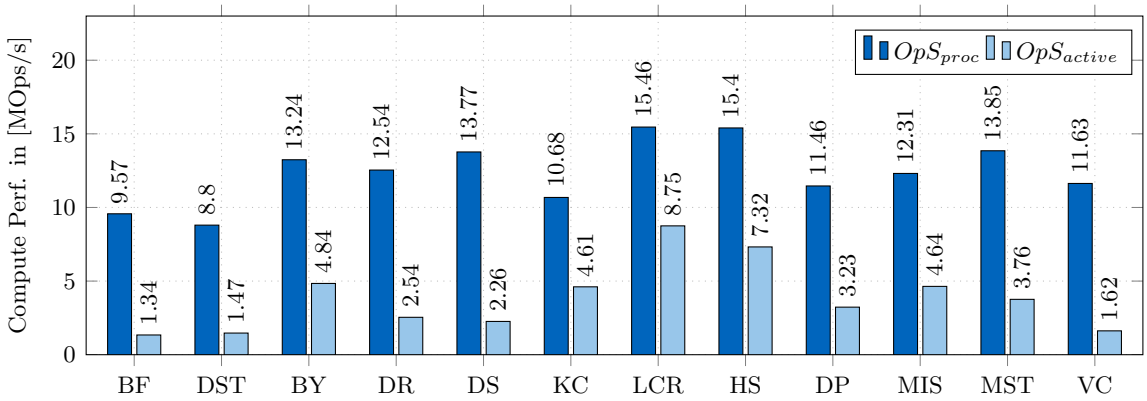



**Figure 5.5: Compute Performance of PEGASUS.** The inter-process communication has a degrading effect on the entire application since the $OpS_{proc}$ of the pure processing part are significantly higher than of the entire IMSuite benchmark $OpS_{active}$ (processing plus communication).

### 5.4.5 Summary

In conclusion, the analysis unveiled a substantial bottleneck in the PGAS inter-process communication between tiles since the PGAS memory organization requires explicit data

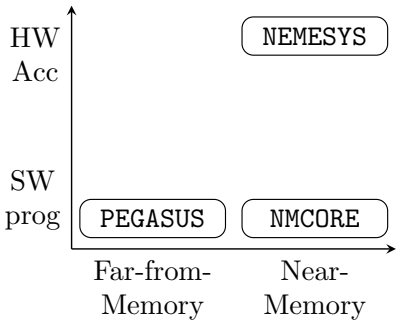
transfers between the respective memory partitions. Based on the qualitative analysis and the three quantitative metrics, the graph copy operation has been identified as the subtask with the highest potential for improvement. It contributes significantly to the overall execution time, is very memory intensive with many high-latency remote memory accesses, and has a degrading impact on the entire application. Therefore, this pre-investigation argues for optimizing the inter-process communication mechanism with a particular focus on the graph copy operation. Additionally, the graph writeback operation is the second candidate for improvement.

## 5.5 Exploration and Concept

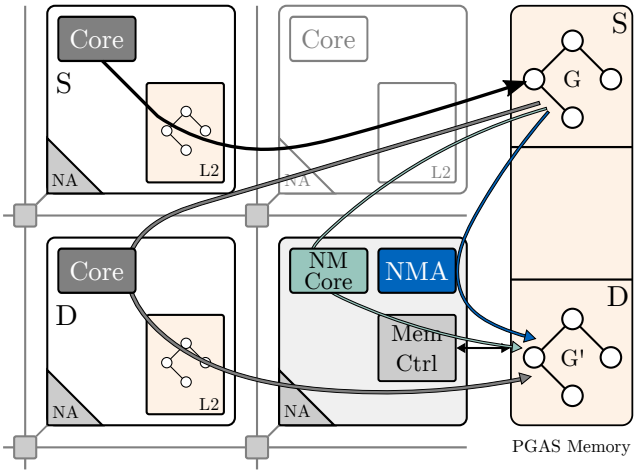
Based on the investigation's findings, this section presents the concept of the proposed NEMESYS (near-memory graph copy enhanced system-software) approach, which is able to mitigate the data locality challenges of the PGAS inter-process communication. This contribution is based on a hardware-software co-design approach that adheres to the goals and requirements defined in Section 5.2. Section 5.5.1 explores the design space, resulting in two reasonable design possibilities. First, the near-memory acceleration approach (NEMESYS) is presented in Section 5.5.2. Second, a design alternative with a near-memory core (NMCORE) is described in Section 5.5.3. While this section provides a brief conceptual overview of the approaches, Section 5.6 describes the respective implementation details.

### 5.5.1 The Design Space

The design space fundamentally contains two basic dimensions, namely the location and the type of function implementation. On a tile-based architecture, the graph copy unit's location can be either near-memory or "far-from-memory", while the function can be executed on a software programmable core or a dedicated hardware accelerator. This is illustrated in Figure 5.6a.




**(a)** The design space has two basic dimensions, i.e., location w.r.t. memory and type of function implementation.




**(b)** The three feasible design alternatives.

**Figure 5.6: The design space and feasible design alternatives.** The graph writeback is required for all variants. The graph copy has three feasible design choices. The existing PEGASUS approach via the gray far-from-memory core, the NMCOREalternative via the green near-memory core and the proposed NEMESYS approach via the blue near-memory accelerator (NMA).

Thus, there exist four possible combinations. The far-from-memory software solution corresponds to the PEGASUS variant, analyzed in Section 5.4, and depicted in Figure 5.6b in gray. The near-memory software approach results in the near-memory core (NMCORE) variant, i.e., a software programmable core integrated near-memory in the memory tile, as depicted in green. Third, the top-right corner of the design space represents the NEMESYS approach using a near-memory hardware accelerator (NMA), depicted in blue. The fourth alternative, i.e., the far-from-memory accelerator, is excluded from the analysis, as the considered memory intensive problem suffers the most from the high-latency remote memory accesses as concluded in Section 5.4.

To determine which portion of the improvement originates from the location (i.e., near- vs. far-from memory) or the type of implementation (i.e., hardware acceleration or software execution), both near-memory solutions will be presented and evaluated. This will clarify whether (1) the near-memory integration is beneficial at all, (2) the hardware accelerator outperforms the software-programmable core for the given task, or (3) if only their combination (i.e., the near-memory accelerator) achieves the goal.

Additionally, NEMESYS also incorporates the near-cache graph writeback accelerator (NCA). Theoretically, the NCA could also be combined with the NMCORE. However, since the NMCORE is not a hardware accelerator, it will not be used together with the NCA.

### 5.5.2 Concept of the Near-Memory Acceleration Approach

After having described the design space, this section presents the NEMESYS approach. This contribution mitigates the locality wall by outsourcing the memory-intensive graph copy and graph writeback operations of the PGAS inter-process communication to near-memory and near-cache accelerators, respectively. Both are integrated into the run-time system.

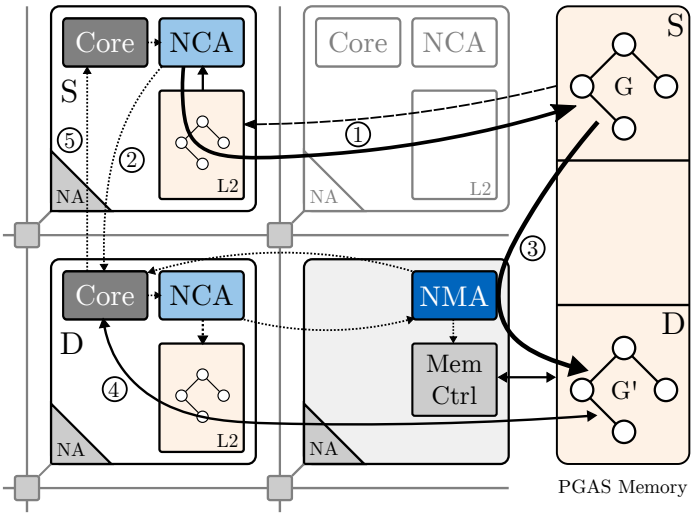



**Figure 5.7: Concept of the proposed NEMESYS inter-process communication.** The NMA is added to each memory tile, an NCA is added to each compute tile. The hardware-software interaction works as follows: (1) The NCA writes back $G$ to the memory, (2) the NCA signals a core on $D$, (3) the NMA copies $G$ to $G'$ near-memory, (4) a core on $D$ uses $G'$, and (5) the result is sent back to $S$. Bold arrows indicate bulk data transfer, dotted arrows designate control/metadata messages, and dashed arrows denote possible traffic due to cache misses/evictions.

As depicted in Figure 5.7, and similar to the PEGASUS variant, the five fundamental steps are required: (1) the source graph $G$ needs to be written back to memory, (2) the receiver $D$ needs to be signaled by the senders $S$, (3) the graph $G$ needs to be copied to $G'$, (4) the function `func` needs to be executed [1] on the graph $G'$ on place $D$, and (5) the result is sent back to $S$. Both operations that were identified as bottlenecks (i.e., the memory-intensive and cache-unfriendly graph copy (3) and graph writeback (1)) are outsourced to hardware accelerators. The near-cache accelerator (NCA) performs the graph writeback in step (1), while the near-memory accelerator (NMA) is dedicated to the graph copy task in step (3).

The following subsections elaborate on the three key attributes of the concept: near-memory integration, hardware acceleration, and system-software integration.

#### 5.5.2.1 Near-Memory Integration

Near-memory and near-cache integration are the key features of the proposed NEMESYS concept. Thereby, the data-to-task locality and effective memory bandwidth are increased. Since the graph copy operation is no longer performed remotely via the NoC but instead close to memory, the overall data movement is massively reduced. Not only are data access latencies lowered, but also the interconnect traffic over the NoC is decreased. All these enhancements contribute to overall performance improvements and energy savings, resulting in the mitigation of the locality wall.

A further benefit of the near-memory graph copy is the reduced L2 cache pollution (i.e., evictions caused or cache capacity used by the graph copy) for the other cores on the destination place $D$. This is different from the previously analyzed PEGASUS approach, which required remote load-store operations via the L2 cache during the graph copy operation. Since it is a realistic scenario for today's applications that the available cache capacity is outgrown by their data sets, minimizing unnecessary cache pollution is crucial. This phenomenon's analysis will be part of the evaluation in Section 5.7.3.7.

#### 5.5.2.2 Graph Accelerators

The key incentive for the near-memory graph copy and writeback acceleration is efficient inter-process communication for object-oriented programming languages. Although a dedicated hardware accelerator is not inherently required for the approach – since one could also migrate the graph copy task to a near-memory core (cf. Section 5.5.3) –, an accelerator implicates multiple advantages: (1) dedicated hardware modules perform the operations more efficiently with regard to performance, power, and resource utilization, (2) the cores can focus on processing actual application workload, as they are relieved from copying the object graphs, and (3) the efficient support to copy graph- and pointer-based data structures is the natural enhancement of a DMA unit that can solely copy flat data.

**Near-Memory Graph Copy Accelerator (NMA).** Figure 5.7 and Figure 5.13 (Page 127) illustrate how the NMA is integrated into every memory tile. Every core in the system can trigger the NMA. Incoming requests will be buffered in the sequence of arrival in the request FIFO of the NMA. No global software locking is required to use the NMA as the exclusive access is handled in hardware, and automatic back-pressure is created if the FIFO is full.

[1] Although this step belongs to the `AT statement` it is not attributed to the inter-process communication but the processing part of the application.

The cores trigger the NMA asynchronously, i.e., the cores do not poll for the completion of the graph copy operation. Instead, the NMA is capable of spawning/scheduling a user-defined task to a core on the destination place $D$. This mechanism is used to notify the completion of the respective graph copy operation with minimal software involvement.

The NEMESYS approach has two further requirements to be able to copy object graphs of arbitrary structure and size. First, the NMA needs to be capable of copying an object graph without using the software recursion stack. This is achieved by adopting the concept of pointer reversal introduced by Schorr and Waite [189]. The necessary state information is stored in the NMA itself and in an auxiliary data structure called the *copy map*. Second, the hardware-software co-design approach requires a small extension of the object model so that the NMA has access to the necessary object layout information. This extension will be described in Section 5.6.2.1.

In general, tile-based architectures have several physically distributed memory tiles to avoid access hot spots. However, this seems to threaten the near-memory aspect when a graph needs to be copied between these memories, as this requires remote accesses. An advanced mechanism to also efficiently master this challenge is described in Section 5.6.1.2

**Near-Cache Graph Writeback Accelerator (NCA).** Furthermore, an NCA is added to each compute tile, as depicted in Figure 5.7 and Figure 5.12 (Page 125). The NCA consists of two sub-modules (NCA-WB and NCA-RO) which are responsible for two tasks related to the graph writeback operations: First, while traversing an arbitrary object graph, the NCA-WB is issuing cache writeback commands for each cache line of every object of the graph. This operation is triggered asynchronously by a core on $S$. Thus, the NCA-WB can also dispatch a user-defined task to the destination place $D$ upon completion of the graph writeback. This replaces the signaling in step (2) by a core on $S$ and saves unnecessary additional system calls on the sender side. Second, the NCA-RO can issue range-based cache operations [2] (similar to [169]). However, it is further equipped with the ability to subsequently trigger the NMA and pass user-defined parameters to it. This is important as the destination buffer has to be invalidated on the receiving place before the actual graph copy operation to avoid data corruption during the graph copy process. Otherwise, cache evictions based on unrelated memory accesses could potentially occur.

#### 5.5.2.3 System-Software Integration

The near-cache and near-memory accelerators are tightly integrated into the system software. Since the proposed NEMESYS approach is a more efficient implementation of the PGAS inter-process communication mechanism between tiles, the necessary software adaptations are mainly limited to the internal implementation of the `AT statement` residing in the run-time system. The operating system has to provide the accelerator's hardware driver code and support memory allocations, remote task invocation, and notification signaling. Thus, the main interface of the accelerators to the operating and run-time system is the hardware driver to initiate the accelerator calls and the existing hardware scheduler, which can schedule tasks received from the accelerators upon their completion.

The detailed hardware-software interaction will be presented in Section 5.6.1. Besides, further necessary adaptations and auxiliary data structures for hardware graph copy and writeback will be motivated and described in Section 5.6.2. The asynchronous offloading of

[2] i.e., cache operations on contiguous address ranges specified by start address, length, and cache operation.

the graph copy and graph writeback operations to the NMA and NCA, respectively, relieves the cores of these duties. Since the involved cores do not have to wait for the completion of the operations synchronously, the actual application workload can be processed on them in the meantime. Up-calls to the system software are reduced to a minimum by equipping the accelerators with the respective functionality to signal their respective completion and schedule subsequent user-defined tasks. Therefore, the software involvement in the inter-process communication between tiles can be reduced to a minimum, as the entire transfer of the graph $G$ (including the required cache writebacks and invalidations) from place $S$ to place $D$ is outsourced to the hardware units. Solely, the NCA's initial triggering on the sender side and the intermediate destination buffer allocation on the receiver side (place $D$) with the subsequent triggering of the NMA remain software tasks.

This approach provides ease of programmability for the application programmer as no changes to the application code, or the API are required. Thus, all applications that use the PGAS inter-process communication will implicitly benefit. This is contrary to many other approaches which employ their accelerators directly on the application level.

#### 5.5.2.4 Summary

The proposed near-memory graph copy and near-cache graph writeback accelerator integrated into the system software enable the efficient transfer of object graphs on tile-based many-core architectures. The architecture awareness of the system software is increased to reflect the high importance of data-to-task locality better. NEMESYS provides near-memory acceleration to mitigate PGAS inter-process communication challenges between tiles.

### 5.5.3 Concept of the Design Alternative Near-Memory Core

The previous section described the concept of the near-memory hardware acceleration approach (NEMESYS). Nevertheless, the design space contains a design alternative (cf. Section 5.5.1). As depicted in Figure 5.6b, a near-memory core (NMCORE) is also conceivable to perform the graph copy operation near-memory. It is a mixture of the NEMESYS and PEGASUS approaches. This section briefly elaborates on the commonalities and differences. In Section 5.7.4, also a quantitative evaluation will be performed.

The NMCORE approach likewise benefits from its near-memory integration. Fundamentally, the entire Section 5.5.2.1 applies as is. Theoretically, the NCA could also be part of the NMCORE concept. However, since the NMCORE is not a hardware accelerator, it will not be combined with the NCA. The graph copy operation is outsourced to the NMCORE – which is integrated into the memory tile close to the memory – via a remote function call. The benefits of hardware acceleration do not apply. However, the graph copy software algorithm used by the PEGASUS approach can be employed with only minor adjustments. Since the NMCORE is located in the memory tile instead of tile $D$, it cannot allocate memory for new objects of $G'$ on-the-fly as PEGASUS does. However, it is possible to use the same pre-allocation mechanism leveraged for the NMA variant. This will be explained in Section 5.6.1 in more detail.

Similar to the NEMESYS approach which has one NMA per memory tile, also only one NMCORE exists per memory tile. In contrast, the PEGASUS approach can perform several independent graph copies on separate tiles. It is essential to analyze whether the near-memory units can cope with the graph copy workload outsourced to them by many cores.

## 5.6 Architecture and Implementation

The previous section presented the main features of the NEMESYS concept. Now, several architectural and implementation details of the design are described. First, the hardware-software co-design approach (i.e., the interplay between the run-time system and the proposed hardware units) is explained in Section 5.6.1. Moreover, Section 5.6.2 elaborates on the requirements and fundamentals of copying object graphs in hardware. Then, Section 5.6.3 presents the details of the proposed hardware units. Finally, Section 5.6.4 describes the implementation details of the NMCORE design alternative.

### 5.6.1 Hardware-Software Co-Design

This section presents the interplay between the run-time system and the proposed hardware units. Section 5.6.1.1 describes the necessary steps of the NEMESYS inter-process communication mechanism when copying object graphs between memory partitions residing in the same physical memory, as depicted in Figure 5.7. Then, in Section 5.6.1.2, a slight adaptation of the mechanism is illustrated to handle inter-memory graph copies efficiently.

#### 5.6.1.1 Intra-Memory Graph Copy

The fundamental mechanism of the PGAS inter-process communication leveraging the NEMESYS approach has been described in Section 5.5.2 alongside Figure 5.7. Identical to the PEGASUS approach, the graph $G$ needs to be transferred from source place $S$ to destination place $D$. As before, additional metadata information $M$ needs to be passed between these places. It contains, e.g., the pointer $G$ to the closure, the function $func$ to be executed on the remote place $D$, or synchronization objects to notify the completion. Additionally, the graph size of $G$ needs to be measured and appended to the metadata, which becomes relevant for the pre-allocation of the destination graph $G'$.

The required steps are now described in more detail, with a particular focus on the interplay between hardware units and the run-time system. The corresponding message sequence chart, depicted in Figure 5.8, is extended by the NCA and the NMA. Furthermore, Algorithm 1 shows the corresponding pseudo source code of the `AT statement` executed on place $S$, while Algorithm 2 and Algorithm 3 show the corresponding pseudo source code executed on place $D$, before and after the graph copy operation, respectively.

(1a) The sender $S$ remotely allocates the metadata buffer $M'$ on $D$.

(1b) The sender $S$ allocates and sets up a metadata structure $M$ that needs to be transferred to the receiver $D$ (into $M'$).

(1c) The $\texttt{NCA}_S$ is triggered on tile $S$ asynchronously, passing the descriptor of task $T_1$ as well. $S$ then waits for the completion of the remaining parts of the at statement in a non-blocking manner.

(1d) The source graph $G$ is written back into its memory partition by the $\texttt{NCA}_S$, which additionally measures the graph size of $G$.

(1e) The measured graph size is appended to the metadata structure $M$ by the $\texttt{NCA}_S$.

(2) Without a software up-call, the $\texttt{NCA}_S$ triggers a DMA of the metadata $M$ to $M'$. The DMA engine subsequently invokes the task $T_1$ on a core on $D$ via the hardware scheduler.

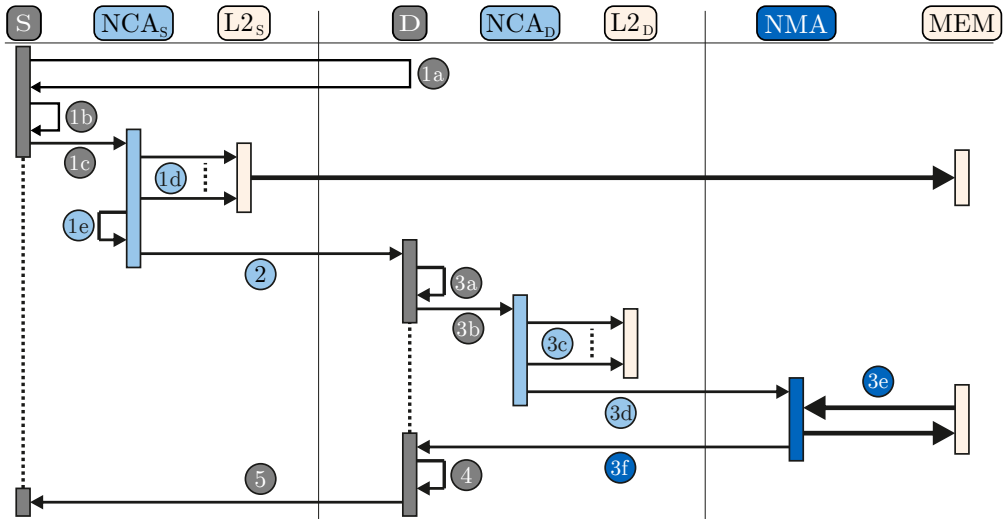



**Figure 5.8: Message sequence chart of the NEMESYS inter-process communication.** The NMA and NCA are incorporated in the hardware-software co-design approach. The NCA$_S$ performs the graph writeback in step (1d), appends the graph size to the metadata $M$ (1e), and initiates the DMA to tile $D$ (2). The NCA$_D$ performs the invalidation of $G'$ (3c) and triggers the NMA (3d). The NMA performs the near-memory graph copy (3e) and signals the completion to tile $D$ (3f). The thick arrow indicates the remote memory accesses of the graph writeback (1d).

Then, on the destination place $D$, the task $T_1$ is executed:

(3a) A core on $D$ reads the metadata information from $M'$ and, based on this information, pre-allocates a destination buffer for the to-be-copied graph $G'$ in the memory partition of $D$ and invalidates $G'$ in its L1 cache.

(3b) Next, NCA$_D$ on tile $D$ is triggered asynchronously. Additionally, the parameters to subsequently trigger the NMA are passed to the NCA$_D$. These also contain the descriptor of the task $T_2$ scheduled on tile $D$ upon completion of the NMA.

(3c) The NCA$_D$ invalidates the destination buffer allocated for $G'$ to avoid interfering cache evictions during the graph copy operation.

(3d) Upon completion of the invalidation, the NMA is directly triggered by the NCA$_D$.

(3e) The graph $G$ is copied to $G'$ by the NMA.

(3f) Upon completion, the NMA spawns the task $T_2$ onto a core on $D$ via the hardware scheduler.

(4) The task $T_2$ is then scheduled on a core on $D$ which reads necessary metadata information from $M'$, invalidates $G'$ in its L1 cache, and executes the function `func` on the copied graph $G'$.

(5) Finally, the termination of the inter-process communication mechanism is signaled back to $S$. If a result graph was generated in step (4), the same mechanism is applied to copy the result to place $S$. Thereby, the metadata structures $M$ and $M'$ can be reused.

