TECHNISCHE UNIVERSITÄT MÜNCHEN

Lehrstuhl für Integrierte Systeme

Image Processing on Heterogeneous Multiprocessor System-on-Chip using Resource-aware Programming

Johny Paul

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

Doktor-Ingenieurs (Dr.-Ing.)

genehmigten Dissertation.

Vorsitzende(r): Prof. Dr.-Ing. Ulf Schlichtmann

Prüfer der Dissertation:

1. apl. Prof. Dr.-Ing. Walter Stechele
2. Prof. Dr.-Ing. Tamim Asfour

Die Dissertation wurde am 28.02.2017 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 29.06.2017 angenommen.

To my family.

Abstract

Multiprocessor system-on-chip (MPSoC) designs offer a lot of computational power assembled in a compact design. The computing power of MPSoCs can be further augmented by adding massively parallel processor arrays (MPPA) and specialized hardware with instruction-set extensions. On-chip MPPAs can be used to accelerate low-level image-processing algorithms with massive inherent parallelism. With the arrival of new **heterogeneous MPSoCs**, various applications that run on dedicated compute units on humanoid robots like **ARMAR**, can now be combined and executed on a single-chip.

However, the presence of multiple processing elements (PE) with different characteristics raises issues related to programming and application mapping. The conventional approach used for programming heterogeneous MPSoCs results in a static mapping of various parts of the application to different PE types, based on the nature of the algorithm and the structure of the PEs. Yet, such a mapping scheme independent of the instantaneous load on the PEs may lead to under-utilization of some type of PEs while overloading others. At the same time it is difficult to dynamically control and distribute different types of resources among different applications running on a single chip, achieving high resource utilization under high-performance constraints.

This thesis focus on investigating the benefits and challenges of using a resource-aware programming model called *Invasive Computing* for dynamically mapping image-processing applications (used on humanoid robots) to different types of PEs available on a heterogeneous MPSoC named *InvasIC*. Different types of computer vision algorithms were investigated and implemented on the heterogeneous MpSoC. Based on the new model, the applications can dynamically acquire and release hardware resources, considering the level of parallelism available in the algorithm and time-varying load. Our results indicate that resource-aware programming helps to improve the **performance** (throughput and worst observed latency) of various computer vision applications along with better overall **workload distribution** within the heterogeneous MPSoC.

Acknowledgment

I would like to express my gratitude to everyone, who have supported me in successfully realizing this thesis. First of all, I want to express my deep gratitude towards my advisor, Prof. Dr.-Ing. Walter Stechele for his exceptional support throughout my work at LIS, particularly for the freedom offered to me during my research and also for the constant feedback that I received. I am also grateful to Prof. Dr.-Ing. Tamim Asfour (our partner from the Karlsruhe Institute of Technology) for the feedback and suggestions I received during the past 5 years. Furthermore, I would like to thank Prof. Dr. sc.techn. Andreas Herkersdorf and Dr.-Ing. Thomas Wild for their guidance, support and suggestions during my work in the project Invasive Computing.

Finally I want to thank all my colleagues, especially my fellow doctoral candidates Aurang Zaib, David May and Ravi Kumar Pujari for the cooperation and fruitful discussion we had over the past several years. Additionally I would like to thank my colleagues outside LIS, especially from Karlsruhe Institute of Technology and University of Erlangen-Nuremberg, for providing necessary support at various levels including hardware and operating system layers. I would also like to thank the students who have contributed to my work with their Diplom- and master-thesis at LIS. Last but not the least, my gratitude goes towards my family, for their support during my stay and work in Germany.

Munich, November 2015
Johny Paul

Contents

List of Figures

List of Tables

Index of Algorithms

1. Introduction

Robotics, especially **humanoid robotics** is a challenging research field, which has received great attention during the past decade and will continue to play a central role in the 21st century. Humanoid robots can perform several tasks resembling humans. The technological progress in the fields of computational power and mechatronics has led to the development of humanoid robots which are rich in sensory and motor capabilities and hence provide a platform to study emergent cognitive capabilities in artificial systems. Cognitive humanoid robots will be able to operate in an autonomous way, interacting