Since the operations of the NMA are not visible to the cores, the memory accesses are not part of the tile-local snooping-based L1 cache coherence protocol. Therefore, the cache

**Algorithm 1** Pseudo code of the AT Statement on source place $S$. This code corresponds to steps (1a) – (1c)

```
1: function AT_STATEMENT(graph G, place D, function func)
2:     M' = remote_allocation(sizeof(M), place D);              ▷ (1a)
3:     M = allocate_and_setup(sizeof(M), G, func);              ▷ (1b)
4:     trigger_NCA_writeback_dispatch(G, M, M', task_T1);       ▷ (1c)
5:     result = wait_for_completion();                          ▷ (1c)
6:     .
7:     .
8:     return result;
```

**Algorithm 2** Pseudo code of task $T_1$ executed on the destination place $D$ before the graph copy. This code corresponds to steps (3a) – (3b)

```
9: function TASK_T1(metadata M', function func)
10:    read_metadata(M');                                       ▷ (3a)
11:    G' = mem_allocate(size(G));                              ▷ (3a)
12:    invalidate_L1_cache(G');                                 ▷ (3a)
13:    trigger_NCA_invalidate_and_trigger_NMA(G, G', task_T2);  ▷ (3b)
```

**Algorithm 3** Pseudo code of task $T_2$ executed on the destination place $D$ after the graph copy. This code corresponds to steps (4) – (5)

```
14: function TASK_T2(metadata M', function func)
15:    read_metadata(M');                                       ▷ (4)
16:    invalidate_L1_cache(G');                                 ▷ (4)
17:    result = func(G');                                       ▷ (4)
18:    notify_completion(place S, result);                      ▷ (5)
```

lines need to be explicitly invalidated in step (3a) to avoid interfering evictions of the L1 cache and before performing step (4) to ensure a clean view on the copied graph $G'$.

Since the graph writeback and graph copy operations are offloaded asynchronously to the NCAs and NMA, respectively, the software involvement in the NEMESYS inter-process communication between tiles is minimal. Between steps (1c) and (5), the cores on $S$ are not involved in the mechanism. Similarly, the cores on $D$ are solely active during steps (3a), (3b), and (4).

#### 5.6.1.2 Advanced Inter-Memory Graph Copy

The previous section described the fundamental mechanism of the NEMESYS inter-process communication between tiles. It works very efficiently when the memory partitions of $S$ and $D$ reside in the same physical memory. However, when several physically distributed memories are present, the involved partitions are likely [3] on different memory tiles. In this case, copying the graph $G$ directly *near-memory* is not possible since the NMA could only

[3] Compute tiles 1 to 7 have their memory partition in the first memory tile while compute tiles 8 to 14 have their partitions in the second memory tile. Thus, when communicating between the first and second set of compute tiles, the respective memory partitions are on different memory tiles.

access the copied graph $G'$ via high-latency remote load-store operations over the NoC. [4] This is diminishing the near-memory aspect.

Therefore, copying object graphs efficiently between physically distributed memories is also necessary. This section presents an advanced mechanism for *inter-memory* graph copy. It leverages an intermediate buffer $G^*$, which is additionally allocated in the source partition $S$, residing in the same physical memory as the original graph $G$. A two-step procedure is required to copy the graph $G$ via the intermediate buffer $G^*$. These steps are illustrated

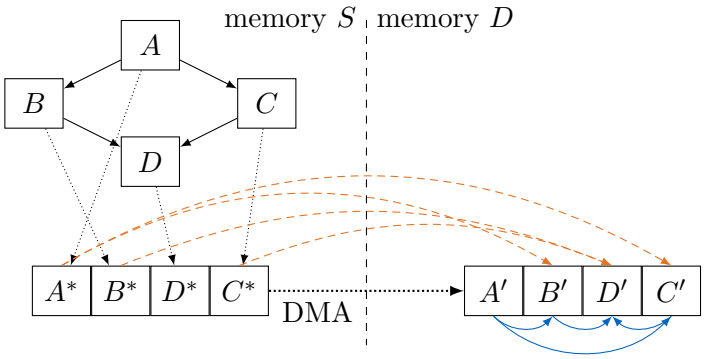



**Figure 5.9: Advanced inter-memory graph copy mechanism.** The source graph $G$ (in memory tile $S$) is copied via the intermediate buffer $G^*$ (in memory tile $S$) to the destination graph $G'$ (in memory tile $D$). The `NMA` copies $G$ into $G^*$ near-memory inside memory tile $S$ and directly uses the final pointers (dashed orange). Then, the `NMA` initiates a DMA transfer of $G^*$ to $G'$ resulting in the blue pointers.

in Figure 5.9: (1) $G$ is copied into the intermediate buffer $G^*$. Thereby, all pointers are adapted to directly point (dashed, orange) to the corresponding object in the destination graph $G'$. This is done near-memory by the `NMA` in the memory tile of the source graph $G$. (2) The intermediate flattened graph $G^*$ is transferred via DMA to $G'$.

This mechanism does not require post-processing or deserialization on the receiver side. The `NMA` can set the pointers in the intermediate graph $G^*$ (orange dashed arrows) so that they already point to the respective objects in the destination graph $G'$. Therefore, after simply copying $G^*$ to $G'$ via DMA, all pointers match the objects in $G'$ (blue arrows). The `NMA` itself initiates the DMA asynchronously upon completion of its graph copy operation. Thus, it can already process the next graph copy request waiting in its FIFO queue. Upon completion of the inter-memory DMA transfers, the DMA engine spawns the task $\mathrm{T}_2$ on a core on the receiver tile $D$.

A slight modification of the hardware-software co-design mechanism presented above is required. Changes (marked in **bold**) on software side occur for steps (1c) – (2):

**(1c')** The $\texttt{NCA}_S$ is **now** triggered with an **additional parameter**, namely the descriptor of another task $\mathbf{T}_0$, whose pseudo code is given in Algorithm 4.

(1d) The source graph $G$ is written back into its memory partition by the $\texttt{NCA}_S$, which additionally measures the graph size of $G$.

(1e) The measured graph size is appended to the metadata structure $M$ by the $\texttt{NCA}_S$.

**(1f') The $\texttt{NCA}_S$ spawns the task $\mathbf{T}_0$ on a local core on $S$ itself.**

[4] It is assumed that always the `NMA` in the memory tile corresponding to the source place $S$ is used.

**(1g') The task $T_0$ is executed locally on $S$. $S$ reads necessary metadata information from $M$. The intermediate buffer $G^*$ is allocated and invalidated, before a reference to it is appended to the metadata structure $M$.**

**(2') Then, the core on** $S$ transfers the metadata structure $M$ to $M'$ via DMA and subsequently invokes the task $T_1$ on a core on place $D$.

**Algorithm 4** Pseudo code of task $T_0$ executed on the source place $S$ before the signaling of the receiver. This code corresponds to steps (1g') – (2')

```
19: function TASK_T0(metadata M, task_T1)
20:     read_metadata(M);                                          ▷ (1g')
21:     G* = mem_allocate(size(G));                                ▷ (1g')
22:     invalidate_L1_cache(G*);                                   ▷ (1g')
23:     append_metadata(M, G*);                                    ▷ (1g')
24:     transfer_metadata_and_call_remote_function(M, M', task_T1);  ▷ (2')
```

In Section 5.7.3.5, the performance overhead of this approach will be evaluated.

## 5.6.2 Fundamentals of Hardware Graph Copy and Writeback

The previous section presented the interplay between the run-time system and the hardware units. Before the next section describes the implementation details of the respective hardware units, this section elaborates on the fundamentals of hardware graph copy and writeback. The hardware accelerators must be capable of handling arbitrary object graphs without intermediate software interaction. This requires the ability to traverse an object graph, access an object's run-time type information (*RTTI*), know how to copy/writeback a particular object type, store transient state information, and allocate memory for new objects. Therefore, Section 5.6.2.1 presents the underlying object model and its necessary extensions. Then, the mechanisms and required auxiliary data structures of hardware graph copy and writeback are described in Section 5.6.2.2 and 5.6.2.3, respectively.

### 5.6.2.1 Object Model

Before the required extensions to the object model are presented, the existing object model of the X10 run-time system is described.

**Existing Object Model.** The underlying object model is compatible with Java-like object-oriented programming languages and consists of a specific *header* and varying *payload*. Figure 5.10 depicts an exemplary object layout. The object header comprises a *vptr* (virtual pointer) pointing to the *vtable* (virtual table), which in turn contains pointers to the class's virtually bound methods and a pointer to the class's *RTTI* structure. The *RTTI* contains, among other information, the object's size and *pointer masks* which are, for instance, required for the graph traversal. The pointer masks are metadata information about the object's memory layout, like the offset where references to other objects reside in the payload. Similar metadata is used in other run-time systems (e.g., the Go run-time [190] or HotSpot VM [191]) to enable garbage collection. However, in this use case, the *RTTI* is leveraged to enable the hardware graph traversal, copy, and writeback of arbitrary objects.

The object payload can be classified into five groups, as presented in Table 5.1: For each payload type, a distinctive pointer mask exists, of which always two bits correspond to the

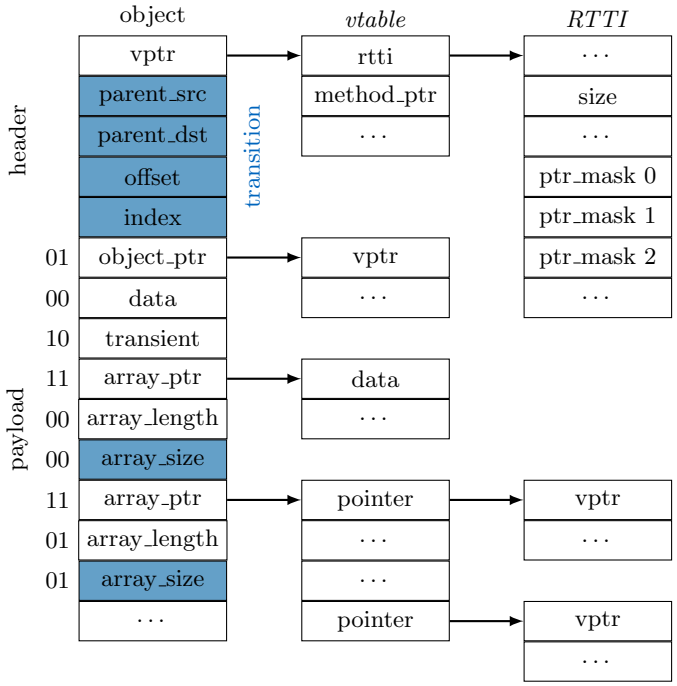



**Figure 5.10: Exemplary object layout and object model extensions.** The object model comprises a specific header and varying payload. Additional members (i.e., the transition structure and the `array_size` field) required by the `NEMESYS` accelerators are marked in blue.

individual payload words (32 bits) of an object. Variable object sizes are allowed by locating these pointer masks at the end of the *RTTI*. For illustration purposes in Figure 5.10, these pointer masks are additionally annotated next to the object payload. For primitive data (`00`), pointers to other objects (`01`), and transient [5] words (either pointer or data) (`10`) two-bit pointer masks are sufficient. In contrast, an array is described by an *array descriptor* which consists of two words in the existing object model and three words in the necessary object model extension. In both cases, the first word (`array_ptr`) is marked with `11` and is pointing to the array's *backing store* containing its actual data. The second word (`array_length`) denotes the number of array elements with the same type. In the X10 run-time, these arrays can either be primitive arrays (i.e., arrays of primitive data such as strings) or arrays of pointers to other objects. To distinguish between these two types of arrays, the pointer mask, defined in Table 5.1, also stretches over the second (and third) word of the array descriptor, as depicted in Figure 5.10.

**Object Model Extensions.** In order to support hardware graph copy and writeback, two extensions to the object model are required. First, the array descriptor has to be extended by the **array size information** as depicted in Figure 5.10. Note that the array size differs from the object size since the object size only incorporates the array descriptor and not the array's backing store. In software, the compiler would use static type information to determine this size. However, the hardware accelerator is unknowing of this information. Therefore, it needs to be added to the object model explicitly. In the case of an array of pointers, the `array_size` would be `array_length` times four Byte, assuming a pointer size

[5] A word is designated as transient if it is not required to be copied. Instead, it needs to be set to zero.

**Table 5.1: Classification and description of object payload types.** The object payload can be classified into five groups. Each type has a distinctive pointer mask. The blue extension to the array pointer masks represents the `array_size` field.

| payload type | description | pointer mask |
|---|---|---|
| data | primitive data | 00 |
| pointer | pointer to another object | 01 |
| transient | a word that must not be copied but set to zero | 10 |
| primitive array | pointer to an array of primitive data (e.g., strings) | 11 00 00 |
| array of pointer | pointer to an array of pointers to other objects | 11 01 01 |

of 32 Bit. However, in the case of a primitive array, the `array_size` denotes the size of the array's backing store, dependent on the primitive data type.

Second, the **transition structure** is added to the object header, residing in normal memory, as illustrated in Figure 5.10. It is required as both the hardware graph copy and writeback algorithm require a place to store their state information of the graph traversal. Since it is required to handle arbitrarily sized and structured object graphs, it is impossible to store the information in dedicated hardware resources. The necessary state information is: (1) `parent_src`: the pointer to the parent object. It is used to retreat to the predecessor object via pointer reversal. (2) `parent_dst`: the pointer to the corresponding clone of the parent object. It is stored to avoid a superfluous copy map lookup. (3) `offset`: the position in the object payload up to which the copying of the object has already been completed. (4) `index`: the array index up to which the cloning of an array of pointers has been processed. The `offset` and `index` fields are used to continue the graph copy process at the proper payload or array index after retreating to a predecessor object. The root object is identified by setting its `parent_src` field to NULL. If the current object is the root and the `offset` has reached the end of the object payload, the graph copy is complete.

During the graph copy, the transition structures of the objects in the destination graph $G'$ ($G^*$ in case of the inter-memory graph copy) must be used for several reasons: (1) It is possible that the same graph $G$ or a subgraph of it is concurrently copied by another thread to a different destination graph. In this case, the graph copy associated with the first thread is written back in parallel by another thread to prepare the second graph copy. Thus, the transition structures of $G$ would be overwritten by the graph writeback during step (1d). (2) Since $G$ still resides in the L2 cache, an eviction by another cache line, which aliases with it, would overwrite the transition structures in $G$ as well. This may happen, even for evictions caused by an unrelated thread. (3) In contrast, the objects of $G'$ (or $G^*$) reside in a newly allocated and invalidated buffer, which is neither cached in any cache nor allowed to be accessed by any thread yet. Thus, the transition structures of $G'$ (or $G^*$) are safe to utilize.

#### 5.6.2.2 Requirements of Hardware Graph Copy

Copying an object graph requires a traversal of the entire graph to reach and clone [6] all objects which are in general dispersed in memory. The proposed algorithm uses a depth-first search (DFS) and is based on the idea of pointer reversal introduced by Schorr and Waite [189]. Therefore, the object's *RTTI* is accessed to look up the object's layout. Then, the object is partially cloned until a reference to another object is reached. This reference

[6] Note that in this thesis, the terms *graph copy* and *graph cloning* are used as synonyms.

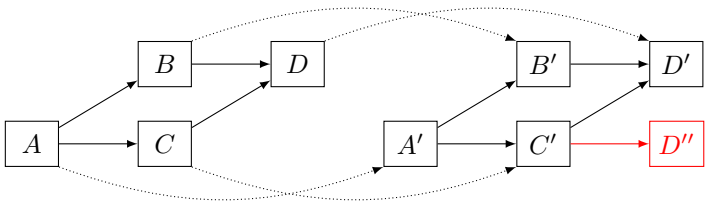



**Figure 5.11: Exemplary graph to illustrate graph copy and writeback algorithm.** Object $D$ has to parent objects. The graph copy algorithm has to ensure that only one copy $D'$ of $D$ is made and not the additional $D''$ marked in red. Thus, the auxiliary copy map is required.

is followed, and the respective child object [7] is cloned before retreating to the *parent* object. To illustrate this mechanism, the example graph, shown in Figure 5.11, is assumed.

The graph copy is started by cloning the root object $A$, containing two references, one to $B$ and one to $C$. When reaching the first pointer in the object's payload (i.e., to $B$), the cloning of object $A$ is paused, and the algorithm advances to the next object $B$. During the processing of object $B$, the further algorithm advances to $D$ before it finishes $B$ and subsequently retreats to $A$ and continues its copy process. Since $A$ also points to $C$, $C$ is cloned afterward, and after retreating from $C$ to $A$, the clone of $A$ is finalized and thus the entire graph. To summarize the depth-first search approach: if an object has multiple references, its children are finished prior to the object itself.

*Advancing* to the next object and *retreating* to an object previously visited are the two main operations of the graph traversal. Usually, a recursion stack would be leveraged to support these operations in software. However, to avoid using a separate recursion stack in hardware, the transition fields of the extended object header are used to store the required state information.

As depicted in Figure 5.11, an object graph may have cycles or several references to the same object. Besides handling the recursion mentioned above, the hardware accelerator must detect these cycles and identify objects already visited. Besides, the pointer to the already existing copy of the object needs to be retrieved. Continuing the previous example, where copying of the objects $A$, $B$, and $D$ has already been started, the algorithm must detect that $C$ is pointing to $D$ as well. No new copy of $D$ to $D''$ may be made; instead, the already existing reference to $D'$ must be used. This requires an additional auxiliary data structure, the so-called *copy map*, which associates each object with its copy. Before creating a new object $D'$, the algorithm searches the copy map for an entry $D$. If it is found, $D'$ is returned and used. Otherwise, a new object $D'$ needs to be created, and the mapping $D \mapsto D'$ is appended to the copy map. The implementation of the copy map is explained in more detail below.

Finally, when creating new objects, the respective memory has to be allocated. The graph copy unit avoids costly intermediate up-calls to the operating system to use the dynamic memory allocator using an internal linear bump allocator, which operates on a preallocated memory range provided to it when triggered. The details will be explained below.

**Copy Map.** The copy map is an auxiliary data structure required by the graph copy hardware accelerator to detect and handle cyclic graphs. It is statically allocated in every memory tile by the run-time system with adequate size for the particular application, and

[7] In a directed graph, $B$ is a child node of the parent node $A$ if $A$ has a pointer to $B$

a pointer to it is passed to the graph copy accelerator when triggered. When cloning a graph, the copy map is used to store and lookup the mapping of an object and its clone (e.g., $A \mapsto A'$). Therefore, the copy map is a buffer divided into two halves, one half for the pointer to the original object $A$, and the second half for the reference to its clone $A'$. $A$ and $A'$ are stored at the same offset in the copy map's respective halves. The exact offset depends on the copy map's actual implementation, which exists in two variants: either based on linear search or implemented as a hash map. Although the hash map implementation has a higher initial setup time, its scaling behavior for a graph with many objects is better. Since there is a trade-off between both, the choice for one can be made dynamically at run-time for every accelerator call. In the following, both variants are explained.

The **implementation via linear search** keeps the pointers to the original objects $A$ at consecutive offsets in the first half of the copy map. Inserting a new object reference simply requires incrementing the current offset by one position. However, to look up if an object already exists takes a brute force search over the entire range of the occupied slots in the first half of the copy map. Thus, the scaling behavior for a graph with $o$ objects is $\mathcal{O}(o^2)$ because for every new object, the copy map has to be searched.

As the linear search can degrade the performance for graphs with many objects, the copy map's second variant leverages the idea of hashing. In the **implementation as hash map**, the offset at which $A$ and $A'$ are inserted in the first and second half of the copy map, respectively, is determined by hash($A$). In the best case, the offset is found immediately. In case hashing collisions occur, linear probing is used to resolve them. The implementation is based on the family of universal hash functions $H_3$ introduced by Carter [192].

Analyzing the IMSuite benchmarks employed for the evaluation revealed that the number of objects per graph is relatively small. Since the linear search implementation of the copy map performs sufficiently well, hashing will not be activated for the evaluation in this thesis. The interested reader is referred to the previously published article [3] for further implementation details and evaluation.

**Bump Allocator.** The graph copy accelerator is equipped with an internal linear bump allocator so that no intermediate up-calls to the operating system for dynamic memory allocation of new objects are required. The bump allocator operates on a preallocated buffer $G'$ in memory, which the run-time system has allocated in step (3a) (cf. Section 5.6.1.1). The buffer's size is calculated as the measured graph size of $G$ plus additional bytes for cache line alignment. The `NCA` measures the graph size during graph writeback. When triggered, the pointer to the preallocated buffer is passed to the graph copy accelerator. Then, the internal hardware bump allocator allocates a new object by solely incrementing the *next address* by the new object's size. This causes all objects to be stored consecutively in the allocated memory space. However, this should not be confounded with serialization.

In the current implementation, the support for garbage collection is limited as no additional allocator information is added before the individual objects by the bump allocator. However, this can be easily extended in future work. The bump allocator already leaves the two words before the new objects blank so that the software allocator can insert the required allocator information afterward, i.e., after the completion of the graph copy. Another possibility would be not only to provide one big buffer to the graph copy accelerator but a list of individual small buffers according to the object sizes measured during the graph writeback.

#### 5.6.2.3 Requirements of Hardware Graph Writeback

Writing back an object graph from cache to memory requires a complete graph traversal. Writeback commands for all cache lines of each object have to be issued. These are sent to the cache controller by the near-cache accelerator. The cache controller either executes the individual cache lines' writeback if they are in a modified state or returns instantly.

In contrast to the depth-first-search of the graph copy mechanism, the graph writeback algorithm is better off with a breadth-first-search (BFS) because no pointer reversal approach is required. Instead, the near-cache accelerator first issues a writeback command for the entire object based on the object's size information obtained from the *RTTI*. Whenever a pointer is found in an object, it is pushed to the so-called *visit stack*. The *visit stack* is another auxiliary data structure that is statically allocated in memory per tile with adequate size for the particular application. After completing the previous object's writeback, the `NCA` continues with the next object popped from the visit stack. This sequence is repeated until the stack remains empty.

Additionally, cyclic graphs are handled differently than in the case of graph copy. Since there is no need to create and lookup mappings in the copy map, the `NCA` directly uses the processed object's transition structure to store the information if the object has already been visited. The marker in the transition structure is checked before writing back the object and set when the object is processed.

For the example graph in Figure 5.11, this works as follows. $A$ is written back, marked as done, and the pointers to $B$ and $C$ are pushed onto the *visit stack*. As a stack operates according to the last-in-first-out principle, $C$ is popped from the visit stack next, written back, and marked as done. This is followed by a push of pointer $D$ to the visit stack. Then, $D$ is popped, written back, and marked as done. Finally, the visit stack is empty after $B$ is popped, written back, and marked as done. As $B$ points to $D$ as well, $D$ would have been processed again if the marker in the transition structure of $D$ would not be set.

**Visit Stack.** The visit stack is an auxiliary data structure required by the graph writeback accelerator only. It is statically allocated in the memory partition of every compute tile by the run-time system with adequate size for the particular application. It is used to keep track of all outstanding objects to be written back. When a reference to another object is found during the breadth-first-search, it is pushed to the visit stack. The visit stack is a simple linear buffer with a stack pointer. Since only the respective `NCA` of the tile is allowed to access its visit stack during the graph writeback, the stack pointer can easily be incremented and decremented by the `NCA` without the need for atomic operations.

#### 5.6.2.4 Summary.

In conclusion, both hardware graph copy and writeback access the object's *RTTI* to extract the required object and graph information. Then, the hardware graph copy mechanism traverses the graph with a depth-first-search approach based on the idea of pointer reversal. Therefore, the object model was extended by the transition structure, and an internal bump allocator is necessary. Moreover, the auxiliary copy map data structure is required to handle cyclic graphs.

In contrast, the hardware graph writeback algorithm uses a breadth-first-search mechanism to traverse the graph. Therefore, it requires the auxiliary visit stack data structure, which tracks still-to-be-handled objects. Furthermore, the transition structure in the object header, not the copy map, is used to mark objects already visited to detect cycles.

### 5.6.3 Graph Accelerator Hardware Units

While the previous section elaborated on the fundamental requirements of hardware graph copy and writeback, this section presents the implementation details of the respective near-cache and near-memory hardware units. It includes the NCA (cf. Section 5.6.3.1) with its submodules NCA-RO (range operations unit) and NCA-WB (graph writeback and dispatch unit), as well as the NMA graph copy unit (cf. Section 5.6.3.2).

#### 5.6.3.1 The Near-Cache Accelerator

The near-cache accelerator is integrated into each compute tile as depicted on the left side of Figure 5.12. It is placed near the cache controller on a dedicated L2 front-end bus to enable faster access and reduce the load on the main bus. The modular design is independent of the particular cache controller implementation and uses the same cache-control commands as the CPUs.

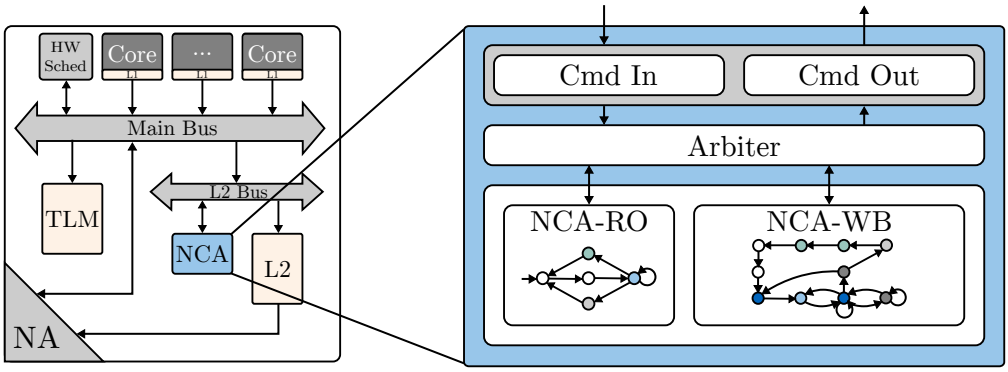



**Figure 5.12: Integration and architecture of the near-cache accelerator (NCA).** Overview of a compute tile including the NCA (left) and block diagram of the internal structure of the NCA (right). The NCA-RO performs range based cache operations and is able to subsequently trigger the NMA. The NCA-WB performs the graph writeback, appends the graph size to the metadata $M$, and initiates the DMA to tile $D$ via the bus and network adapter. The detailed functionality of the depicted FSMs are found in Appendix A and Appendix B, respectively.

The internal structure of the NCA is depicted on the right side of Figure 5.12 as a block diagram. Each NCA consists of two submodules: the range operations unit (NCA-RO) and the graph writeback unit (NCA-WB). The functionality of these submodules is described in the following paragraphs. The NCA further contains a bus interface with a CMD In interface to receive incoming commands and status checks from cores and a CMD Out interface to send out commands to the cache controller to initiate cache operations. Moreover, an arbiter is responsible for forwarding the incoming requests to the respective submodule and handling access to the CMD Out interface.

**The Near-Cache Range Operations Unit.** The range operations unit (NCA-RO) can perform cache operations (i.e., invalidation, writeback, or flush commands) on a contiguous address range. Instead of occupying the CPU with looping over the address range and issuing the respective cache operation for each cache line, this potentially long-lasting task is outsourced to the range operations unit. A similar unit is already present in the PEGASUS system [169]. However, it was integrated into the L1 cache instead of near the L2

cache on which it is operating. Furthermore, in contrast to the prior L1 cache range unit, the NCA-RO unit is equipped with the capability to trigger the NMA with user-defined parameters. This is leveraged to minimize the hardware-software interaction as the range invalidation of the destination buffer $G'$ is directly followed by sending a trigger to the NMA via the bus and network adapter (NA). Therefore, the parameters for the NMA are also passed to the NCA when called asynchronously. This extended sub-functionality is referred to as *invalidate-and-trigger*.

In conclusion, the NCA-RO unit relieves the CPUs of long-lasting cache range operations. Although it features a relatively simple functionality, it serves well as it is placed at the proper position near the cache. The NCA-RO's full functionality is described in Appendix A with the help of a finite state machine diagram (FSM).

**The Near-Cache Graph Writeback Unit.** The second sub-module of the NCA is already more complex than the NCA-RO. The near-cache graph writeback accelerator (NCA-WB) can traverse an arbitrary graph and issue writeback operations for all cache lines touched by every object of the graph.

To enable asynchronous offloading of the graph writeback operations, the NCA-WB can update the metadata $M$ in memory with the object count and graph size information, initiate the DMA transfer of the metadata $M$ to $M'$ and dispatch the corresponding task $T_1$ to the receiver $D$ via the bus and network adapter (NA). This extended sub-functionality is referred to as *writeback-and-dispatch*. It frees the core on $S$ to perform other tasks instead of waiting for the completion of the graph writeback.

The NCA-WB accelerator can traverse and writeback arbitrary object graphs without intermediate software involvement. Its detailed functionality is described in Appendix B with the help of an FSM diagram.

#### 5.6.3.2 The Near-Memory Graph-Copy Accelerator

While the NCA is present in every compute tile near the L2 cache, the NMA is integrated into each memory tile as depicted on the left side of Figure 5.13. It is placed near the memory controller on a dedicated memory bus to enable faster access and reduce the load on the main bus. The modular design is independent of the particular memory controller implementation.

The internal structure of the NMA is illustrated on the right side of Figure 5.13 as a block diagram. The NMA contains a slightly different bus interface since the NMA-GC does not receive its commands via the bus but from the FIFO, which can buffer 16 incoming requests and is filled via the network adapter (NA). The Config In bus interface only allows access to the NMA-internal performance counters and to configure the hash map matrix. The CMD Out interface is used to issue memory reads and writes to the memory via the bus.

The incoming command from the FIFO contains several parameters: (1) the pointer to the source graph $G$, (2) the pointer to the preallocated destination buffer $G'$, (3) *the pointer to the preallocated intermediate buffer $G^*$ (only in case of the advanced inter-memory graph copy*[8]*)*, (4) the pointer to the statically allocated copy map, (5) the task descriptor of the completion task, which is sent to the receiver tile upon completion of the graph copy, and (6) *the copy map length (only if the hash map variant is used).*

[8] Note that both the standard intra-memory graph copy from $G$ directly to $G'$ and the advanced inter-memory graph copy from $G$ via $G^*$ to $G'$ use the same FSM.

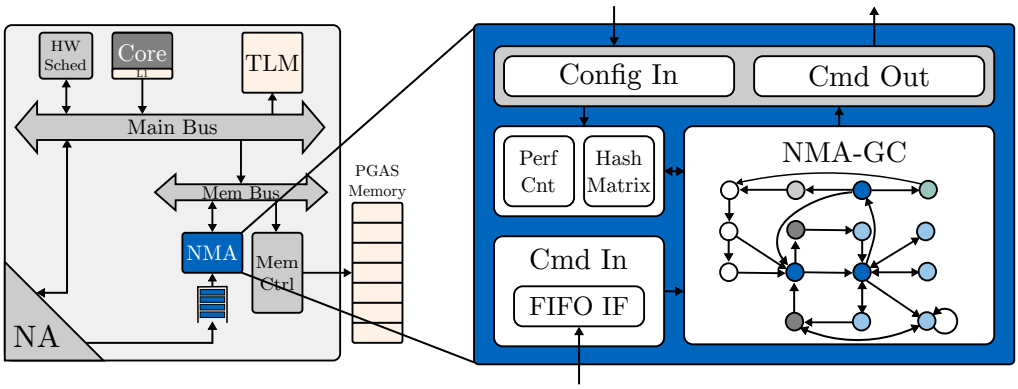



**Figure 5.13: Integration and architecture of the near-memory accelerator (`NMA`).** Overview of a memory tile including the `NMA` (left) and block diagram of the internal structure of the `NMA` (right). The `NMA` receives its command via the FIFO which is fed by the network adapter (NA). The `NMA-GC` performs the graph copy operations and is able to subsequently send the completion task to tile $D$ or, in case of the advanced inter-memory graph copy, initiate the DMA of $G^*$ to $G'$. The detailed functionality of the depicted FSM is found in Appendix C.

The `NMA` retrieves all information about the graph $G$ and its objects from the objects' *RTTI* residing in normal memory. The detailed functionality of the graph copy module (`NMA-GC`) is explained in Appendix C with the help of a finite state machine diagram. Upon completion of a normal intra-memory graph copy, the `NMA` sends the completion task descriptor `task_T1` asynchronously to a core on the destination tile via the network adapter and hardware scheduler. In the case of the advanced inter-memory graph copy, the `NMA` cloned the graph $G$ to the intermediate buffer $G^*$ so far. The finalization of the graph copy to $G'$ requires an asynchronous DMA transfer of $G^*$ to $G'$. The DMA trigger is sent to the network adapter accompanied by the descriptor of `task_T1` which will be scheduled on the destination tile upon completion of the DMA. In either case, the `NMA` can directly handle the subsequent graph copy request from its FIFO.

### 5.6.4 Implementing the Design Alternative Near-Memory Core

This section elaborates on the near-memory core (`NMCORE`) design alternative after having detailed the `NMA`'s and `NCA`'s design in the previous sections. As described in Section 5.5.3, the `NMCORE` design alternative of the PGAS inter-process communication between tiles is a mixture of the related `PEGASUS` and the proposed `NEMESYS` approach. Identical to both approaches, the graph $G$ needs to be transferred from source place $S$ to destination place $D$. As before, additional metadata information $M$ needs to be passed between these places. It contains e.g., the pointer $G$ to the closure, the function $func$ to be executed on the remote place $D$, or synchronization objects to notify the completion. Additionally, the graph size of $G$ needs to be measured and appended to the metadata, which becomes relevant for the pre-allocation of the destination graph $G'$.

The required steps are now described in more detail. The corresponding message sequence chart, depicted in Figure 5.14, is extended by the `NMCORE`. The initial steps of the `AT statement` are almost identical to the `PEGASUS` variant (Changes marked **bold**).

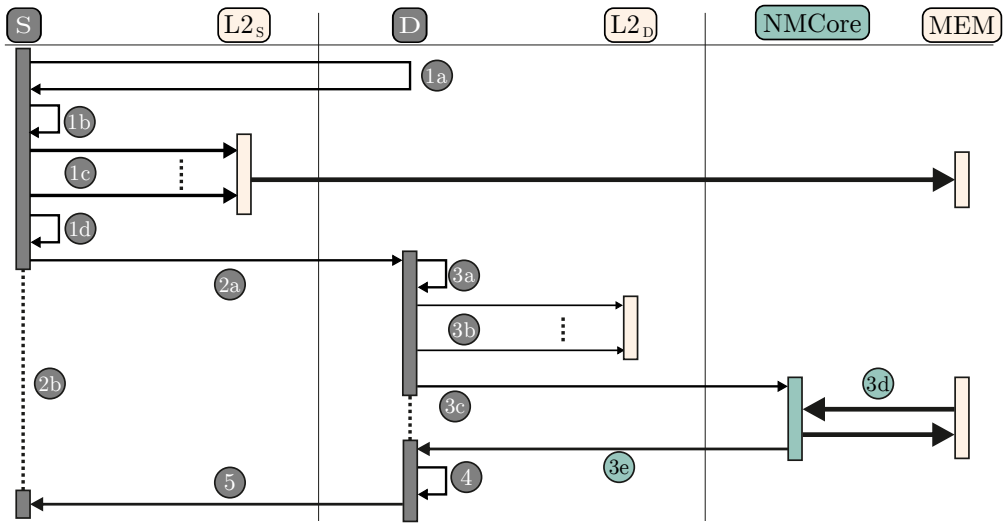



**Figure 5.14: Message sequence chart of the `NMCORE` inter-process communication.** The steps (1a) until (3a) are identical to the `PEGASUS` variant. Until here, only step (1d) is added. Then, $G'$ has to be invalidated (3b) before the `NMCORE` is triggered (3c) which performs the near-memory graph copy (3d) and signals its completion (3e). The thick arrow indicates the remote memory accesses of the graph writeback (1c) which does not use the `NCA`.

(1a) The sender $S$ performs a remote allocation of the metadata buffer $M'$ on $D$.

(1b) The sender $S$ allocates and sets up a metadata structure $M$ that needs to be transferred via DMA to the receiver $D$ (into $M'$).

(1c) $S$ synchronously writes back the source graph $G$ into its memory partition.

**(1d) $S$ measures the graph size and appends it to the metadata structure $M$.**

(2a) $S$ initiates a DMA of the metadata $M$ to $M'$ with subsequent invocation of the `remote_task` on $D$.

(2b) $S$ waits for the completion of the remote part of the `AT statement` on $D$.

Then, on place $D$, the steps executed before the graph copy operation are similar to the `NMA` variant, including the pre-allocation of $G'$. However, it is slightly different since the `NMCORE` instead of the `NMA` has to be triggered, and the invalidation of the L2 cache is not outsourced to the `NCA`. The steps performed are:

(3a) On the destination place $D$, the `remote_task` is executed. A core on $D$ reads the metadata information from $M'$ and, based on this information, pre-allocates a destination buffer for the to-be-copied graph $G'$ in the memory partition of $D$.

(3b) $D$ invalidates the destination buffer allocated for $G'$ in both the L1 and L2 cache to avoid interfering cache evictions during the graph copy operation.

(3c) $D$ triggers the `NMCORE` on the memory tile via remote function call of `task_GC`. Additionally, the descriptor of the task $T_2$, which is scheduled on tile $D$ upon completion of the graph copy, is passed.

Next, the task_GC is executed on the NMCORE in the memory tile:

(3d) The NMCORE performs same cloning based software graph copy algorithm as the PEGASUS approach. However, locally near-memory instead of remotely via the NoC and cache hierarchy.

(3e) Upon completion, the NMCORE spawns the task $T_2$ onto a core on $D$ via the network adapter and hardware scheduler.

After the graph copy operation, the steps executed on place $D$ are again identical to the PEGASUS variant.

(4) The task $T_2$ is scheduled on a core on $D$ which executes the function func on the copied graph $G'$.

(5) Finally, the termination of the inter-process communication mechanism is signaled back to $S$. If a result graph was generated in step (4), the same mechanism is applied to copy back the result to place $S$. Thereby, the metadata structures $M$ and $M'$ can be reused.

## 5.7 Evaluation

This section demonstrates the viability and efficiency of the proposed near-memory and near-cache accelerators (NEMESYS). They are integrated into the FPGA-based prototype platform of the tile-based many-core architecture described in Section 2.4. The necessary adaptations to the inter-process communication mechanism were made to the run-time support system. The benchmark applications themselves remained unmodified. The evaluation methodology is defined in Section 5.7.1, including an overview of the to be performed analysis. First, the hardware cost is evaluated in Section 5.7.2. Then, Section 5.7.3 presents an in-depth analysis of the proposed accelerators when integrated into the run-time support system and the X10 IMsuite benchmarks. Finally, Section 5.7.4 compares the near-memory accelerator against the near-memory core (NMCORE) design alternative.

### 5.7.1 Methodology

The evaluation methodology builds upon the quantitative metrics defined in Section 5.4.1.2. The hardware cost is analyzed by comparing the FPGA resources required for the proposed accelerators. The entirety of evaluation aspects is summarized in Table 5.2. The relative communication time, memory intensiveness, and compute efficiency were already the basis for the pre-investigation conducted in Section 5.4. Besides analyzing them for the NEMESYS approach, the performance, scalability, and cache-friendliness of NEMESYS are evaluated against PEGASUS. Furthermore, the effect of the NCA, the inter-memory graph copy, and the design alternative are analyzed.

The extraction of the $t_{gc}$ and $t_{wb}$ times is supported by adding performance counters to the near-memory and near-cache accelerators. The NMA contains 64 Bit counters to track its active time $t_{GC,\texttt{NMA}}$ and memory accesses. The NCA supplies the corresponding counters as well. These performance counters are reset before entering the region of interest and read out after leaving the RoI. Thus, they are non-intrusive.

**Table 5.2: Overview of NEMESYS evaluation aspects.** Except for the hardware cost and frequency analysis, all evaluation are performed with the IMSuite benchmarks.
*Evaluations marked with * are based on the fine-grained and slightly intrusive quantitative metrics. They are performed with one core per tile instead of four.*

| evaluation aspect | based on metric | section |
|---|---|---|
| hardware cost | FPGA resources | Section 5.7.2 |
| attainable frequency | FPGA frequency | Section 5.7.2 |
| performance improvement | IMSuite $t_{exec}$ | Section 5.7.3.1 |
| relative communication time * | IMSuite $t_{proc}$, $t_{ipc}$, $t_{gc}$, $t_{wb}$ | Section 5.7.3.2 |
| memory intensiveness * | IMSuite remote memory bytes | Section 5.7.3.3 |
| compute efficiency * | IMSuite $OpS_{proc}$, $OpS_{active}$, $OpS_{bench}$ | Section 5.7.3.4 |
| inter-memory graph copy | IMSuite $t_{exec}$ | Section 5.7.3.5 |
| scalability | IMSuite $t_{exec}$ | Section 5.7.3.6 |
| cache friendliness | IMSuite $t_{exec}$ | Section 5.7.3.7 |
| effect of NCA | IMSuite $t_{exec}$ | Section 5.7.3.8 |
| design alternative | IMSuite $t_{exec}$ | Section 5.7.4 |

### 5.7.2 Hardware Cost and Frequency Evaluation

The proposed accelerators come at the cost of hardware resources. Table 5.3 shows the FPGA resource requirements of the NMA and NCA, including their sub-modules, when synthesized onto the prototype platform consisting of four Virtex-7 2000T FPGAs. Additionally, the hardware cost of three reference modules (a Leon 3 core, the L2 cache controller, and the main memory controller) are provided. The prototype is operated at a clock frequency of 50 MHz due to bottlenecks in other components.

The NMA graph copy module, even when including the hash map module, requires fewer slices than a Leon 3 core. At the same time, it handles the graph copy operations of up to 64 cores in the system. Furthermore, compared to the main memory controller, near which the NMA is integrated, the NMA requires less than half of the resources. An additional 30% of resource overhead is added by the FIFO of incoming requests and the logic to connect the NMA to the network adapter across the separated clock domains. The FIFO is dimensioned

**Table 5.3: Resource utilization of NEMESYS.** Synthesized on a Virtex-7 2000T FPGA where one slice contains 4 LUTs, 8 Registers, and 2 Muxes. The size of the NMA(once per memory tile) is comparable to a single Leon 3 core (five cores per tile). The size of the NCA(once per compute tile) is one-third the size of the NMA. Thus, both have reasonable resource utilization.

| HW Module | Slices | LUT | Register | Mux | BRAM | DSP |
|---|---|---|---|---|---|---|
| NMA graph copy | 1 862 | 6 078 | 1 163 | 117 | 0 | 0 |
| NMA hash map module | 379 | 1 273 | 554 | 10 | 0 | 0 |
| $\sum$ NMA | 2 241 | 7 351 | 1 717 | 127 | 0 | 0 |
| FIFO & trigger | 547 | 1 354 | 1 264 | 0 | 5 | 0 |
| NCA-RO | 165 | 425 | 357 | 0 | 0 | 0 |
| NCA-WB | 662 | 2 064 | 766 | 12 | 0 | 0 |
| $\sum$ NCA | 827 | 2 489 | 1 123 | 12 | 0 | 0 |
| Leon 3 CPU core | 2 499 | 8 160 | 2 587 | 33 | 18 | 4 |
| L2 cache | 3 440 | 6 092 | 8 898 | 100 | 139 | 0 |
| Main MEM controller | 4 825 | 13 275 | 11 455 | 398 | 0 | 0 |

to accommodate and buffer 16 graph copy requests. The clock domain crossing is necessary as the entire design can only run at 50 MHz due to bottlenecks in some other components, while the minimum frequency of the memory controller is 78 MHz. While the NMA is able to run at 164 MHz as a standalone module, it is operated at the same 100 MHz as the memory controller to provide an integer value clock domain crossing factor.

The lower algorithmic complexity of the NCA compared to the NMA is visible in the required hardware resources. The NCA-RO sub-module is roughly one quarter the size of the NCA-WB sub-module. The entire NCA accumulates to rougly 30% (or 36%) of the resources of the NMA including (or excluding) the FIFO. When comparing the NCA to the L2 cache controller near which it is integrated, its size is roughly one-quarter. Furthermore, the NCA is in the same 50 MHz clock domain as the entire compute tile, although it would even run at 261 MHz. This seems like a downgrade. However, since the NCA is performing a memory-intensive task, its impact is highly dependent on the memory access latency to the main memory via the L2 cache and the NoC. Therefore, a higher NCA frequency would not directly result in an overall performance increase.

Hence, the hardware cost and operating frequency of the NMA and NCA are reasonable.

### 5.7.3 Macrobenchmark Experiments and Results

The previous sections analyzed the hardware cost of the accelerators. This section evaluates the influence of near-memory and near-cache accelerators on whole applications when integrated into the inter-process communication mechanism. In the following, all experiments and results are based on the IMSuite benchmarks (cf. Section 2.4.3).

**Preamble.** An analysis of the copy map implementation revealed that the linear search implementation performs sufficiently well for the graph sizes present in the IMSuite benchmarks. Thus, hashing will not be activated for the copy map implementation. Furthermore, all experiments will be carried out with the prototype platform parameters specified in Table 2.2 (Page 36). The only deviation occurs in Section 5.7.3.7 when the cache friendliness is analyzed. In this case, the L2 cache set size varies between 8 and 128 kBytes.

#### 5.7.3.1 Performance Improvement

The first part of the in-depth analysis evaluates the performance improvement of the whole application when using the NMA and NCA. Therefore, the benchmark execution times $t_{exec}$ of the twelve IMSuite benchmarks in the 4x4 – single configuration (i.e., with one memory tile in use) are measured for the PEGASUS and NEMESYS variant. Figure 5.15 depicts the relative benchmark execution times of the twelve benchmarks for both variants as a bar plot diagram. Each tuple of bars is normalized to the respective PEGASUS variant.

It can be observed that the NEMESYS approach improves the overall benchmark execution times significantly compared to the PEGASUS variant. For all benchmarks, it is reduced at least below 75%, for eight of the twelve benchmarks below 50%, and three of them (BF, DST, and DR) even below 30%. Thus, the overall speedup lies between 1.35x (LCR) and 3.85x (BF). On average, the benchmarks are speedup by 2.2x (i.e., a reduction in execution time to 45%). The lowest speedup is encountered by the LCR (1.35x), the HS (1.43x), and the KC (1.59x) benchmarks since they contain the smallest graph sizes (cf. Table 2.5). Thus, they profit the least from the near-memory graph copy.

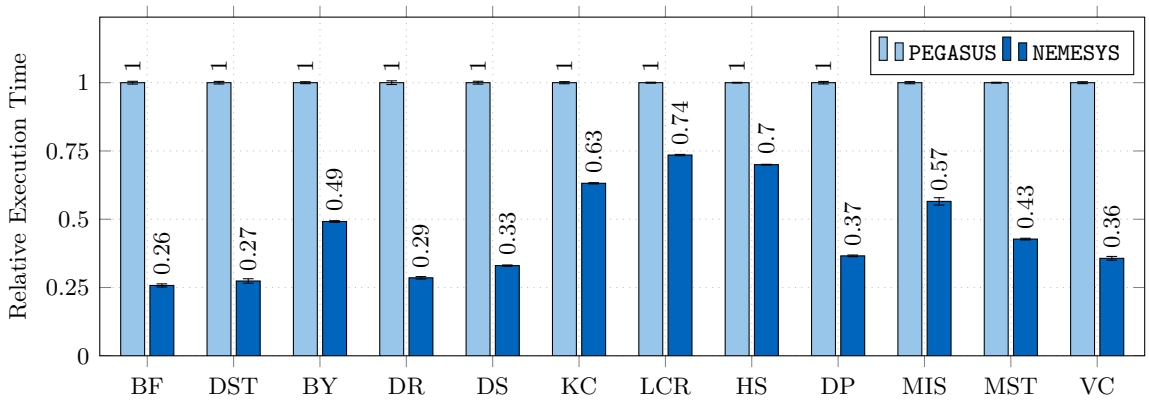



**Figure 5.15: Performance improvement of the IMSuite benchmarks through `NEMESYS`.** The optimized `NEMESYS` inter-process communication leads to a performance speedup of the entire benchmark between 1.35x (LCR) and 3.85x (BF) with an average of 2.2x.

In summary, the proposed near-memory and near-cache accelerators benefit the inter-process communication and the entire application astonishingly. A more detailed analysis of the reasons will be presented in Section 5.7.3.2 and Section 5.7.3.3.

### 5.7.3.2 Relative Communication Time

Section 5.4.4.1 of the pre-investigation revealed that especially the graph copy operation as part of the inter-process communication contributes significantly to the overall run-time of the benchmarks in the `PEGASUS` variant. This section measures and analyzes the same metric (i.e., the relative time of the inter-process communication between tiles) for the `NEMESYS` approach and compares it to the `PEGASUS` variant.

Figure 5.16 depicts the results as tuples of stacked bar plots for the twelve IMsuite benchmarks. Each stacked bar contains the relative times for graph copy $t_{gc}$, graph writeback $t_{wb}$, the remaining minor parts of the `AT statement` (e.g., allocations, notifications) $t_{ipc,rest}$, and the actual processing time $t_{proc}$ of the application apart from inter-process communication. Each tuple of bars is normalized to the run-time $t_{active}$ of the `PEGASUS` variant of the respective benchmark.

Figure 5.16 shows that the inter-process communication time (particularly graph copy and graph writeback) is reduced remarkably in all cases. Most notable is the graph copy time. While the graph copy portion of the `PEGASUS` variant lies between 29% (LCR) and 82% (BF, VC) with an average of 60%, these bounds (when normalized to $t_{active,\texttt{NEMESYS}}$) are lowered to 13% (MIS) and 33% (VC) with an average of 23% for the `NEMESYS` variant. Comparing the absolute graph copy times reveals a reduction in $t_{gc}$ of 60% up to remarkable 93% with an average of 80%. As expected, the most significant reduction occurs for benchmarks with large graph sizes, e.g., BF and DST reduced by 93%. In contrast, benchmarks with smaller graph sizes like KC, LCR, and HS only reduce by 57%, 60%, and 60%, respectively. This fact, together with the shorter absolute graph copy times of these benchmarks in the `PEGASUS` variant, further explains the lower speedup encountered by these benchmarks (cf. Figure 5.15).

Moreover, also the graph writeback time is reduced for `NEMESYS`. While the portion lies between 3% and 14% (8% on average) for `PEGASUS`, it is accelerated to 2% – 5% (4% on average) by `NEMESYS`. Comparing the absolute graph writeback times reveals a reduction in $t_{wb}$ of 74% – 81% with an average of 78%. Furthermore, the remaining parts $t_{ipc,rest}$ of the

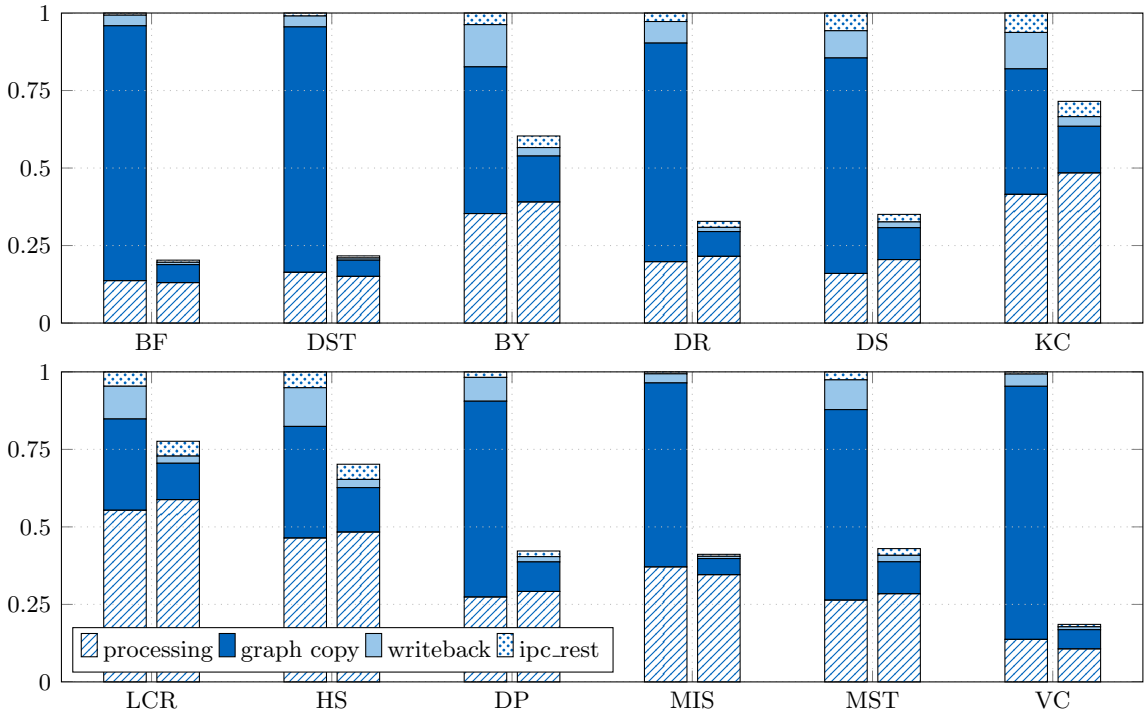



**Figure 5.16: Relative communication time of NEMESYS.** The remarkable reduction of the graph copy time is the largest contributor to the performance speedup presented before. Outsourcing it to the NMA proved crucial for optimizing the inter-process communication.

AT statement have become faster by 12% on average. This is because of reduced software involvement since notifications and completion tasks are initiated by the accelerators.

Finally, when comparing the actual processing times, one would expect them to remain constant for all benchmarks as solely the inter-process communication, and not the processing itself, has been accelerated. Even an increase in processing time could be possible since NEMESYS in contrast to PEGASUS does not have the graph $G'$ partially in its L2 cache after the graph copy. Thus, most benchmarks behave as expected, except for BF, DST, MIS, and VC, since their processing times even decrease slightly. A reason can be that these benchmarks have by far the largest graph sizes (cf. Table 2.5 on Page 39) and may profit from the fact that NEMESYS arranges the copied graph $G'$ into a contiguous chunk of memory. This leads to higher spatial data locality when operating on $G'$. The described effect does not benefit the other eight benchmarks in the NEMESYS variant as they suffer from an increased amount of remote memory accesses during the processing part, which can be seen in the next Figure 5.17.

In summary, it has been shown that the inter-process communication between tiles, and especially the graph copy operation, profit immensely from near-memory acceleration. The remarkable reduction of graph copy time is the largest contributor to the performance speedup presented in the previous Section 5.7.3.1.

### 5.7.3.3 Memory Intensiveness

A second goal of the pre-investigation was to show that the inter-process communication between tiles is a very memory-intensive task with a lot of remote memory accesses (cf. Section 5.4.4.2). This section evaluates how the proposed near-memory acceleration of the

inter-process communication improves this situation. Figure 5.17 depicts the results as tuples of stacked bar plots for the twelve different IMsuite benchmarks. Each stacked bar contains the relative remote memory bytes (i.e., the memory accesses caused by L2 cache misses) for the graph copy, graph writeback, and remaining minor parts of the at statement (e.g., allocations, notifications), as well as the actual processing part of the application apart from inter-process communication. Each tuple of bars is normalized to the total remote memory bytes of the `PEGASUS` variant of the respective benchmark.

Figure 5.17 shows several interesting results. First, the total number of remote memory bytes of the entire benchmark is decreased significantly for the `NEMESYS` variant. The reduction lies between 20% (LCR) and a remarkable 87% (BF), with an average of 63% over all benchmarks. For the `PEGASUS` variant, the most considerable portion of the remote memory bytes has been attributed to the graph copy operation (70% on average). For the `NEMESYS` variant, these remote memory bytes are reduced to zero since the graph copy is performed near-memory instead of remotely via the NoC.

However, as expected, the remote memory bytes caused by the graph writeback operation basically remain the same for the respective benchmarks. Although the operation is outsourced to the `NCA`, the `NCA` still has to writeback the graph $G$ to memory via remote memory accesses.

In contrast, the remote memory bytes caused by the actual processing increase for almost all benchmarks. This is because the destination graph $G'$ is not cached in the L2 cache after the graph copy in the `NEMESYS` approach but has to be read from the main memory.

The analysis shows that the near-memory accelerated inter-process communication reduces remote memory accesses and improves overall performance.

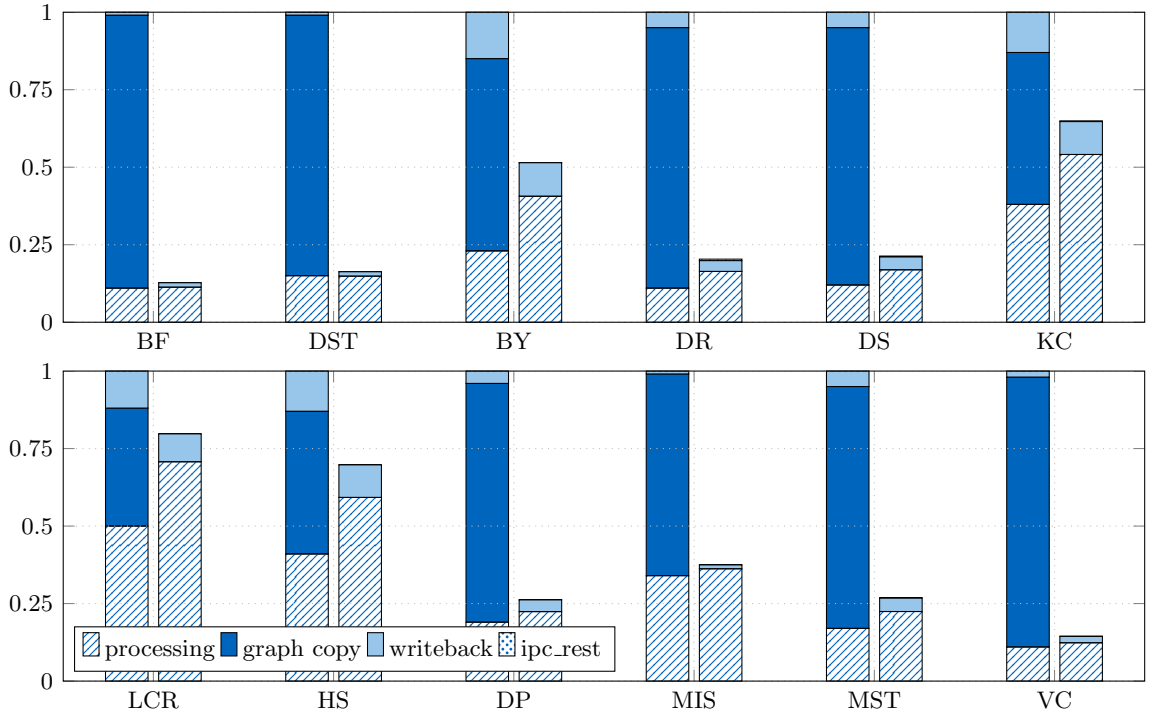



**Figure 5.17: Memory intensiveness of `NEMESYS`.** The extremely memory intensive `PEGASUS` graph copy mechanism is replaced by the near-memory graph copy of the `NEMESYS` which drastically reduces the high-latency remote memory accesses.

#### 5.7.3.4 Compute Efficiency

The third main insight of the pre-investigation was that the far-from-memory graph copy sub-task of the PEGASUS variant has a degrading effect on the entire application (cf. Section 5.4.4.3). In this section, it will be analyzed how the proposed accelerators improve compute efficiency.

Figure 5.18 top, middle, and bottom depict the absolute $OpS_{proc}$, $OpS_{active}$, and $OpS_{bench}$, respectively, for all twelve IMSuite benchmarks for the PEGASUS and the NEMESYS variants in MOps/s. Figure 5.18 top shows that the pure processing performance $OpS_{proc}$ (processing operations in relation to the processing time) of the PEGASUS and NEMESYS variant are similar. While the NEMESYS variant is better in four of the twelve benchmarks (BF, DST, MIS, and VC), the PEGASUS variant has a higher $OpS_{proc}$ for the other eight benchmarks. First of all, the processing performance is expected to stay roughly constant since, solely, the inter-process communication part has been accelerated. Second, the slight performance decrease of the processing part for the NEMESYS variant can be explained: the destination graph is not cached in the L2 cache after the near-memory graph copy. As explained in

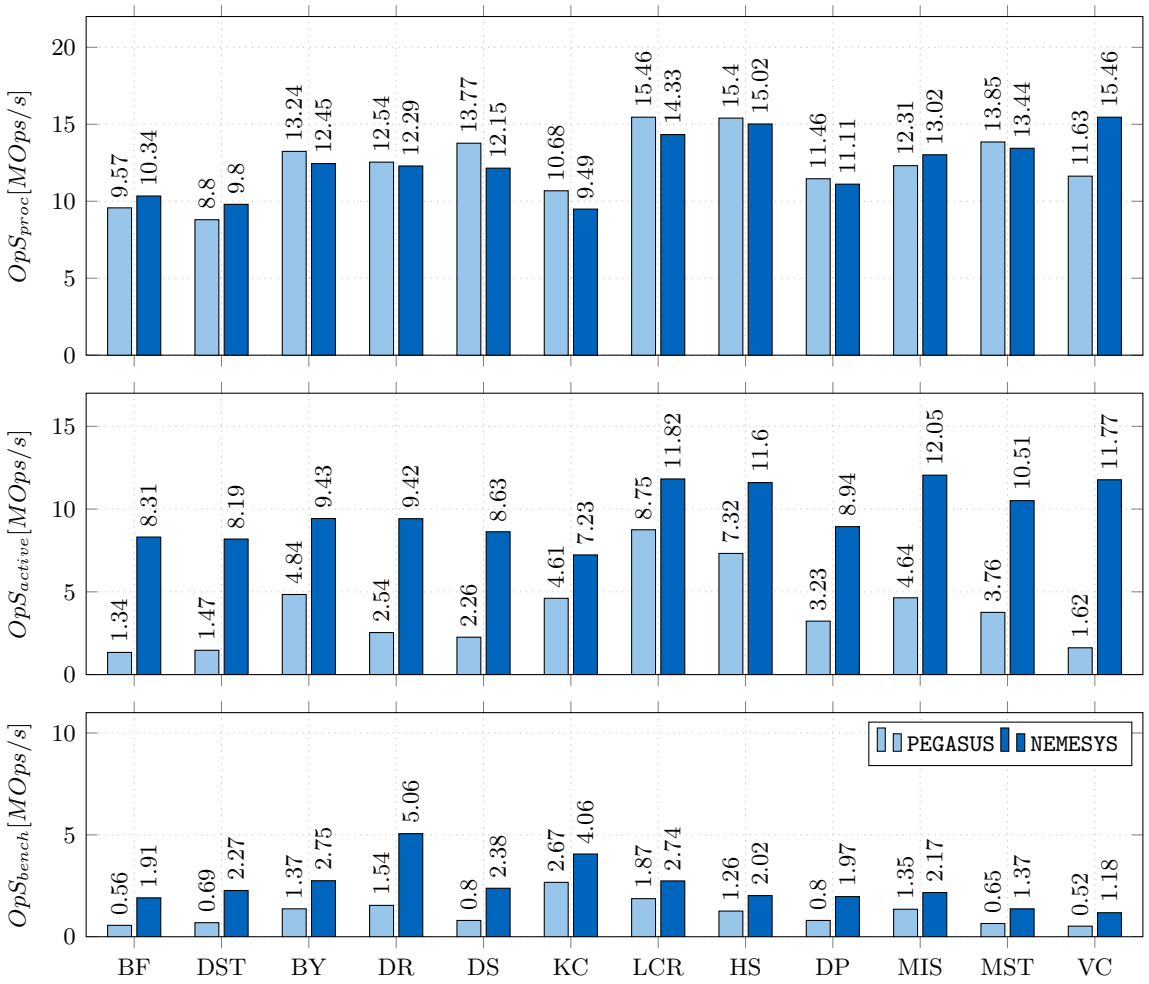



**Figure 5.18: Compute efficiency of NEMESYS.** The accelerated inter-process communication has a highly positive impact on the application's compute efficiency. The $OpS_{active}$ (processing plus communication) are only 1.27x instead of 4.18x lower than $OpS_{proc}$ (pure processing).

the previous section, this is the same effect that leads to the processing part's increased memory intensiveness.

When analyzing the processing operations in relation to the total active time of the cores during the region of interest $OpS_{active}$ (cf. Figure 5.18 middle), different results occur. While, the $OpS_{active,\texttt{PEGASUS}}$ reduce significantly and are between 1.77x and 7.18x (4.18x on average) lower than $OpS_{proc,\texttt{PEGASUS}}$, $OpS_{active,\texttt{NEMESYS}}$ is only between 1.08x and 1.41x (1.27x on average) lower than $OpS_{proc,\texttt{NEMESYS}}$. This is expected and in line with the relative communication times depicted in Figure 5.16.

However, these times do not directly translate to the achieved performance speedup encountered by the application (cf. Section 5.7.3.1) since the application is also inherently bound by limited parallelism according to Amdahl's law. Thereby, a subset of the cores will be partially idle during the benchmark execution. This fact is incorporated in the metric $OpS_{bench} = \frac{Ops_{proc}}{t_{proc}+t_{ipc}+t_{idle}}$, which is plotted in Figure 5.18 bottom. The ratio of these numbers is well in line with the application's actual performance speedup (cf. Section 5.7.3.1).

In summary, the near-memory accelerated inter-process communication has a highly positive impact on the compute efficiency of the entire application.

#### 5.7.3.5 Inter-Memory

The previous sections analyzed the performance speedup, the relative communication time, the memory intensiveness, and the compute efficiency of `NEMESYS` for the 4x4`-single` configuration, i.e., with one memory tile in use. This section evaluates the performance of the proposed concept for the 4x4`-twin` against the 4x4`-single` configuration.

Figure 5.19 depicts the relative benchmark execution times of all twelve IMSuite benchmarks in two bar plot diagrams. The upper bar plot diagram compares `PEGASUS-single` versus `PEGASUS-twin`, whereas the lower bar plot diagram adds the bars for `NEMESYS-single` and `-twin`. All four bars corresponding to the same benchmark are normalized to the respective `PEGASUS-single` variant.

Figure 5.19 reveals several insights. For all benchmarks, `PEGASUS-twin` benefits from two physically distributed memory tiles since memory access hot-spots are thereby mitigated. The performance improvement lies between 2% (HS) and 36% (DR). In contrast, `NEMESYS-twin` seldom profits from physically distributed memory tiles. A noticeable improvement from 63% down to 48% is achievable only for the KC benchmark. While two other benchmarks (BY 1% and DR 2%) profit insignificantly, the other nine benchmarks suffer from a minor performance degradation between 1% (DS) and 15% (VC) (normalized to `PEGASUS-single`). This can be explained by the overhead of the advanced inter-memory graph copy mechanism (cf. Section 5.6.1.2) requiring the additional inter-memory DMA. Statistically, it is required in half of the graph copy operations (assuming that half of the compute tiles have their memory partition in each of two memory tiles).

Nevertheless, when comparing `NEMESYS-twin` to `PEGASUS-twin`, the speedup factor lies between 1.3x (LCR) and 2.37x (DR), with an average speedup of 1.86x over all twelve benchmarks. Compared to the `PEGASUS-single` variant, the average speedup of `NEMESYS-twin` is 2.12x, which is well in the range of the `NEMESYS-single` speedup of 2.2x.

In summary, although the performance of `NEMESYS-twin` suffers from the additional inter-memory DMA, it still outperforms `PEGASUS-twin` remarkably. Thus, the proposed concept is well applicable to tile-based architectures with multiple memory tiles.

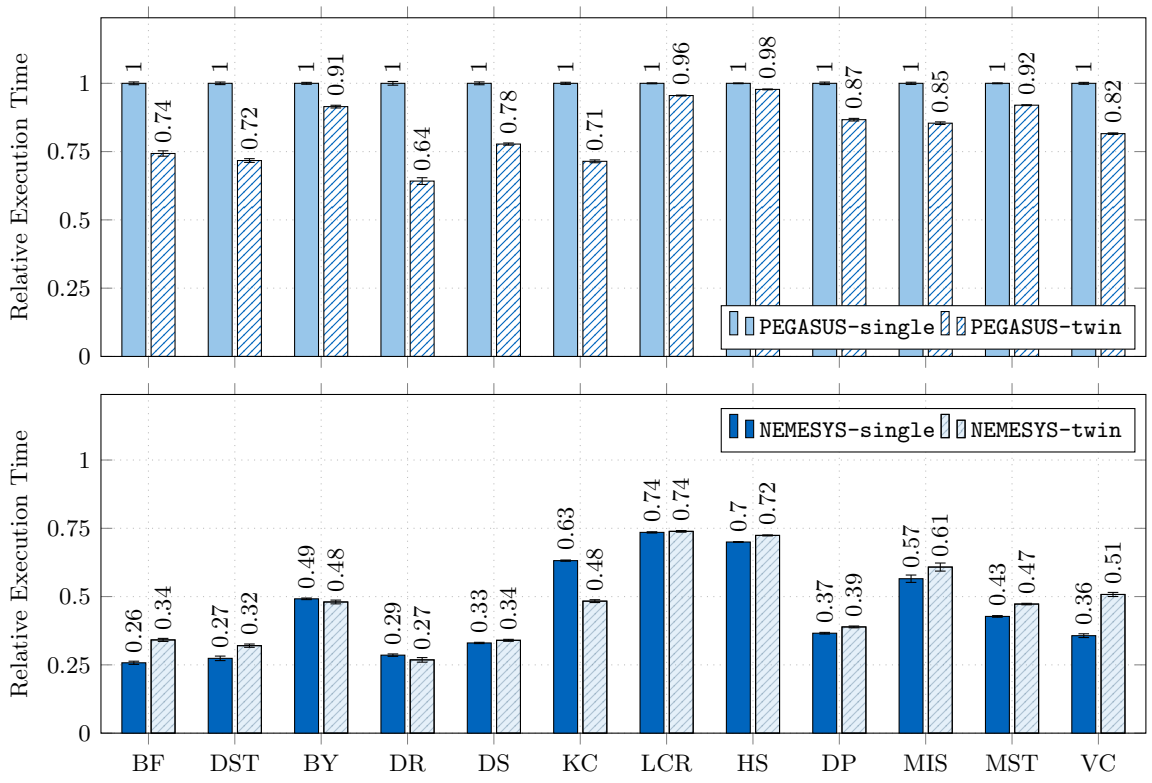



**Figure 5.19: Inter-memory graph copy performance of NEMESYS.** Through the advanced inter-memory graph copy mechanism of the NMA the important near-memory benefits are preserved also for inter-memory graph copies. Thus, NEMESYS is applicable to architectures with multiple memory tiles.

#### 5.7.3.6 Scalability Analysis

This section investigates the scaling behavior of the near-memory graph copy approach by comparing the speedups achieved in the 4x4 – single configuration versus a 2x2 tile configuration. Figure 5.20 depicts the speedup of the NEMESYS over to the PEGASUS variant for the 2x2 and 4x4 – single configuration of all twelve IMSuite benchmarks in a bar plot diagram. Each bar is normalized to the respective PEGASUS variant, i.e., either to the PEGASUS 2x2 or 4x4 – single variant.

Figure 5.20 reveals that the majority of benchmarks encounters a larger speedup of the near-memory graph copy approach in the 4x4 than in the 2x2 variant. For example, BF-2x2-NEMESYS is 1.88x faster than BF-2x2-PEGASUS, while BF-4x4-NEMESYS is 3.88x faster than BF-4x4-PEGASUS. The analysis of the IMSuite benchmarks shows a super-linear relation between the amount of inter-process communication and the number of tiles [193]. Thus, the 4x4 variant has to handle more inter-process communication between tiles, both in absolute and relative numbers. Furthermore, the analysis of the NMA utilization reveals a mean NMA utilization of 50 % in the 4x4 – single variant (i.e., one NMA is serving all 14 compute tiles) and of 25.1 % in the 4x4 – twin variant (i.e., using two memory tiles and thus, each NMA is serving seven compute tiles). The peak NMA utilization in these three scenarios, encountered by the DR benchmark, is 90.5 %, and 52 %, respectively. However, real-world applications usually consist of a mix of different benchmark kernels. Thus, the mean utilization is an eligible indicator for the NMA utilization of real-world applications.

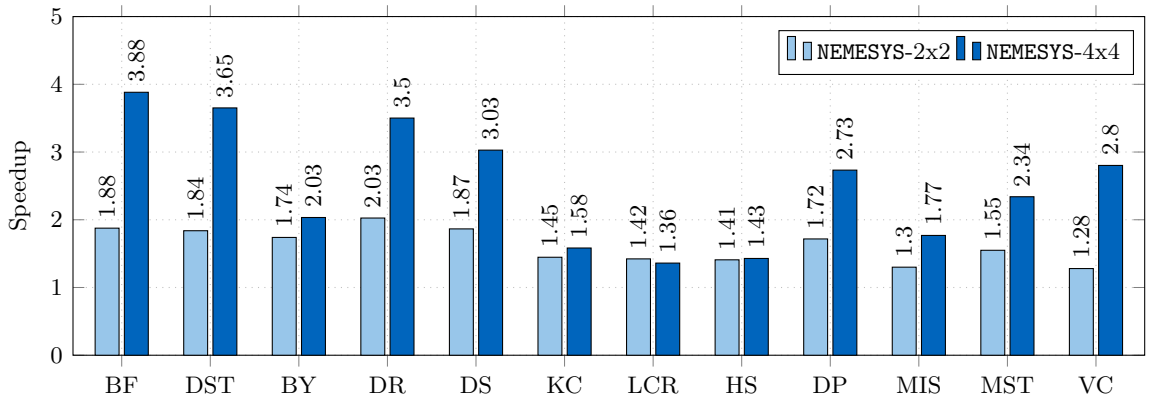



**Figure 5.20: Scalability analysis of NEMESYS.** The vast majority of IMSuite benchmarks encounters a larger relative speedup of NEMESYS compared to PEGASUS for bigger system sizes as the inter-process communication demand increases. Thus, NEMESYS scales well to larger architectures.

On the other hand, there also exist benchmarks that do not scale much better in the 4x4 variant (e.g., KC or HS) or even worse (LCR). Table 2.5 in Section 2.4.3 shows that these benchmarks use mostly small graphs during graph copy. They have a higher relative overhead of the remaining parts of the at statement in comparison to the actual accelerated graph copy.

In summary, the near-memory graph copy approach scales well for larger tile-based architectures. Even a single NMA can service the inter-process communication needs of a 4x4 tile design with 14 compute tiles and four application cores each.

#### 5.7.3.7 Analysis of Cache Friendliness

This section investigates the cache friendliness of the approach, i.e., how the performance is affected by various L2 cache sizes. Throughout all twelve benchmarks, the largest graph size is approximately 16 kBytes (cf. Table 2.5 on Page 39). Therefore, the evaluation is performed with three different L2 cache set sizes of the 4-way associative L2 cache: small (8 kByte), medium (16 kByte), or large (128 kByte).

The benchmark execution times of the twelve IMSuite benchmarks in the 4x4 – single configuration are measured for the PEGASUS and NEMESYS variant. Figure 5.21 Top and Figure 5.21 Bottom depict the results for PEGASUS and NEMESYS as bar plot diagrams, respectively. Both diagrams contain a triple of bars per benchmark. Each triple is normalized to the respective PEGASUS-4x128 kB or NEMESYS-4x128 kB variant, i.e., the variant with an L2 cache size of 4x128 kByte (4 sets à 128 kByte).

Figure 5.21 reveals that both approaches experience performance degradation with smaller L2 cache sizes, which is expected. However, the PEGASUS approach suffers worse with smaller cache sizes than the NEMESYS approach, especially for benchmarks with mostly large-sized graphs like BF, DST, or VC. In contrast, the LCR benchmark has mostly small graphs with a size of less than 1 kB. Thus, the difference between PEGASUS and NEMESYS is negligible for LCR, even for the small cache set size of 8 kB since it is still much bigger than the graph size.

In general, the PEGASUS graph copy via the NoC and L2 cache pollutes the cache and causes massive cache line evictions for other cores on the same tile. This does not happen for the cache friendlier NEMESYS near-memory graph copy. Furthermore, the analysis shows

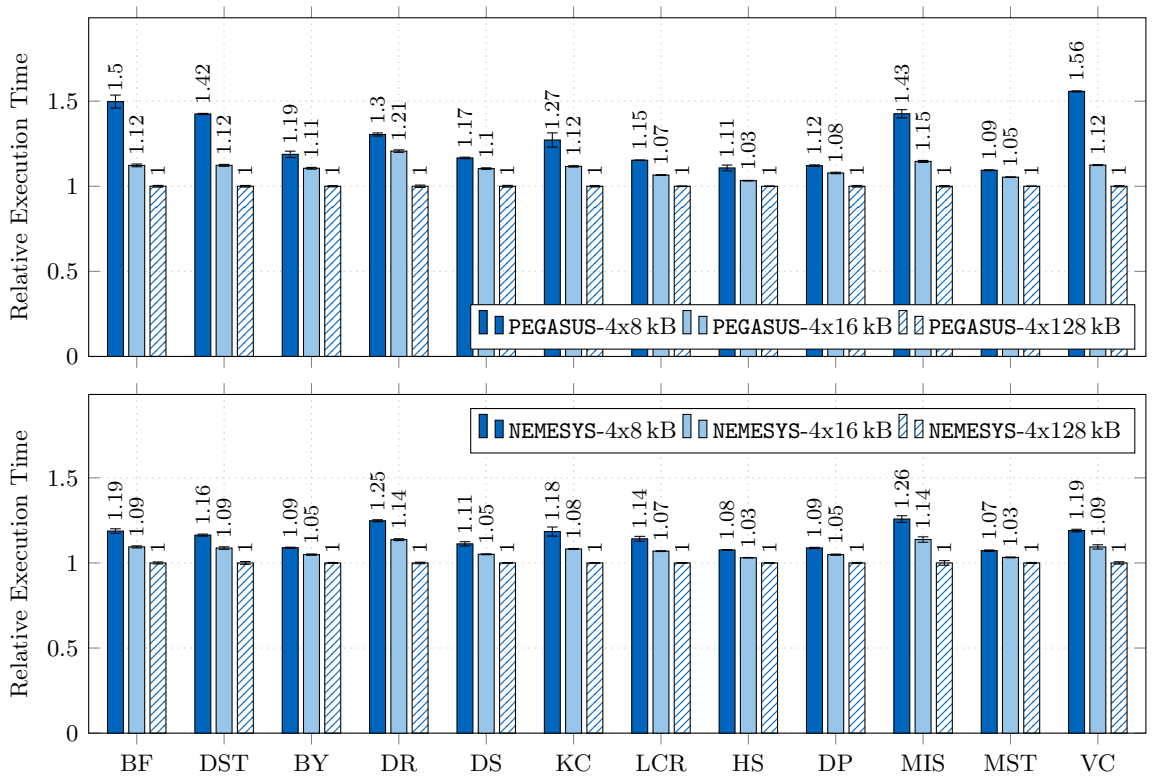



**Figure 5.21: Analysis of NEMESYS's cache friendliness.** If the L2 cache size is smaller, NEMESYS experiences much less performance degradation than PEGASUS. This is because PEGASUS produces more cache pollution during its remote graph copy operations via the cache hierarchy.

that the L2 cache size of 4x128 kB used in all other evaluations does not have an unfair influence on the results as it is much bigger than the to-be-copied graph sizes. If, however, the observation by Peter Kogge [19] applies that today's application data sets outgrow the cache sizes by far, the results for the small 4x8 kB might be more relevant. It will depend on the actual use case.

#### 5.7.3.8 Effect of Near-Cache Accelerators

In the previous sections, various aspects of the near-memory and near-cache accelerated inter-process communication have been investigated with NMA and NCAs activated. Since it may be costly to include an NCA in every compute tile, this section analyzes the effect of the near-cache accelerators on the benchmark performance. In other words, this section evaluates and compares the speedup achieved when only using the NMA without the NCAs.

Figure 5.22 depicts the results as bar plot diagrams comparing the three variants: PEGASUS (no accelerators), NEMESYS-noNCA (only the NMA), and NEMESYS (both NMA and NCA). For each of the twelve IMSuite benchmarks, the triple of bars is normalized to the respective PEGASUS variant. The 4x4 – single configuration is used.

Figure 5.22 reveals that for the vast majority of benchmarks (except LCR and HS), most of the speedup is achieved by the NMA itself. On top of that, a measurable contribution is added by the NCA for these ten benchmarks. In contrast, for LCR and HS, the better part of the speedup arises from the NCA. As these two benchmarks have the smallest graph

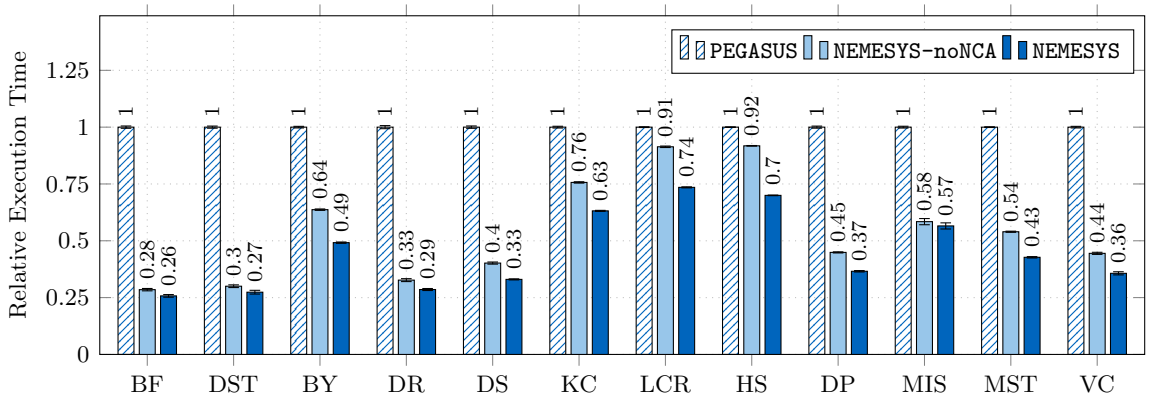



**Figure 5.22: Effect of the near-cache accelerator (NCA) on the benchmark performance.** This plot compares PEGASUS (no accelerators), NEMESYS-noNCA (only the NMA), and NEMESYS (both NMA and NCA). For most benchmarks (except LCR and HS), the bigger part of the speedup is attributed to the NMA while the NCA still adds a measurable improvement.

sizes, they do not profit as much from the NMA as the other benchmarks with medium and large graph sizes. Nevertheless, they have a relatively higher benefit from the asynchronous offloading of the graph writeback to the NCA.

### 5.7.4 Comparing against the Design Alternative Near-Memory Core

The previous sections presented the evaluation results of the near-memory accelerated inter-process communication. Finally, this section analyzes the NMCORE design alternative. This investigation aims to determine which portion of the speedup arises from the location (i.e., near-memory vs. far-from memory) and the type of function implementation (i.e., dedicated hardware accelerator vs. software programmable core).

Thus, the IMSuite benchmark execution times were additionally measured for the NMCORE variant. It is compared against the PEGASUS and NEMESYS variants in the same 4x4 – single configuration as above. For a fair comparison, all variants do not use the NCA. Furthermore, the NMCORE is supported by a state-of-the-art local DMA engine in the memory tile used to copy contiguous memory ranges above 128 Bytes. Figure 5.23 depicts the relative benchmark execution times of the twelve benchmarks for all three variants as a bar plot diagram. Each triple of bars is normalized to the respective NMCORE variant.

Figure 5.23 reveals that the overall speedup of the NMCORE design alternative is in almost all cases in between the performance of the PEGASUS and the NMA variant. In eight of the twelve benchmarks (BF, DST, BY, DS, DP, MIS, MST, and VC), the bigger part of the speedup is achieved by the near-memory vs. far-from-memory integration. For example, BF has a speedup of 2.39x from the PEGASUS to the NMCORE variant, whereas the latter is only improved by another 1.47x when using the NMA (relative duration: 0.68). However, in two of the twelve benchmarks (DR and KC), the most speedup is added by the NMA, e.g., DR improves by 1.67x from PEGASUS to the NMCORE and by another 1.82x from the NMCORE to the NMA. Again, LCR and HS stand out. For these remaining two benchmarks, the NMCORE performs slightly better than the NMA itself. This can be explained by the small graphs used in these benchmarks, which have a higher overhead of the graph traversal in comparison to the actual payload copying. This acts in favor of the NMCORE, which can

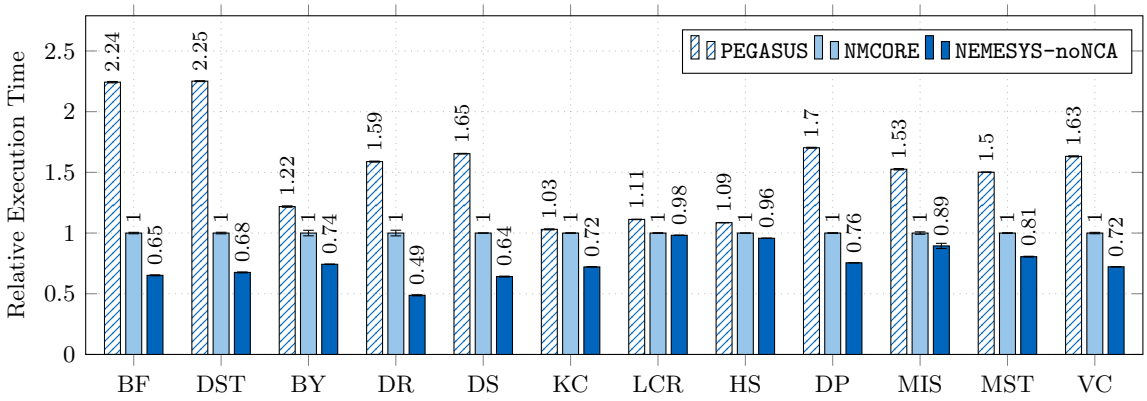



**Figure 5.23: Comparing NEMESYS against the NMCORE design alternative.** This plot compares PEGASUS (no accelerators), NMCORE (no accelerators), and NEMESYS-noNCA (only the NMA). The bigger part of the performance improvement is achieved by the location, i.e., the near-memory integration. Furthermore, the NMA adds a significant benefit in most cases.

cache the information necessary for the graph traversal (e.g., the objects' *RTTI*) in its L1 cache. In contrast, the NMA has to access the memory for every object afresh.

In summary, the bigger part of the performance improvement is achieved by the location, i.e., the near-memory implementation. Moreover, the dedicated hardware accelerator adds a significant benefit in most cases.

## 5.8 Summary

Efficient inter-process communication is crucial for the performance of parallel applications on many-core architectures with distributed shared memory. The previous chapters revealed that increased data-to-task locality and minimized software involvement in the inter-process synchronization and communication are vital in such systems. Near-memory accelerated concurrent data structures and queue management were proposed. While the previous chapters focused on flat (i.e., non-pointered) data, object-oriented programming languages also need efficient transfers of pointered data structures like object graphs.

This chapter presented a near-memory accelerated graph copy approach integrated into the run-time support system. The goal was to mitigate the data locality challenges of the graph copy-based inter-process communication between tiles through near-memory acceleration. Although the contribution was showcased with the object-oriented PGAS programming language X10, it also applies to other systems.

In the first step, a qualitative and quantitative pre-investigation of related approaches was performed to identify and quantify the potential for improvement. The analysis of the closest related work unveiled the graph copy and writeback sub-tasks of the inter-process communication as substantial bottlenecks. For instance, the graph copy sub-task contributes significantly to the overall benchmark execution time (29% – 82%) and the high-latency remote memory accesses (38% – 88%). This memory-intensive task further has a degrading impact on the entire application.

In the next step, the design space was explored qualitatively, and the concepts of NEMESYS and a design alternative (NMCORE) were presented. NEMESYS outsources the memory-intensive graph copy operations to the NMA and the graph writeback operation to the NCA. Both types

of accelerators were integrated into the run-time support system to allow seamless integration without changing the application code or API. The design alternative does not use the accelerators but instead outsources the graph copy sub-task to the `NMCORE`.

The evaluation of `NEMESYS` with the PGAS-based IMSuite benchmarks showed that, for instance, the entire benchmark applications achieved a performance speedup between 1.35x and 3.85x (2.2x on average). Moreover, the relative portion of the inter-process communication between tiles was reduced from 45% – 86% (71% on average) to 16% – 43% (32% on average). Finally, comparing `NEMESYS` to the `NMCORE` design alternative revealed that the more significant part of the improvement was achieved by the near-memory location, whereas the dedicated accelerator still adds a significant benefit in most cases.

# 6 Conclusion and Outlook

> "If it weren't for Moore's law changing the playing field continuously, I would have been long gone. The rapid pace of hardware evolution still keeps things fresh for me."
>
> — John Carmack [194]

This thesis contributes research results to the field of near-memory acceleration for run-time support and operating systems on tile-based many-core architectures. In contrast to numerous application-specific near-memory computing approaches, the contributions of this thesis focused on run-time support functionalities integrated at OS or library level. Specifically, inter-process synchronization and communication, which are crucial to the performance of parallel applications, were addressed. The goal was to mitigate data-to-task locality challenges arising from spatially distributed processing and memory nodes in tile-based systems. The contributions were inspired by the statement of John Backus: "Surely there must be a less primitive way of making big changes in the store than by pushing vast numbers of words back and forth through the von Neumann bottleneck."

## 6.1 Conclusion

Work on this thesis was initially motivated by the finding that hardware support for inter-process synchronization between tiles is mostly missing on tile-based many-core architectures. Especially remote memory access latencies and software overhead of existing software approaches limited their efficiency.

Similarly, for inter-process communication between tiles, existing mechanisms often relied on DMA-based message-passing and remote task invocation to notify the receiver. Multiple remote round-trips and increased software involvement were necessary to establish effective communication. Moreover, no dedicated support for transferring more complex data structures such as object graphs existed.

**Contributions.** Therefore, this thesis presented three contributions that facilitate efficient inter-process synchronization and communication on tile-based architectures and mitigate data-to-task locality challenges. Their commonalities are *remote near-memory execution*, *hardware acceleration*, and *support for run-time functionalities*. The concepts were integrated and evaluated with an FPGA-based prototype of a tile-based many-core architecture. They are based on a hardware-software co-design approach which provided additional degrees of freedom, enabling optimizations that would not have been possible otherwise.

**The first solution proposes a near-memory synchronization accelerator** which executes inter-tile atomic operations near-memory instead of remotely via the NoC. Experiments revealed that lightweight operations significantly reduce the overhead of synchronizing shared data structures, particularly for remote operations. Moreover, it was shown that

the concept scales well to larger system sizes and handles increased concurrency on the data structures with pristine performance while related approaches degrade significantly. Furthermore, the accelerator was employed to build an efficient memory allocator for recurring inter-tile allocations of communication metadata buffers. Whereas a speedup of 19.3x was achieved for standalone operations, the integration into the IMSuite benchmarks revealed an improvement of up to 8% (4% on average). The performance is compelling since the accelerated functionality accounts for only a minor part of the inter-process communication and an even smaller portion of the entire application.

This first contribution served as a proof-of-concept for run-time support near-memory acceleration, which was applied to more complex scenarios by the other two contributions. While it focused on synchronization via remote atomic operations, the following solutions target inter-process communication, including data transfer and receiver notification.

**The second solution extends the concept to a near-memory queue management accelerator** which facilitates efficient inter-process communication since queues are beneficial and widely employed, e.g., for message-passing or task distribution. The proposed concept features software-defined queues to enable configurability and flexibility known of software queue approaches during queue creation. However, the queues are hardware-managed by the presented near-memory accelerator to achieve performance, scalability, and minimized software overhead. The entire queue and memory management, including data transfer and receiver notification, are outsourced to hardware near the memory where the particular queue resides. The conducted study revealed that this is crucial for data-to-task locality, particularly in tile-based systems with physically distributed memory. Especially remote queue operations benefit significantly from the near-memory acceleration and reduced software involvement. For instance, the duration of the queue management excluding the data transfer was reduced from 3111 (software-managed queue) to 311 (10x) clock cycles (hardware-managed queue) for a remote enqueue operation. Moreover, the integration into the MPI library and evaluation with the MPI-based NAS parallel benchmarks showed a speedup in execution time of up to 75% for the most communication-intensive kernel. Furthermore, the approach can dynamically adapt to run-time requirement changes in the queue's memory footprint. Thus, the memory consumption of the NAS benchmark was reduced significantly compared to a static queue allocation.

The evaluation showed that the proposed software-defined hardware-managed queue approach is a powerful extension to the portfolio of inter-process communication primitives. Whereas this contribution targeted queue-based communication where data could be transferred efficiently via DMA, there also exist more complex inter-process communication scenarios which require the transfer of object graphs.

**Therefore, the third solution presents a near-memory graph copy accelerator** which mitigates data locality challenges of the PGAS inter-process communication between tiles. A pre-investigation identified the cache-unfriendly and memory-intensive graph copy and graph writeback operations, required by the inter-process communication across tile borders, as the most and second most significant bottlenecks. Thus, the proposed near-memory accelerator (NMA) performs the graph copy operation near-memory instead of remotely over the NoC and the cache hierarchy. Furthermore, additional near-cache accelerators (NCAs) relieve the cores of the graph writeback operation. Both types of accelerators were tightly integrated into the run-time support system to allow seamless integration without changing the application code or API.

The evaluation with the PGAS-based X10 IMSuite benchmarks showed that the entire benchmark applications achieved a performance speedup between 1.35x and 3.85x with an average of 2.2x. The relative portion of the inter-process communication between tiles was reduced from 45% – 86% (71% on average) to 16% – 43% (32% on average). Furthermore, the approach was well applicable for designs with multiple memory tiles, and the performance improvements were even better on bigger (16 tiles) than on smaller (4 tiles) systems. Moreover, the more significant part of the speedup is attributed to the NMA, although the NCA adds a measurable improvement. The moderate hardware costs of the NMA instantiated once per memory tile are roughly the same as of a LEON 3 CPU core. The less complex NCA, instantiated once per compute tile, is roughly one-third the size of a LEON 3 core.

Further analysis was performed compared to a design alternative that does not incorporate the proposed accelerators but instead outsources the graph copy sub-task to a near-memory core. The conducted study revealed that the more significant part of the improvement was achieved by the near-memory location, whereas the dedicated accelerator still adds a significant benefit in most cases. Thus, the dedicated graph copy accelerator works more efficiently than a programmable core and relieves the cores of this duty. It is the natural enhancement of a DMA unit to support graph- and pointer-based data structures.

The work presented in this thesis contributes a first step towards tackling inter-process communication challenges of tile-based many-core architectures in an interdisciplinary manner by integrating near-memory accelerators into communication libraries and the run-time system. Finally, the applicability and limitations of the contributions will be discussed.

**Applicability and Limitations.** The presented contributions have in common that dedicated near-memory accelerators were employed to support library and run-time functionalities like inter-process communication. Nevertheless, their usage is not limited to the scenarios or programming paradigms in which they were employed in this thesis. Some general aspects are addressed before the applicability and limitations of the specific contributions are discussed.

First, all concepts were designed, implemented, and evaluated on the FPGA-based prototype of the tile-based many-core architecture. The prototype implementation and the architecture itself have certain characteristics (e.g., clock frequency, memory access latency and bandwidth, or cache hierarchy) which are different from other systems. While this may influence some design details and evaluation results, the overarching data locality challenge remains as it is pervasive in many-core systems. For instance, due to the single clock domain of the prototype, the CPU cores run at a much slower frequency than the residual system than usual. However, speeding up the cores would further worsen the memory-boundedness of an already memory-bound system.

Second, the modular design of the accelerators as standalone components connected to the tile-local bus creates implementation independence of other components and the particular memory technology and controller. For instance, on the prototype, all accelerators are able to operate without adaptations on both tile-local SRAM memory and global DDR-SDRAM memory. Thus, they can easily be integrated into other systems. However, as the accelerators have to access the memory via the tile-local bus, they are not as *near* to memory as possible (e.g., integrated into the memory controller or the memory itself). Nevertheless, the biggest part of the distance is bridged by being on the right side of the global interconnect and not accessing the memory remotely via the NoC and cache hierarchy.

Third, the accelerators were integrated into the OctoPOS operating system developed in the context of the Invasive Computing research project. It helps create and exploit several degrees of freedom in the hardware-software co-design approach like using a hardware scheduler or the lightweight interaction with the accelerators. While the mentioned aspects are no conceptual requirements, they enabled optimizations that might not be possible with other operating systems.

After discussing the common aspects, the applicability and limitations of the individual contributions are assessed. **The first contribution** targeted inter-process synchronization of concurrent data structures in the shared-memory programming paradigm. Nevertheless, as was also shown in this thesis, it was applied to build an inter-tile allocator requiring synchronization. It was employed in the PGAS inter-process communication mechanism, i.e., the first contribution has been partially integrated into the third contribution. In general, the concept of the synchronization accelerator applies to more complex data structures. However, a dedicated hardware implementation of the desired functionality would be required, limiting the flexibility of the approach.

**The second contribution** was integrated into the MPI library to accelerate inter-process message-passing communication. Nevertheless, the software-defined hardware-managed queue concept is not limited to this use case. Since it offers user-defined general-purpose queues in normal memory, it can be applied to any queue-based scenario. For instance, Maier et al. [195] employed them as an asynchronous system interface. Traditional system calls of applications were replaced by lightweight enqueue operations with minimal software overhead. The proposed handler task, which, in this case, is running on a dedicated OS core, ensures the processing of the system requests. Thereby, the desired asynchronous decoupling of the requester and handler was achieved.

While the current implementation is limited to FIFO queues, supporting other data structures such as priority queues or trees may be desirable.

Finally, **the third contribution** was integrated into the inter-process communication mechanism of the PGAS programming language X10. Although X10 is used as the programming language to showcase the contribution, the concept also applies to other languages. For instance, the communication mechanism of the PGAS language Chapel is quite similar to X10's. Therefore, Chapel can also benefit from this work when applied to suitable hardware architectures. More general, the concept is applicable whenever object graph transfers are necessary. The near-memory graph copy accelerator can be used as a pointer-aware enhancement of a DMA unit. Like DMAs, it can be used asynchronously since the accelerator can schedule a user-defined task on the receiver side upon completion of the transfer.

On the other hand, if the programming model does not require object graph transfers, the accelerator is of no use to it. Furthermore, the accelerator unfolds its full potential if the source and destination graph reside in the same physical memory; since then, the graph copy can be performed entirely *near-memory*. Nevertheless, an advanced inter-memory graph copy mechanism was also presented which retains most of the crucial *near-memory* benefits (the reader is referred to Section 5.6.1.2).

All in all, since the three contributions are integrated at the operating system and library level, opaque to the application programmer, their applicability is already much broader than if they were application-specific solutions. Nonetheless, they are not limited to the presented scenarios and could be used directly at the application level.

## 6.2 Outlook

There exist several potential directions for future research on the topic. For example, another tile-based architecture prototype with different characteristics could be employed to evaluate the concepts further. Possibilities are, for instance, different clock frequencies, other memory technologies (e.g., HMC or HBM), or different CPU core types and counts. Besides, it would also be worth investigating whether the proposed contributions also benefit inter-process communication in other than tile-based architectures. The modular design of the accelerators enables ease of implementation into them. Moreover, the accelerators could be integrated into a different operating system or other communication libraries like the Pthread library.

Furthermore, more research into accelerating inter-process synchronization and communication can be done. Possible primitives that potentially profit from hardware acceleration and reduced software overhead are, for instance, condition variables that are used to implement signaling and waiting between processes. The conventional implementation requires scheduler involvement to wake up the waiting (i.e., sleeping) thread. Maintaining the primitives and initiating the wake-up in and by hardware can be an idea worth pursuing.

Additionally, research on improving the presented contributions can be done. The synchronization accelerator could be upgraded to support further shared data structures like e.g., hash tables, search trees, or priority queues [140, 22]. Since the proposed concept of outsourcing the critical section, which has to be executed atomically, has similarities to transactional memory, the latter would be a path worth pursuing. It would be a generalization of the proposed complex and compositional atomic operations.

Various directions to improve the software-defined hardware-managed queue concept exist. First, more use-cases other than message-passing could be explored, such as task scheduling or work-stealing. Second, the strict FIFO ordering could be upgraded to priority queues due to their relevance for e.g., load balancing, task scheduling, sorting, or search algorithms. Third, adding software-defined and hardware-enforced access permissions to the queues would enable efficient communication across isolation domains. For instance, it could ease the communication between user and kernel space while unauthorized access to particular queues would be prohibited.

Since the graph copy accelerator is part of an object-oriented system, its compatibility with garbage collection is required. However, it does not yet attach the garbage collector information to the individual objects in its current state of implementation. Two possible solutions were laid out in Section 5.6.2.2. Another direction of future research can be to modify the graph copy accelerator to serve as a hardware garbage collector itself. The core functionality (i.e., traversing object graphs and copying all reachable objects) of a semi-space garbage collector already exists. Moreover, as mentioned before, the graph copy accelerator can be used as a generalized pointer-aware enhancement of a DMA unit to support data transfers in object-oriented languages.

Finally, besides focusing on inter-process communication, further research on the integration and harmonization of near-memory computing with existing systems and programming paradigms is desirable to avoid the von Neumann bottleneck and data-to-task locality challenges.

# Bibliography


[1] S. Rheindt, A. Schenk, A. Srivatsa, T. Wild, and A. Herkersdorf. CaCAO: Complex and Compositional Atomic Operations for NoC-Based Manycore Platforms. In *Architecture of Computing Systems - ARCS 2018 - 31st International Conference, Braunschweig, Germany, April 9-12, 2018, Proceedings*, pages 139–152, 2018. doi:10.1007/978-3-319-77610-1_11.

[2] S. Rheindt, S. Maier, F. Schmaus, T. Wild, W. Schröder-Preikschat, and A. Herkersdorf. SHARQ: Software-Defined Hardware-Managed Queues for Tile-Based Manycore Architectures. In *Embedded Computer Systems: Architectures, Modeling, and Simulation - 19th International Conference, SAMOS 2019, Samos, Greece, July 7-11, 2019, Proceedings*, pages 212–225, 2019. doi:10.1007/978-3-030-27562-4_15.

[3] S. Rheindt, A. Fried, O. Lenke, L. Nolte, T. Wild, and A. Herkersdorf. NEMESYS: Near-Memory Graph Copy Enhanced System-Software. In *Proceedings of the International Symposium on Memory Systems, MEMSYS 2019, Washington, DC, USA, September 30 - October 03, 2019*, pages 3–18. ACM, 2019. doi:10.1145/3357526.3357545.

[4] S. Rheindt, A. Fried, O. Lenke, L. Nolte, T. Sabirov, T. Twardzik, T. Wild, and A. Herkersdorf. X-CEL: A Method to Estimate Near-Memory Acceleration Potential in Tile-Based MPSoCs. In *Architecture of Computing Systems - ARCS 2020 - 33rd International Conference, Aachen, Germany, May 25-28, 2020, Proceedings*, pages 109–123, 2020. doi:10.1007/978-3-030-52794-5_9.

[5] S. Rheindt, T. Sabirov, O. Lenke, T. Wild, and A. Herkersdorf. X-Centric: A Survey on Compute-, Memory- and Application-Centric Computer Architectures. In *MEMSYS 2020: The International Symposium on Memory Systems, Washington, DC, USA, September, 2020*, pages 178–193. ACM, 2020. doi:10.1145/3422575.3422792.

[6] S. Rheindt, S. Maier, N. Pohle, L. Nolte, O. Lenke, F. Schmaus, T. Wild, W. Schröder-Preikschat, and A. Herkersdorf. DySHARQ: Dynamic Software-Defined Hardware-Managed Queues for Tile-Based Architectures. *International Journal of Parallel Programming*, pages 1–35, 2020. doi:10.1007/s10766-020-00687-7.

[7] S. Rheindt, A. Srivatsa, O. Lenke, L. Nolte, T. Wild, and A. Herkersdorf. *Tackling the MPSoC Data Locality Challenge*, chapter 5, pages 85–117. John Wiley & Sons, Ltd, 2021. doi:10.1002/9781119818298.ch5.

[8] A. Srivatsa, S. Rheindt, T. Wild, and A. Herkersdorf. Region-Based Cache Coherence for Tiled MPSoCs. In M. Alioto, H. H. Li, J. Becker, U. Schlichtmann, and R. Sridhar, editors, *30th IEEE International System-on-Chip Conference, SOCC 2017, Munich, Germany, September 5-8, 2017*, pages 286–291. IEEE, 2017. doi:10.1109/SOCC.2017.8226059.

[9] A. Srivatsa, S. Rheindt, D. Gabriel, T. Wild, and A. Herkersdorf. CoD: Coherence-on-Demand - Runtime Adaptable Working Set Coherence for DSM-Based Manycore Architectures. In *Embedded Computer Systems: Architectures, Modeling, and Simulation - 19th International Conference, SAMOS 2019, Samos, Greece, July 7-11, 2019, Proceedings*, pages 18–33, 2019. doi:10.1007/978-3-030-27562-4_2.

[10] A. Srivatsa, M. Mansour, S. Rheindt, D. Gabriel, T. Wild, and A. Herkersdorf. DynaCo: Dynamic Coherence Management for Tiled Manycore Architectures. *International Journal of Parallel Programming*, pages 1–30, 2021. doi:10.1007/s10766-020-00688-6.

[11] J. Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[12] Charles Arthur, The Guardian. Tech giants may be huge, but nothing matches big data. https://www.theguardian.com/technology/2013/aug/23/tech-giants-data, 2013.

[13] David Parkins, The Economist. The world's most valuable resource is no longer oil, but data. https://www.economist.com/leaders/2017/05/06/the-worlds-most-valuable-resource-is-no-longer-oil-but-data, 2017.

[14] David Turek. The Transformation of High Performance Computing: Simulation and Cognitive Methods in the Era of Big Data, 2018. http://www.hpcadvisorycouncil.com/events/2018/swiss-workshop/pdf/Monday09April/Turek_HPCTransformation_Keynote_Mon090418.pdf.

[15] D. Reinsel, J. Gantz, and J. Rydning. The digitization of the world – from edge to core. *IDC White Paper US44413318*, 2018. URL: https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf.

[16] M. M. Sabry Aly, M. Gao, G. Hills, C. Lee, G. Pitner, M. M. Shulaker, T. F. Wu, M. Asheghi, J. Bokor, F. Franchetti, K. E. Goodson, C. Kozyrakis, I. Markov, K. Olukotun, L. Pileggi, E. Pop, J. Rabaey, C. Ré, H. . P. Wong, and S. Mitra. Energy-efficient abundant-data computing: The n3xt 1,000x. *Computer*, 48(12):24–33, 2015.

[17] Peter Sondergaard. Gartner Symposium/ITxpo 2011, October 16–20, Orlando.

[18] G. E. Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35, 2006. doi:10.1109/N-SSC.2006.4785860.

[19] P. Kogge. Memory Intensive Computing, the 3rd Wall, and the Need for Innovation in Architecture, 2017. Keynote at The International Symposium on Memory Systems (MEMSYS), https://extremecomputingtraining.anl.gov/files/2017/08/ATPESC_2017_Dinner_Talk_6_8-4_Kogge-Data_Intensive_Computing.pdf.

[20] O. Arnold and G. P. Fettweis. Power aware heterogeneous mpsoc with dynamic task scheduling and increased data locality for multiple applications. In F. J.

Kurdahi and J. Takala, editors, *Proceedings of the 2010 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS 2010), Samos, Greece, July 19-22, 2010*, pages 110–117. IEEE, 2010. doi:10.1109/ICSAMOS.2010.5642075.

[21] W. J. Dally. Computer architecture in the many-core era. In *24th International Conference on Computer Design (ICCD 2006), 1-4 October 2006, San Jose, CA, USA*, page 1. IEEE, 2006. doi:10.1109/ICCD.2006.4380784.

[22] M. Moir and N. Shavit. Concurrent data structures. In *Handbook of Data Structures and Applications*, 2004.

[23] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, et al. Exascale computing study: Technology challenges in achieving exascale systems. *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, 15, 2008.

[24] M. Radulovic, D. Zivanovic, D. Ruiz, B. R. de Supinski, S. A. McKee, P. Radojković, and E. Ayguadé. Another trip to the wall: How much will stacked dram benefit hpc? In *Proceedings of the 2015 International Symposium on Memory Systems*, pages 31–36, 2015.

[25] P. M. Kogge. Execube-a new architecture for scaleable mpps. In *1994 International Conference on Parallel Processing Vol. 1*, volume 1, pages 77–84. IEEE, 1994.

[26] N. Jayasena. Memory-centric accelerators in high-performance systems. In *55th Design Automation Conference (DAC 2018), 24-28 June 2018, San Francisco, CA USA, Special Session on "Memory-centric Architectures: Industry Perspective from Embedded Systems to High Performance Computing"*, 2018.

[27] B. Falsafi, M. Stan, K. Skadron, N. Jayasena, Y. Chen, J. Tao, R. Nair, J. H. Moreno, N. Muralimanohar, K. Sankaralingam, and C. Estan. Near-memory data services. *IEEE Micro*, 36(1):6–13, 2016. doi:10.1109/MM.2016.9.

[28] P. Siegl, R. Buchty, and M. Berekovic. Data-centric computing frontiers: A survey on processing-in-memory. In B. Jacob, editor, *Proceedings of the Second International Symposium on Memory Systems, MEMSYS 2016, Alexandria, VA, USA, October 3-6, 2016*, pages 295–308. ACM, 2016. doi:10.1145/2989081.2989087.

[29] G. Singh, L. Chelini, S. Corda, A. J. Awan, S. Stuijk, R. Jordans, H. Corporaal, and A.-J. Boonstra. Near-memory computing: Past, present, and future. *Microprocessors and Microsystems*, 71:102868, 2019.

[30] M. Motoyoshi. Through-silicon via (tsv). *Proceedings of the IEEE*, 97(1):43–48, 2009.

[31] J. T. Pawlowski. Hybrid memory cube (hmc). In *2011 IEEE Hot Chips 23 Symposium (HCS)*, pages 1–24, 2011.

[32] J. Standard. High bandwidth memory (hbm) dram. *JESD235*, 2013.

[33] D. A. Patterson, T. E. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. E. Kozyrakis, R. Thomas, and K. A. Yelick. A case for intelligent RAM. *IEEE Micro*, 17(2):34–44, 1997. doi:10.1109/40.592312.

[34] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. Intelligent ram (iram): Chips that remember and compute. In *1997 IEEE International Solids-State Circuits Conference. Digest of Technical Papers*, pages 224–225. IEEE, 1997.

[35] NVIDIA Corporation. NVIDIA A100 Tensor Core GPU Architecture. Technical report, 2020. https://www.megware.com/fileadmin/user_upload/LandingPage%20NVIDIA/nvidia-ampere-architecture-whitepaper.pdf.

[36] M. Deo, J. Schulz, and L. Brown. Intel Stratix 10 MX Devices with Samsung HBM2 Solve the Memory Bandwidth Challenge. Technical report, Intel Corporation, 2016. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01264-stratix10mx-devices-solve-memory-bandwidth-challenge.pdf.

[37] M. S. Won. Intel Agilex FPGAs Deliver a Game-Changing Combination of Flexibility and Agility for the Data-Centric World. Technical report, Intel Corporation, 2019. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/intel-agilex-fpgas-deliver-game-changing-combination-wp.pdf.

[38] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu. Knights landing: Second-generation intel xeon phi product. *Ieee micro*, 36(2):34–46, 2016.

[39] D. Patterson. The trouble with multi-core. *IEEE Spectrum*, 47(7):28–32, 2010.

[40] J. Henkel, A. Herkersdorf, L. Bauer, T. Wild, M. Hübner, R. K. Pujari, A. Grudnitsky, J. Heißwolf, A. Zaib, B. Vogel, V. Lari, and S. Kobbe. Invasive manycore architectures. In *17th Asia and South Pacific Design Automation Conference*, pages 193–200, 2012.

[41] George Santayana. *The Life of Reason: Reason in Common Sense*. Scribner's, 1905.

[42] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Computer Architecture News*, 23(1):20–24, 1995. doi:10.1145/216585.216588.

[43] J. Parkhurst, J. Darringer, and B. Grundmann. From single core to multi-core: Preparing for a new exponential. In *2006 IEEE/ACM International Conference on Computer Aided Design*, pages 67–72, Nov 2006. doi:10.1109/ICCAD.2006.320067.

[44] R. Dennard, F. Gaensslen, H.-N. Yu, V. Rideout, E. Bassous, and A. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974. doi:10.1109/JSSC.1974.1050511.

[45] D. Kirk and W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach (Second Edition)*. Morgan Kaufmann, 2013.

[46] T. Burger. Intel multi-core processors: Quick reference guide, 2005. https://www.researchgate.net/publication/228386317_Intel_Multi-Core_Processors_Quick_Reference_Guide.

[47] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C. Miao, J. F. B. III, and A. Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5):15–31, 2007. doi:10.1109/MM.2007.89.

[48] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The stanford dash multiprocessor. *Computer*, 25(3):63–79, March 1992. doi:10.1109/2.121510.

[49] B. D. de Dinechin, R. Ayrignac, P.-E. Beaucamps, P. Couvert, B. Ganne, P. G. de Massas, F. Jacquet, S. Jones, N. M. Chaisemartin, F. Riss, and T. Strudel. A clustered manycore processor architecture for embedded and accelerated applications. In *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2013. doi:10.1109/HPEC.2013.6670342.

[50] Advanced Micro Devices, Inc. AMD Ryzen Threadripper 3990X Processor. https://www.amd.com/en/products/cpu/amd-ryzen-threadripper-3990x/, 2021.

[51] Intel Corporation. Intel Core Processor Family. https://www.intel.com/content/www/us/en/products/details/processors/core.html, 2021.

[52] Advanced Micro Devices, Inc. AMD Ryzen Desktop Processors. https://www.amd.com/en/ryzen, 2021.

[53] L. Chai, Q. Gao, and D. K. Panda. Understanding the impact of multi-core architecture in cluster computing: A case study with intel dual-core system. In *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2007), 14-17 May 2007, Rio de Janeiro, Brazil*, pages 471–478. IEEE Computer Society, 2007. doi:10.1109/CCGRID.2007.119.

[54] Y. Durand, P. M. Carpenter, S. Adami, A. Bilas, D. Dutoit, A. Farcy, G. Gaydadjiev, J. Goodacre, M. Katevenis, M. Marazakis, E. Matús, I. Mavroidis, and J. Thomson. EUROSERVER: energy efficient node for european micro-servers. In *17th Euromicro Conference on Digital System Design, DSD 2014, Verona, Italy, August 27-29, 2014*, pages 206–213. IEEE Computer Society, 2014. doi:10.1109/DSD.2014.15.

[55] NVIDIA Corporation. NVIDIA Titan RTX. https://www.nvidia.com/en-us/deep-learning-ai/products/titan-rtx/, 2018.

[56] Arm Limited. big.LITTLE Technology: The Future of Mobile. Technical report, 2013. https://img.hexus.net/v2/press_releases/arm/big.LITTLE.Whitepaper.pdf.

[57] Arm Limited. ARM DynamIQ: The future of multi-core computing. Technical report, 2017. https://community.arm.com/cfs-file/__key/communityserver-blogs-components-weblogfiles/00-00-00-21-42/ARM-DynamIQ-The-future-of-multi_2D00_core-computing.pdf.

[58] S. Mittal. A survey on evaluating and optimizing performance of intel xeon phi. *Concurrency and Computation: Practice and Experience*, 32:e5742, 2020.

[59] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. F. B. III, M. Mattina, C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. TILE64 - processor: A 64-core soc with mesh interconnect. In *2008 IEEE International Solid-State Circuits Conference, ISSCC 2008, Digest of Technical Papers, San Francisco, CA, USA, February 3-7, 2008*, pages 88–89, San Francisco, CA, USA, 2008. IEEE. doi:10.1109/ISSCC.2008.4523070.

[60] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. V. D. Wijngaart, and T. Mattson. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *2010 IEEE International Solid-State Circuits Conference - (ISSCC)*, pages 108–109, Feb 2010. doi:10.1109/ISSCC.2010.5434077.

[61] T. Mattson, B. Sanders, and B. Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, 2014.

[62] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, and B. Falsafi. Scale-out processors. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, page 500–511, USA, 2012. IEEE Computer Society.

[63] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele. Many-core key-value store. In *Proceedings of the 2011 International Green Computing Conference and Workshops*, IGCC '11, page 1–8, USA, 2011. IEEE Computer Society. doi:10.1109/IGCC.2011.6008565.

[64] G. Chrysos. Intel Xeon Phi X100 Family Coprocessor - the Architecture. Technical report, Intel Corporation, 2012. https://software.intel.com/content/www/us/en/develop/articles/intel-xeon-phi-coprocessor-codename-knights-corner.html.

[65] Intel Corporation. Intel Many Integrated Core Architecture. Technical report, 2012. https://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture/.

[66] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan. Heterogeneous chip multiprocessors. *Computer*, 38(11):32–38, November 2005. doi:10.1109/MC.2005.379.

[67] V. J. Reddi, H. Yoon, and A. Knies. Two billion devices and counting. *IEEE Micro*, 38(1):6–21, 2018.

[68] N. Chitlur, G. Srinivasa, S. Hahn, P. K. Gupta, D. Reddy, D. A. Koufaty, P. Brett, A. Prabhakaran, L. Zhao, N. Ijih, S. Subhaschandra, S. Grover, X. Jiang, and R. R. Iyer. Quickia: Exploring heterogeneous architectures on real prototypes. In *18th IEEE International Symposium on High Performance Computer Architecture, HPCA 2012, New Orleans, LA, USA, 25-29 February, 2012*, pages 433–440. IEEE Computer Society, 2012. doi:10.1109/HPCA.2012.6169046.

[69] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation—a performance view. *IBM Journal of Research and Development*, 51(5):559–572, Sep 2007. doi:10.1147/rd.515.0559.

[70] P. Greenhalgh. Big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7. Technical report, Arm Limited, 2011. https://www.eetimes.com/big-little-processing-with-arm-cortex-a15-cortex-a7/#.

[71] S. Mittal and J. S. Vetter. A survey of CPU-GPU heterogeneous computing techniques. *ACM Comput. Surv.*, 47(4):69:1–69:35, 2015. doi:10.1145/2788396.

[72] A. Branover, D. Foley, and M. Steinman. Amd fusion apu: Llano. *IEEE Micro*, 32(2):28–37, Mar 2012. doi:10.1109/MM.2012.2.

[73] M. Yuffe, E. Knoll, M. Mehalel, J. Shor, and T. Kurts. A fully integrated multi-cpu, GPU and memory controller 32nm processor. In *IEEE International Solid-State Circuits Conference, ISSCC 2011, Digest of Technical Papers, San Francisco, CA, USA, 20-24 February, 2011*, pages 264–266. IEEE, 2011. doi:10.1109/ISSCC.2011.5746311.

[74] S. Damaraju, G. Varghese, S. Jahagirdar, T. Khondker, R. Milstrey, S. Sarkar, S. Siers, I. Stolero, and A. Subbiah. A 22nm IA multi-cpu and GPU system-on-chip. In *2012 IEEE International Solid-State Circuits Conference, ISSCC 2012, San Francisco, CA, USA, February 19-23, 2012*, pages 56–57. IEEE, 2012. doi:10.1109/ISSCC.2012.6176876.

[75] E. Sotiriades and A. Dollas. A general reconfigurable architecture for the blast algorithm. *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 48(3):189–208, Sep 2007. doi:10.1007/s11265-007-0069-2.

[76] V. M. Morales, P. Horrein, A. Baghdadi, E. Hochapfel, and S. Vaton. Energy-efficient FPGA implementation for binomial option pricing using opencl. In G. P. Fettweis and W. Nebel, editors, *Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014*, pages 1–6. European Design and Automation Association, 2014. doi:10.7873/DATE.2014.221.

[77] G. R. Goslin. Guide to using field programmable gate arrays (FPGAs) for application-specific digital signal processing performance. In J. Schewel, P. M. Athanas, V. M. B. Jr., and J. Watson, editors, *High-Speed Computing, Digital Signal Processing, and Filtering Using Reconfigurable Logic*, volume 2914, pages 321 – 331. International Society for Optics and Photonics, SPIE, 1996. doi:10.1117/12.255830.

[78] R. J. Petersen and B. L. Hutchings. An assessment of the suitability of fpga-based systems for use in digital signal processing. In W. Moore and W. Luk, editors, *Field-Programmable Logic and Applications*, pages 293–302, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[79] G. Nash and J. Moawad. Intel FPGAs for Deep Learning. https://press3.mcs.anl.gov//atpesc/files/2019/08/ATPESC_2019_Track-1_8_7-29_330pm_Moawad_Nash-FPGAs.pdf, 2019.

[80] W. Feng and P. Balaji. Tools and environments for multicore and many-core architectures. *IEEE Computer*, 42(11):26–27, 2009. doi:10.1109/MC.2009.412.

[81] J. Diaz, C. Muñoz-Caro, and A. Niño. A survey of parallel programming models and tools in the multi and many-core era. *IEEE Transactions on Parallel and Distributed Systems*, 23(8):1369–1386, 2012.

[82] H. Kasim, V. March, R. Zhang, and S. See. Survey on parallel programming model. In J. Cao, M. Li, M.-Y. Wu, and J. Chen, editors, *Network and Parallel Computing*, pages 266–275, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[83] D. A. Kranz, K. L. Johnson, A. Agarwal, J. Kubiatowicz, and B. Lim. Integrating message-passing and shared-memory: Early experience. In M. C. Chen and R. Halstead, editors, *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP), San Diego, California, USA, May 19-22, 1993*, pages 54–63. ACM, 1993. doi:10.1145/155332.155338.

[84] E. Belikov, P. Deligiannis, P. Totoo, M. Aljabri, and H.-W. Loidl. A survey of high-level parallel programming models. *Heriot-Watt University, Edinburgh, UK*, 1:2–2, 2013.

[85] IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(TM)) Base Specifications, Issue 7. *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)*, pages 1–3951, 2018. doi:10.1109/IEEESTD.2018.8277153.

[86] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, Jan 1998. doi:10.1109/99.660313.

[87] P. Pacheco. *Parallel Programming with MPI*. Morgan Kaufman Publishers, 1996.

[88] M. Mohr and C. Tradowsky. Pegasus: Efficient data transfers for PGAS languages on non-cache-coherent many-cores. In D. Atienza and G. D. Natale, editors, *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017*, pages 1781–1786. IEEE, 2017. doi:10.23919/DATE.2017.7927281.

[89] M. De Wael, S. Marr, B. De Fraine, T. Van Cutsem, and W. De Meuter. Partitioned global address space languages. *ACM Comput. Surv.*, 47(4), May 2015. doi:10.1145/2716320.

[90] UPC Consortium and Bonachea, Dan and Funck, Gary. UPC Language and Library Specifications, Version 1.3. Technical report, 2013. https://escholarship.org/uc/item/2hm8m5x0.

[91] R. W. Numrich and J. Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998. doi:10.1145/289918.289920.

[92] K. A. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. N. Hilfinger, S. L. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance java dialect. *Concurrency - Practice and Experience*, 10:825–836, 1998.

[93] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. X10 Language Specification v2.6.2. Technical report, 2019. http://x10.sourceforge.net/documentation/languagespec/x10-latest.pdf.

[94] Hewlett Packard Enterprise Development LP. Chapel documentation v1.24. Technical report, 2021. https://chapel-lang.org/docs/index.html.

[95] NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture. Technical report, 2009. https://www.nvidia.de/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[96] Khronos Group. An introduction to opencl c++. Technical report, 2015. https://www.khronos.org/assets/uploads/developers/resources/Intro-to-OpenCL-C++-Whitepaper-May15.pdf.

[97] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Process. Lett.*, 21(2):173–193, 2011. doi:10.1142/S0129626411000151.

[98] M. M. K. Martin, M. D. Hill, and D. J. Sorin. Why on-chip cache coherence is here to stay. *Commun. ACM*, 55(7):78–89, 2012. doi:10.1145/2209249.2209269.

[99] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. T. Morris, and N. Zeldovich. An analysis of linux scalability to many cores. In R. H. Arpaci-Dusseau and B. Chen, editors, *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, pages 1–16. USENIX Association, 2010. URL: http://www.usenix.org/events/osdi10/tech/full_papers/Boyd-Wickizer.pdf.

[100] G. Blake, R. G. Dreslinski, and T. Mudge. A survey of multicore processors. *IEEE Signal Processing Magazine*, 26(6):26–37, 2009.

[101] D. Burger, J. R. Goodman, and A. Kägi. Memory bandwidth limitations of future microprocessors. In J. Baer, editor, *Proceedings of the 23rd Annual International Symposium on Computer Architecture, Philadelphia, PA, USA, May 22-24, 1996*, pages 78–89. ACM, 1996. doi:10.1145/232973.232983.

[102] D. A. Patterson. Latency lags bandwidth. *Commun. ACM*, 47(10):71–75, October 2004. doi:10.1145/1022594.1022596.

[103] R. H. Dennard. Field-effect transistor memory, June 4 1968. US Patent 3,387,286.

[104] R. Minnick, R. Short, J. Goldberg, H. Stone, and M. Green. Cellular arrays for logic and storage. Technical report, Stanford Research Inst., Menlo Park, Calif., 1966.

[105] H. S. Stone. A logic-in-memory computer. *IEEE Transactions on Computers*, 100(1):73–78, 1970.

[106] D. G. Elliott, W. M. Snelgrove, and M. Stumm. Computational ram: A memory-simd hybrid and its application to dsp. In *Custom Integrated Circuits Conference*, volume 30, pages 1–30, 1992.

[107] P. M. Kogge, J. B. Brockman, T. Sterling, and G. Gao. Processing in memory: Chips to petaflops. In *Workshop on Mixing Logic and DRAM: Chips that Compute and Remember at ISCA*, volume 97. Citeseer, 1997.

[108] A. Saulsbury, F. Pong, and A. Nowatzyk. Missing the memory wall: The case for processor/memory integration. *SIGARCH Comput. Archit. News*, 24(2):90–101, May 1996. doi:10.1145/232974.232984.

[109] E. S. Larsen and D. McAllister. Fast matrix multiplies using graphics hardware. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, pages 55–55, 2001.

[110] R. Egawa, M. Sato, J. Tada, and H. Kobayashi. Vertically integrated processor and memory module design for vector supercomputers. In *2013 IEEE International 3D Systems Integration Conference (3DIC)*, pages 1–6, 2013.

[111] Hybrid Memory Cube Consortium. Hybird Memory Cube Specification 2.1. Technical report, 2014. https://www.nuvation.com/sites/default/files/Nuvation-Engineering-Images/Articles/FPGAs-and-HMC/HMC-30G-VSR_HMCC_Specification.pdf.

[112] Advanced Micro Devices, Inc. High-Bandwidth Memory (HBM) Reinventing Memory Technology. Technical report, 2015. https://www.amd.com/system/files/documents/high-bandwidth-memory-hbm.pdf.

[113] D. U. Lee, K. W. Kim, K. W. Kim, H. Kim, J. Y. Kim, Y. J. Park, J. H. Kim, D. S. Kim, H. B. Park, J. W. Shin, et al. 25.2 a 1.2 v 8gb 8-channel 128gb/s high-bandwidth memory (hbm) stacked dram with effective microbump i/o test methods using 29nm process and tsv. In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 432–433. IEEE, 2014.

[114] M. J. Khurshid and M. H. Lipasti. Data compression for thermal mitigation in the hybrid memory cube. In *2013 IEEE 31st International Conference on Computer Design, ICCD 2013, Asheville, NC, USA, October 6-9, 2013*, pages 185–192. IEEE Computer Society, 2013. doi:10.1109/ICCD.2013.6657041.

[115] Y. Xie. Future memory and interconnect technologies. In E. Macii, editor, *Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013*, pages 964–969. EDA Consortium San Jose, CA, USA / ACM DL, 2013. doi:10.7873/DATE.2013.202.

[116] W. I. Kinney, W. Shepherd, W. Miller, J. Evans, and R. Womack. A non-volatile memory cell based on ferroelectric storage capacitors. In *1987 International Electron Devices Meeting*, pages 850–851, 1987.

[117] K. Matsuyama, H. Asada, S. Ikeda, and K. Taniguchi. Low current magnetic-ram memory operation with a high sensitive spin valve material. *IEEE Transactions on Magnetics*, 33(5):3283–3285, 1997.

[118] X. Dong, X. Wu, G. Sun, Y. Xie, H. Li, and Y. Chen. Circuit and microarchitecture evaluation of 3d stacking magnetic RAM (MRAM) as a universal memory

replacement. In L. Fix, editor, *Proceedings of the 45th Design Automation Conference, DAC 2008, Anaheim, CA, USA, June 8-13, 2008*, pages 554–559. ACM, 2008. doi:10.1145/1391469.1391610.

[119] T. Endoh, H. Koike, S. Ikeda, T. Hanyu, and H. Ohno. An overview of nonvolatile emerging memories - spintronics for working memories. *IEEE J. Emerg. Sel. Topics Circuits Syst.*, 6(2):109–119, 2016. doi:10.1109/JETCAS.2016.2547704.

[120] S. H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li. NDC: analyzing the impact of 3d-stacked memory+logic devices on mapreduce workloads. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2014, Monterey, CA, USA, March 23-25, 2014*, pages 190–200. IEEE Computer Society, 2014. doi:10.1109/ISPASS.2014.6844483.

[121] D. P. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski. TOP-PIM: throughput-oriented programmable processing in memory. In B. Plale, M. Ripeanu, F. Cappello, and D. Xu, editors, *The 23rd International Symposium on High-Performance Parallel and Distributed Computing, HPDC'14, Vancouver, BC, Canada - June 23 - 27, 2014*, pages 85–98. ACM, 2014. doi:10.1145/2600212.2600213.

[122] S. W. Jun, M. Liu, S. Lee, J. Hicks, J. Ankcorn, M. King, S. Xu, and Arvind. Bluedbm: an appliance for big data analytics. In D. T. Marr and D. H. Albonesi, editors, *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*, pages 1–13. ACM, 2015. doi:10.1145/2749469.2750412.

[123] G. H. Loh, N. Jayasena, M. Oskin, M. Nutter, D. Roberts, M. Meswani, D. P. Zhang, and M. Ignatowski. A Processing-in-Memory Taxonomy and a Case for Studying Fixed-function PIM. In *Workshop on Near-Data Processing (WoNDP)*, pages 1–4, 2013.

[124] H. A. Moghaddam, A. F. Farahani, K. Morrow, J. H. Ahn, and N. S. Kim. Near-dram acceleration with single-isa heterogeneous processing in standard memory modules. *IEEE Micro*, 36(1):24–34, 2016. doi:10.1109/MM.2016.8.

[125] S. F. Yitbarek, T. Yang, R. Das, and T. M. Austin. Exploring specialized near-memory processing for data intensive operations. In L. Fanucci and J. Teich, editors, *2016 Design, Automation & Test in Europe Conference & Exhibition, DATE 2016, Dresden, Germany, March 14-18, 2016*, pages 1449–1452. IEEE, 2016. URL: http://ieeexplore.ieee.org/document/7459537/.

[126] S. F. Yitbarek, T. Yang, R. Das, and T. M. Austin. Exploring specialized near-memory processing for data intensive operations. In *2016 Design, Automation & Test in Europe Conference & Exhibition, DATE 2016, Dresden, Germany, March 14-18, 2016*, pages 1449–1452, 2016.

[127] M. A. Neggaz, H. E. Yantir, S. Niar, A. M. Eltawil, and F. J. Kurdahi. Rapid in-memory matrix multiplication using associative processor. In *2018 Design, Automa-*

*tion & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018*, pages 985–990, 2018. doi:10.23919/DATE.2018.8342152.

[128] F. Schuiki, M. Schaffner, F. K. Gürkaynak, and L. Benini. A scalable near-memory architecture for training deep neural networks on large in-memory datasets. *IEEE Trans. Computers*, 68(4):484–497, 2019. doi:10.1109/TC.2018.2876312.

[129] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*, pages 105–117, 2015. doi:10.1145/2749469.2750386.

[130] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. M. Burns, and Ö. Özturk. Energy efficient architecture for graph analytics accelerators. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016*, pages 166–177, 2016. doi:10.1109/ISCA.2016.24.

[131] G. Li, G. Dai, S. Li, Y. Wang, and Y. Xie. Graphia: an in-situ accelerator for large-scale graph processing. In *Proceedings of the International Symposium on Memory Systems, MEMSYS 2018, Old Town Alexandria, VA, USA, October 01-04, 2018*, pages 79–84, 2018. doi:10.1145/3240302.3240312.

[132] Intel Corporation. Intel Stratix 10 MX (DRAM System-in-Package) Device Overview. Technical report, 2020. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-10/s10-mx-overview.pdf.

[133] Intel Corporation. Intel Arria 10 Device Overview. Technical report, 2020. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/arria-10/a10_overview.pdf.

[134] P. Faraboschi, K. Keeton, T. Marsland, and D. S. Milojicic. Beyond processor-centric operating systems. In G. Candea, editor, *15th Workshop on Hot Topics in Operating Systems, HotOS XV, Kartause Ittingen, Switzerland, May 18-20, 2015*. USENIX Association, 2015. URL: https://www.usenix.org/conference/hotos15/workshop-program/presentation/faraboschi.

[135] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler. Transparent offloading and mapping (TOM): enabling programmer-transparent near-data processing in GPU systems. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016*, pages 204–216. IEEE Computer Society, 2016. doi:10.1109/ISCA.2016.27.

[136] A. F. Farahani, J. H. Ahn, K. Morrow, and N. S. Kim. NDA: near-dram acceleration architecture leveraging commodity DRAM devices and standard memory modules. In *21st IEEE International Symposium on High Performance Computer Architecture, HPCA 2015, Burlingame, CA, USA, February 7-11, 2015*, pages 283–295. IEEE Computer Society, 2015. doi:10.1109/HPCA.2015.7056040.

[137] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, K. Hsieh, K. T. Malladi, H. Zheng, and O. Mutlu. Lazypim: An efficient cache coherence mechanism for processing-in-memory. *IEEE Comput. Archit. Lett.*, 16(1):46–50, 2017. `doi:10.1109/LCA.2016.2577557`.

[138] E. I. Vatajelu, P. Prinetto, M. Taouil, and S. Hamdioui. Challenges and solutions in emerging memory testing. *IEEE Transactions on Emerging Topics in Computing*, 7(3):493–506, 2019.

[139] L. Jiang, R. Ye, and Q. Xu. Yield enhancement for 3d-stacked memory by redundancy sharing across dies. In *2010 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 230–234, 2010.

[140] M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2011.

[141] J. Kubiatowicz. *Integrated shared-memory and message-passing communication in the alewife multiprocessor*. PhD thesis, Massachusetts Institute of Technology, 1998.

[142] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

[143] P. Tsigas and Y. Zhang. Integrating non-blocking synchronisation in parallel applications: performance advantages and methodologies. In *Proceedings of the 3rd international workshop on Software and performance*, pages 55–67. ACM, 2002.

[144] Z. Wei, P. Liu, R. Sun, and R. Ying. High-efficient queue-based spin locks for network-on-chip processors. In *Circuits and Systems (APCCAS), 2014 IEEE Asia Pacific Conference on*, pages 260–263. IEEE, 2014.

[145] H. Schweizer, M. Besta, and T. Hoefler. Evaluating the cost of atomic operations on modern architectures. In *Parallel Architecture and Compilation (PACT), 2015 International Conference on*, pages 445–456. IEEE, 2015.

[146] M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2011.

[147] J. S. Gray. *Interprocess communications in Linux*. Prentice Hall Professional, 2003.

[148] M. M. Michael and M. L. Scott. Implementation of atomic primitives on distributed shared memory multiprocessors. In *High-Performance Computer Architecture, 1995. Proceedings., First IEEE Symposium on*, pages 222–231. IEEE, 1995.

[149] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[150] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770, 1993.

[151] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, page 289–300, New York, NY, USA, 1993. Association for Computing Machinery. doi:10.1145/165123.165164.

[152] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The stanford dash multiprocessor. *Computer*, 25(3):63–79, 1992.

[153] Z. Wei, P. Liu, Z. Zeng, J. Xu, and R. Ying. Instruction-based high-efficient synchronization in a many-core network-on-chip processor. In *Circuits and Systems (ISCAS), 2014 IEEE International Symposium on*, pages 2193–2196. IEEE, 2014.

[154] X. Chen, Z. Lu, A. Jantsch, and S. Chen. Handling shared variable synchronization in multi-core network-on-chips with distributed memory. In *SOC Conference (SOCC), 2010 IEEE International*, pages 467–472. IEEE, 2010.

[155] Tilera Corporation. Tile Processor Architecture Overview for the Tile-Gx Series. Technical report, 2012. https://www.scribd.com/document/210769647/UG130-ArchOverview-TILE-Gx.

[156] M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *journal of parallel and distributed computing*, 51(1):1–26, 1998.

[157] G. Tian and O. Hammami. Performance measurements of synchronization mechanisms on 16pe noc based multi-core with dedicated synchronization and data noc. In *Electronics, Circuits, and Systems, 2009. ICECS 2009. 16th IEEE International Conference on*, pages 988–991. IEEE, 2009.

[158] M. M. Michael and M. L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *ACM Symposium on Principles of Distributed Computing*, pages 267–275, 1996. doi:10.1145/248052.248106.

[159] Y. Wang, R. Wang, A. Herdrich, J. Tsai, and Y. Solihin. CAF: Core to Core Communication Acceleration Framework. In *Conference on Parallel Architectures and Compilation (PACT)*, pages 351–362, 2016. doi:10.1145/2967938.2967954.

[160] S. Lee, D. Tiwari, Y. Solihin, and J. Tuck. HAQu: Hardware-Accelerated Queueing for Fine-Grained Threading on a Chip Multiprocessor. In *Conference on High-Performance Computer Architecture (HPCA)*, pages 99–110, 2011. doi:10.1109/HPCA.2011.5749720.

[161] S. Kumar, C. J. Hughes, and A. D. Nguyen. Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors. In *Symposium on Computer Architecture (ISCA)*, pages 162–173, 2007. doi:10.1145/1250662.1250683.

[162] R. R. Sharma, Y. Rajasekhar, and R. Sass. Exploring Hardware Work Queue Support for Lightweight Threads in MPSoCs. In *Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pages 1–6, 2012. doi:10.1109/ReConFig.2012.6416747.

[163] D. Sánchez, R. M. Yoo, and C. Kozyrakis. Flexible Architectural Support for Fine-Grain Scheduling. In *ASPLOS Conf. Proc.*, pages 311–322, 2010. doi:10.1145/1736020.1736055.

[164] J. Lee, C. Nicopoulos, H. G. Lee, S. Panth, S. K. Lim, and J. Kim. IsoNet: Hardware-Based Job Queue Management for Many-Core Architectures. *IEEE Trans. VLSI Syst.*, 21(6):1080–1093, 2013. doi:10.1109/TVLSI.2012.2202699.

[165] R. K. Pujari, T. Wild, and A. Herkersdorf. TCU: A Multi-Objective Hardware Thread Mapping Unit for HPC Clusters. In *High Performance Computing, ISC*, pages 39–58, 2016. doi:10.1007/978-3-319-41321-1_3.

[166] E. A. Brewer, F. T. Chong, L. T. Liu, S. D. Sharma, and J. Kubiatowicz. Remote Queues: Exposing Message Queues for Optimization and Atomicity. In *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 42–53, 1995. doi:10.1145/215399.215416.

[167] M. Mohr. *Aspects of Code Generation and Data Transfer Techniques for Modern Parallel Architectures*. PhD thesis, Karlsruhe Institute of Technology, Germany, 2018. URL: https://publikationen.bibliothek.kit.edu/1000085052.

[168] K. Nguyen, L. Fang, C. Navasca, G. Xu, B. Demsky, and S. Lu. Skyway: Connecting managed heaps in distributed big data systems. In *ACM SIGPLAN Notices*, volume 53, pages 56–69. ACM, 2018.

[169] M. Mohr and C. Tradowsky. Pegasus: efficient data transfers for pgas languages on non-cache-coherent many-cores. In *Proceedings of the Conference on Design, Automation & Test in Europe*, pages 1785–1790. European Design and Automation Association, 2017.

[170] I. A. C. Ureña, M. Riepen, and M. Konow. RCKMPI - lightweight MPI implementation for intel's single-chip cloud computer (SCC). In Y. Cotronis, A. Danalis, D. S. Nikolopoulos, and J. J. Dongarra, editors, *Recent Advances in the Message Passing Interface - 18th European MPI Users' Group Meeting, EuroMPI 2011, Santorini, Greece, September 18-21, 2011. Proceedings*, volume 6960 of *Lecture Notes in Computer Science*, pages 208–217. Springer, 2011. doi:10.1007/978-3-642-24449-0_24.

[171] S. Christgau and B. Schnor. Software-managed cache coherence for fast one-sided communication. In P. Balaji and K. Leung, editors, *Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM@PPoPP 2016, Barcelona, Spain, March 12-16, 2016*, pages 69–77. ACM, 2016. doi:10.1145/2883404.2883409.

[172] Cobham Gaisler AB. GRLIB IP Core User's Manual. Technical report, 2021. https://gaisler.com/products/grlib/grip.pdf.

[173] SPARC International, Inc. *The SPARC Architecture Manual, Version 8*, 1992. https://gaisler.com/doc/sparcv8.pdf.

[174] L. Masing, A. Srivatsa, F. Kreß, N. Anantharajaiah, A. Herkersdorf, and J. Becker. In-NoC Circuits for Low-Latency Cache Coherence in Distributed Shared-Memory

Architectures. In *12th IEEE International Symposium on Embedded Multicore/Many-core Systems-on-Chip, MCSoC 2018, Hanoi, Vietnam, September 12-14, 2018*, pages 138–145. IEEE Computer Society, 2018. doi:10.1109/MCSoC2018.2018.00033.

[175] M. A. Zaib. *Network on Chip Interface for Scalable Distributed Shared Memory Architectures.* Dissertation, Technische Universität München, München, 2018.

[176] A. Zaib, T. Wild, A. Herkersdorf, J. Heißwolf, J. Becker, A. Weichslgartner, and J. Teich. Efficient Task Spawning for Shared Memory and Message Passing in Many-Core Architectures. *Journal of Systems Architecture - Embedded Systems Design*, 77:72–82, 2017. doi:10.1016/j.sysarc.2017.03.004.

[177] J. Heißwolf. *A scalable and adaptive network on chip for many-core architectures.* Dissertation, Karlsruhe Institute of Technology, 2014.

[178] J. Heißwolf, A. Zaib, A. Weichslgartner, M. Karle, M. Singh, T. Wild, J. Teich, A. Herkersdorf, and J. Becker. The Invasive Network on Chip – A Multi-Objective Many-Core Communication Infrastructure. In *Conference on Architecture of Computing Systems (ARCS), Workshop Proceedings*, pages 1–8, 2014.

[179] PRO DESIGN Electronic GmbH. quad V7 Prototyping System. Technical report, 2014. https://www.profpga.com/files/profpga_quadv7_product_brief_2.pdf.

[180] Xilinx. 7 Series FPGAs Data Sheet: Overview. Technical report, 2020. https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf.

[181] B. Oechslein, J. Schedel, J. Kleinöder, L. Bauer, J. Henkel, D. Lohmann, and W. Schröder-Preikschat. OctoPOS: A parallel operating system for invasive computing. In R. McIlroy, J. Sventek, T. Harris, and T. Roscoe, editors, *Proceedings of the International Workshop on Systems for Future Multi-Core Architectures (SFMA)*, volume USB Proceedings of *Sixth International ACM/EuroSys European Conference on Computer Systems (EuroSys)*, pages 9–14. EuroSys, 2011.

[182] S. Gupta and V. K. Nandivada. Imsuite: A benchmark suite for simulating distributed algorithms. *J. Parallel Distrib. Comput.*, 75:1–19, 2015. doi:10.1016/j.jpdc.2014.10.010.

[183] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, et al. The nas parallel benchmarks. *The International Journal of Supercomputing Applications*, 5(3):63–73, 1991.

[184] J. Subhlok, S. Venkataramaiah, and A. Singh. Characterizing NAS Benchmark Performance on Shared Heterogeneous Networks. In *Parallel and Distributed Processing Symposium (IPDPS)*, 2002. doi:10.1109/IPDPS.2002.1015659.

[185] A. Faraj and X. Yuan. Communication characteristics in the NAS parallel benchmarks. In S. G. Akl and T. F. Gonzalez, editors, *International Conference on Parallel and Distributed Computing Systems, PDCS 2002, November 4-6, 2002, Cambridge, USA*, pages 724–729. IASTED/ACTA Press, 2002.

[186] Nipun Varma. *Adventures of an Indian Techie*. Notion Press, 2019.

[187] Scott Adams. *The Dilbert principle: a cubicle's-eye view of bosses, meetings, management fads & other workplace afflictions*. Harper Business, 1996.

[188] Coco Chanel. Brainyquote.com. `https://www.brainyquote.com/quotes/coco_chanel_119269`.

[189] H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Commun. ACM*, 10(8):501–506, August 1967. `doi:10.1145/363534.363554`.

[190] The Go authors. mbitmap.go, 2019. `https://go.googlesource.com/go/+/d41a0a0690ccb699401c7c8904999895b2c92511/src/runtime/mbitmap.go`.

[191] Oracle Corporation. instanceklass.hpp, 2019. `https://hg.openjdk.java.net/jdk10/jdk10/hotspot/file/5ab7a67bc155/src/share/vm/oops/instanceKlass.hpp`.

[192] J. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143 – 154, 1979. `doi:https://doi.org/10.1016/0022-0000(79)90044-8`.

[193] S. Gupta and V. K. Nandivada. Imsuite: A benchmark suite for simulating distributed algorithms. *J. Parallel Distrib. Comput.*, 75:1–19, 2015. `doi:10.1016/j.jpdc.2014.10.010`.

[194] John Carmack. Brainyquote.com. `https://www.brainyquote.com/quotes/john_carmack_776003`.

[195] S. Maier, T. Hönig, P. Wägemann, and W. Schröder-Preikschat. Asynchronous Abstract Machines: Anti-noise System Software for Many-core Processors. In *Proceedings of the 9th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*, pages 19–26. ACM, 2019. `doi:10.1145/3322789.3328744`.

# Appendix

## A Finite State Machine of the NCA-RO

Figure A.1 depicts the NCA-RO as a finite state machine (FSM) diagram. The individual states and state transitions are explained below in detail.

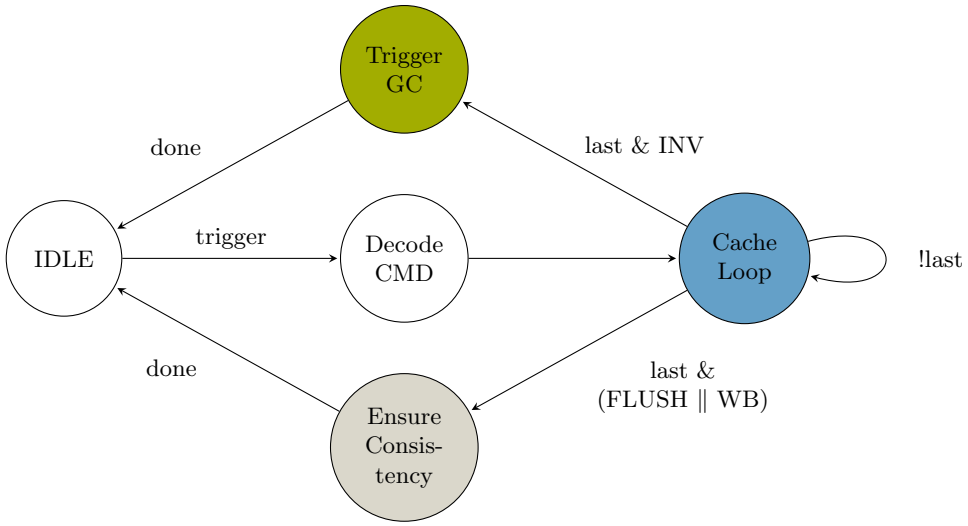



**Figure A.1: Finite State Machine of the NEMESYS NCA-RO.** The NCA-RO performs range based cache operations and is able to subsequently trigger the NMA.

**IDLE** When a trigger is received, the NCA-RO asks for arbitration to the CMD_Out bus interface and transitions to state Decode CMD.

**Decode CMD** The incoming command from a CPU is decoded and the parameters like start_addr, range_length, and mode (INV, WB, FLUSH) are stored internally. When the *invalidate-and-trigger* functionality is used, the parameters (cf. Appendix C) to trigger the NMA after the range invalidation are stored as well. Then, the Cache Loop state is entered.

**Cache Loop** This is the central state of the FSM, looping over every cache line touched by the range beginning with the start_addr and stretching over range_length bytes. For each affected cache line, the invalidation, writeback, or flush operation, defined by mode, is forwarded to the CMD_Out module, which performs the respective bus operations. As long as the last cache line is not reached, the FSM remains in this state. Once the last cache line is finished, three possible transitions, which depend on the mode, exist to the next state. If mode = INV and the NMA should be triggered, the next state is Trigger GC, else directly IDLE. However, if the mode is either WB or FLUSH, the FSM has to ensure that the writeback or flush operation has not only been initiated but that the respective cache line also reached the memory. This is required to

guarantee proper inter-tile memory consistency, which is handled in the state `Ensure Consistency`. It is not necessary for `mode = INV` since an L2 cache line invalidation does not produce any memory accesses.

**Ensure Consistency** A possibility to ensure that all cache lines reached the memory is to perform a dummy clean [1] read operation from memory on e.g., the last cache line of the range. Since the cache performs all operations in order and waits synchronously for their completion, this dummy read will only return from memory once all prior cache lines due to the writeback or flush commands have reached the memory. If `mode = FLUSH` [2], the cache line is invalidated implicitly so that an L2 miss occurs for the read operations. Ensuring that the read cache line will not be cached after the dummy read is guaranteed by invalidating it afterward. If `mode = WB`, the cache line is not necessarily invalidated, so an explicit invalidation before the dummy read operation is required. However, no final invalidation after the dummy read is required in this case. After these operations return, the memory consistency is ensured, and the FSM terminates to `IDLE`.

**Trigger GC** In this state, the `NCA-RO` FSM forwards the `NMA` trigger commands via the bus to the network adapter, which sends them to the `NMA`. The network adapter interface is the same as a CPU core would use when triggering the `NMA`. The parameters were received in the `Decode CMD` state. After the command is sent out asynchronously, the `NCA-RO` directly goes to `IDLE`.

# B Finite State Machine of the NCA-WB

Figure B.1 depicts the `NCA-WB` as an FSM diagram. The individual states and state transitions are explained below in detail. When a state transition does not contain a condition, it is implied that the transition to the next state only occurs once the current state's operation is completed.

**IDLE** When a trigger is received, the `NCA-WB` asks for arbitration to the `CMD_Out` bus interface and transitions to state `Decode CMD`.

**Decode CMD** The incoming command from a CPU is decoded and the parameters like the pointer to $G$ `root_G`, the pointer to the `metadata`, the pointer to the `visit stack`, and the descriptor to the `task_T1` are stored internally. Then, the graph writeback is initiated by entering the state `Get Object Info`.

**Get Object Info** First, the object information is obtained by reading the object's `RTTI` and stored internally. Most importantly, this includes the object layout specified by the pointer masks. Second, a magic word is written into the object's transition structure as a marker to indicate that this object has been visited. Then, the FSM transitions to the state `Range Writeback`.

**Range Writeback** This state has the same functionality as `Cache Loop` state of the `NCA-RO` for `mode = WB`. The `start_addr` and `range_length` are obtained from the object information and array/string descriptor. In general, the header and payload of an object

[1] A clean read from memory is a read operation that causes an L2 cache miss and therefore has to travel to the memory itself.

[2] Reminder: `mode = FLUSH` is a combination of invalidating and writeback, while `mode = WB` does not invalidate the cache line.

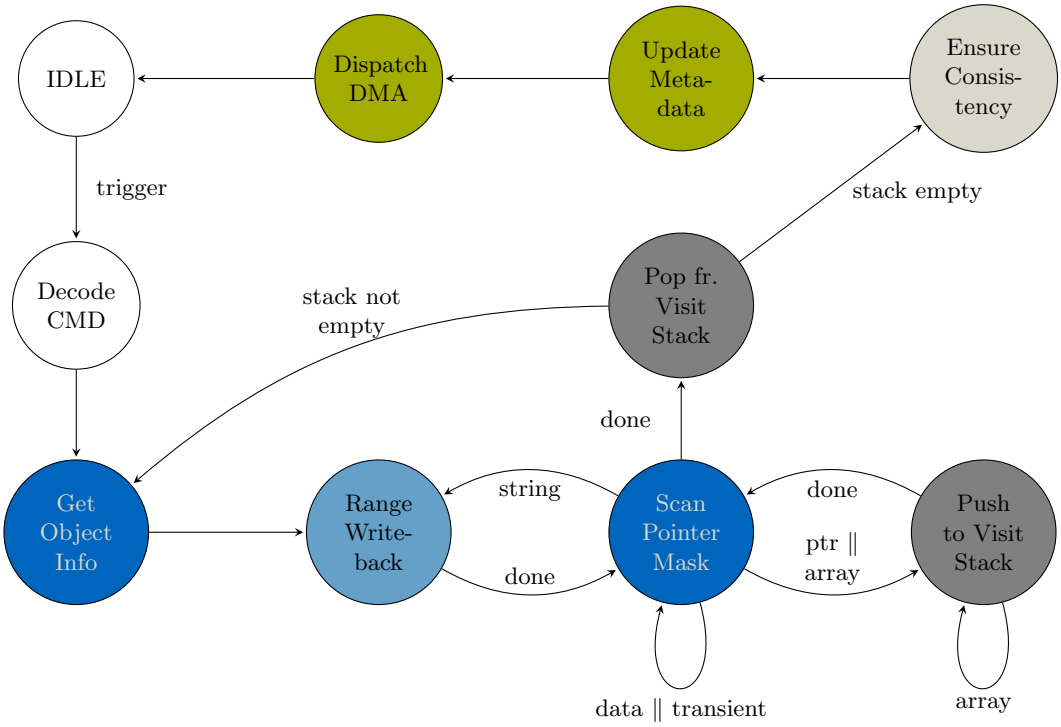



**Figure B.1: Finite State Machine of the NEMESYS NCA-WB.** The NCA-WB performs the graph writeback, appends the graph size to the metadata $M$, and initiates the DMA to tile $D$.

are written back first before the object is scanned for references to other objects or arrays/strings (cf. state `Scan Pointer Mask`). Upon completion of the range writeback, the next state is `Scan Pointer Mask`.

**Scan Pointer Mask** After the header and payload of the object have been written back in the previous state, this state iterates over all payload words and, based on the pointer mask, decides how to handle each payload word. In the case of a data (`00`) or transient (`10`) word, no action is required as they have already been written back during the range writeback. In the case of a string (primitive array, `110000`), a range writeback of the string is required. The string address and length are read from the array descriptor, and handed over as `start_addr` and `range_length` to the state `Range Writeback`. However, in the case of a pointer (`01`) or array (of pointers, `110101`), these references to the new objects have to be followed and pushed to the `visit stack` for subsequent processing once the current object is done. Thus, the FSM transitions to state `Push to Visit Stack`. From there, it always returns after the respective pointer has been handled. Once the entire object payload has been scanned, the current object is finished, and the FSM transitions to the state `Pop from Visit Stack`.

**Push to Visit Stack** This state differentiates between a pointer (`01`) and an array (of pointers, `110101`). In the case of an array of pointers, it iterates over all of them and performs the following steps for each of them. First, the FSM analyzes if the respective object has already been visited by checking if the marker in the transition structure has been set. If yes, this object is completed, and the FSM either returns to the state `Scan Pointer Mask` in case of a pointer (`01`) or continues with the pointer at the next index

of the array of pointers. However, if the marker is not set, the reference to the object has to be pushed to the visit stack to be handled later. Once all array elements are analyzed, the FSM returns to the state `Scan Pointer Mask`.

**Pop from Visit Stack** Upon completion of the previous object, the next object is popped from the visit stack if the stack is not empty. In this case, the FSM transitions to the state `Get Object Info` to start the next object's writeback process. However, if the visit stack is empty, all objects of the graph have already been handled, and the graph writeback is complete. Thus, the FSM goes to state `Ensure Consistency`.

**Ensure Consistency** This state has the same functionality as the state with the same name as the `NCA-RO` for `mode = WB`. It is sufficient to enter this state only once after the entire graph has been written back.

**Update Metadata** Next, the object count and the total graph size, measured during the graph writeback operations, are appended to the metadata structure. A pointer to this structure is provided when the `NCA-WB` is triggered.

**Dispatch DMA** Finally, the `NCA-WB` FSM initiates a DMA transfer of the metadata $M$ with subsequent task invocation of `task_T1` on the receiver tile. This is done by writing the respective DMA descriptor (`src =` $M$`, dst =` $M'$`, length = sizeof(`$M$`)`) followed by the four-word task descriptor `task_T1` via the bus into the network adapter control interface. This network adapter interface is the same as a CPU core would use to trigger a DMA or task invocation. In case of the advanced inter-memory mechanism (cf. Section 5.6.1.2), a slight modification is required. Instead of the invocation of `task_T1` on the receiver tile, the `NCA-WB` invokes `task_T0` on the sender tile itself. After the commands are sent out asynchronously, the `NCA-WB` directly goes to `IDLE`.

# C Finite State Machine of the NMA-GC

Figure C.1 depicts the `NMA-GC` as an FSM diagram. The individual states and the state transitions [3] are explained below in detail.

**IDLE** When a trigger is received, i.e., the FIFO of incoming requests is not empty, the `NMA` transitions to state `Decode CMD`.

**Decode CMD** The incoming command contains several parameters: the pointer to the source graph $G$, the pointer to the pre-allocated destination buffer $G'$, the pointer to the statically allocated `copy map`, and the task descriptor of the completion task, which is sent to the receiver tile upon completion of the graph copy. Note that both the standard intra-memory graph copy from $G$ directly to $G'$ and the advanced inter-memory graph copy from $G$ via $G^*$ to $G'$ use the same FSM. Solely, the pointer to the pre-allocated intermediate buffer $G^*$ has to be provided. Additionally, the copy map length must be provided if the hash map variant is used. In this case, the first half of the hash map needs to be zeroed, which happens in the state `Memset Hmap`. Otherwise, after decoding the trigger command, the graph copy is directly initiated by entering the state `Get Object Info`.

[3] When a state transition does not contain a condition, it is implied that the transition to the next state only occurs once the operation to the current state is completed. This increases the readability.

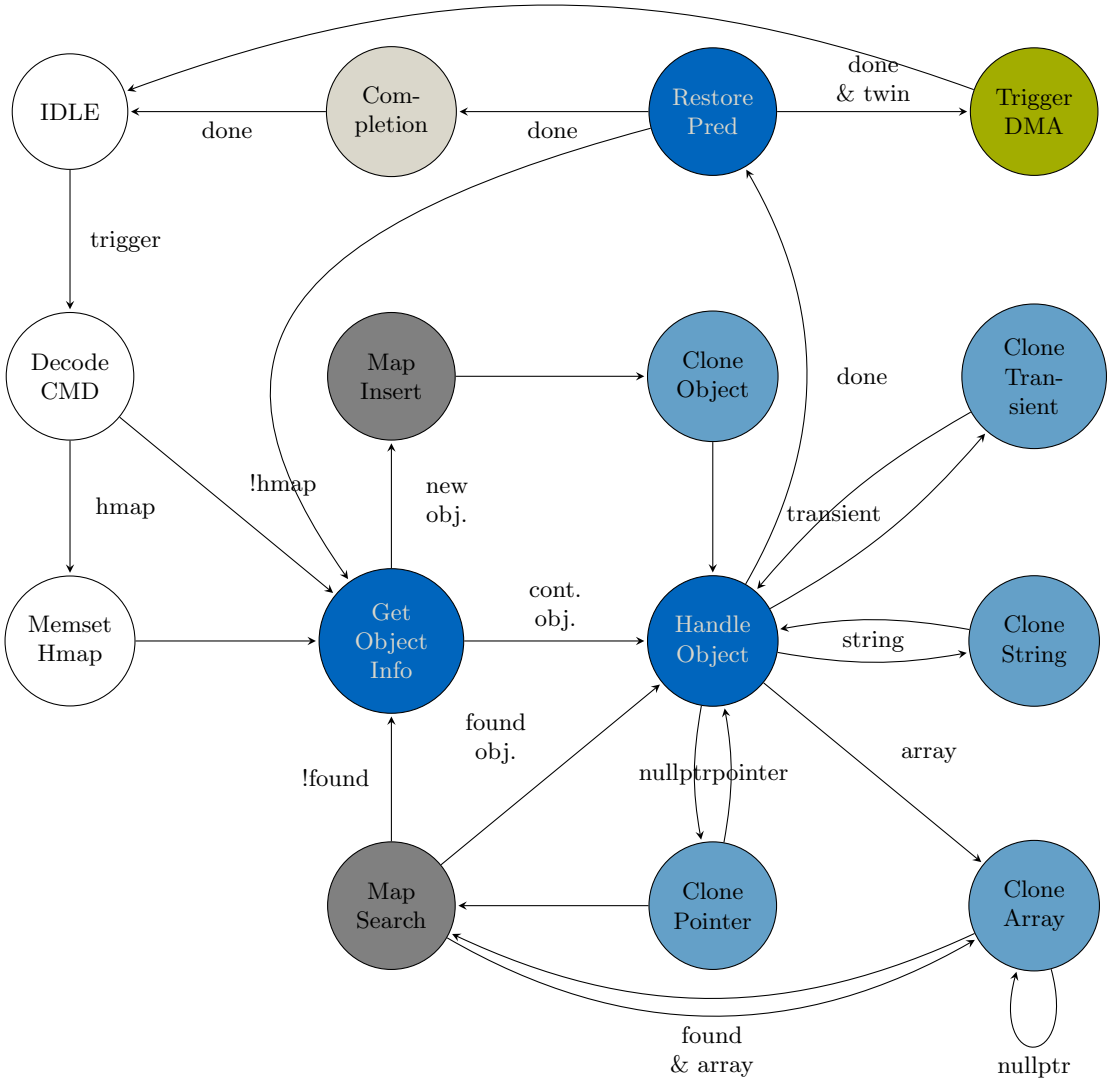



**Figure C.1: Finite State Machine of the NEMESYS NMA.** The NMA performs the graph copy operations and is able to subsequently send the completion task to tile $D$ or, in case of the advanced inter-memory graph copy, initiate the DMA of $G^*$ to $G'$.

**Memset Hmap** The hash map variant of the copy map requires its first half to be zero-initialized. The copy map's actual length is determined beforehand (during the graph writeback) and passed to the NMA so that the NMA initializes only the necessary range of the copy map to save time in the critical path of execution. Afterward, the graph copy is initiated by entering the state `Get Object Info`.

**Get Object Info** This state is visited whenever an object $A$ is reached that has not yet been started, when retreating from a child object $B$, or when handling the next element in an array of pointers that has not been found in the copy map. First, the object information is obtained by reading the object's RTTI and stored internally. Most importantly, this includes the object layout specified by the pointer masks and the object size. Second, the transition structure is read to identify the current position (`offset`) inside the

object, and in the case of an array of pointers, the current array `index`. If it is a new object (`offset = 0`), it has to be inserted into the `copy map`. Thus, the FSM transitions to the state `Copy Map Insert`. However, if it is an object already started (`offset > 0` or (`offset = 0 & index > 0`)), it will be continued in the state `Handle Object`. In this case (i.e., when coming from state `Restore Pred`), the pointer $B'$ to the clone of the child object will now be written at the corresponding `offset` in $A'$.

**Copy Map Insert** This state has twofold functionality. First, the new object $A'$ has to be allocated via the hardware internal `bump allocator` ($A'$ = `bump_allocate(object_size)`). Second, the mapping $A \mapsto A'$ has to be inserted into the `copy map`, either at the next free position in case of the linear search implementation or at the position `hash(A)` of the hash map. After that, the FSM transitions to the state `Clone Object`.

**Clone Object** The next couple of states handle the actual copy process. First, in this state, the entire object (header and payload) is copied one-to-one with a burst memory read and write access of length `object_size`. References to other objects or arrays are not followed yet and will still point to the objects in the source graph. They will be updated when retreating from the respective child objects. After copying the current object's header and payload, the FSM transitions to the state `Handle Object`. Thus, at this point, only the header and the data words (`00`) of the payload are copied correctly.

**Handle Object** This state iterates (`offset++`) over all the payload words of $A$ and, based on the pointer mask, decides how to handle each payload word. If the word at the current `offset` is a transient word (`10`), the state `Clone Transient` is chosen next. If a pointer to a string is detected (`110000`), the FSM transitions to the state `Clone String`. In case of a pointer (`01`) the state `Clone Pointer`, and in case of an array (`110101`) the state `Clone Array` is targeted. Before advancing to any of the `Clone Pointer` or `Clone Array` state, the transition information is stored in the transition structure of $A'$ for later retrieval. Once the end of the object is reached (`offset = object_size`), the FSM initiates the restoring of the predecessor (i.e., parent object) via the state `Restore Pred`.

**Clone Transient** A transient word does not need to be copied but set to zero instead. Thus, the `NMA` write a zero at the appropriate `offset` in the object $A'$. Then, the FSM returns to `Handle Object`.

**Clone String** A string is a contiguous address range characterized by the `array_ptr` and `array_length`. It is copied by simply issuing a burst read and write of size `array_length` from address $S$ = `array_ptr` to $S'$. $S'$ is obtained by calling the `bump allocator`: $S'$ = `bump_allocate(array_length)`. After copying the string, the FSM returns to `Handle Object`.

**Clone Pointer** If the pointer found is a `NULL` pointer, the FSM directly returns to state `Handle Object`, since it remains a `NULL` pointer and thus, it has already been copied correctly in the state `Clone Object`. Otherwise, it has to be determined if the object $B'$, referenced by this pointer $B$, has already been copied. Thus, a lookup of $B$ in the `copy map` is required, and the FSM transitions to the state `Copy Map Search`.

**Clone Array** An array of pointers is characterized by the array descriptor (`array_ptr`, `array_length`, and `array_size`), which is read out as a first action in this state. Next, the memory for the array backing store is allocated via the `bump allocator`. Then, this state iterates (`index++`) over the entire array using the `index` field of the

transition structure. If an array element is a `NULL` pointer, it is skipped as above. Otherwise, a lookup of $B$ in the `copy map` is required, and the FSM transitions to the state `Copy Map Search`. Once the entire array is handled, the FSM returns to state `Handle Object`.

**Copy Map Search** This state searches the first half of the `copy map` for the respective pointer $B$ (either linear search or hash map search depending on the configuration defined by the trigger command). When the pointer $B$ is found, the corresponding clone $B'$ is looked up at the same offset in the second half of the `copy map`. The pointer $B'$ is inserted at the appropriate `offset` in the object $A'$. Then, the FSM transitions to the state `Handle Object` and continues to clone object $A$ if it is not an array. In the case of an array, it directly returns to `Clone Array` and continues with the next array element. However, when the pointer $B$ is not found in the `copy map`, it is pointing to a new object which has to be copied from the beginning. In this case, the FSM goes to the state `Get Object Info`.

**Restore Pred** Once an object $B$ is cloned completely (`offset = object_size`), this state restores the reference to its predecessor object $A$ and the clone $A'$ thereof. It is retrieved from the transition structure of $B'$: $A$ = $B'$`.parent_src` and $A'$ = $B'$`.parent _dst`. $A'$ is available in the transition structure as well to avoid an extra lookup in the `copy map`. Then, the FSM retreats to the state `Get Object Info` to continue with the predecessor object. If the predecessor $B'$`.parent_src = NULL`, the root object is reached and the entire graph is cloned completely. In the standard intra-memory graph copy, the FSM goes to state `Signal Completion`. In the case of the advanced inter-memory graph copy, the FSM detours to the state `Trigger DMA`.

**Signal Completion** In this state, the completion task `task_T1` is sent out asynchronously to a core on the destination tile. Then, the FSM directly returns to `IDLE`.

**Trigger DMA** When the advanced inter-memory graph copy is performed, the `NMA` cloned the graph $G$ to the intermediate buffer $G^*$ so far. The finalization of the graph copy to $G'$ requires a DMA transfer of $G^*$ to $G'$. The DMA trigger is sent to the network adapter accompanied by the descriptor of `task_T1` which will be scheduled on the destination tile upon completion of the DMA. Thus, the state `Signal Completion` can be omitted in this case, and the FSM directly goes to `IDLE` to handle the next graph copy request.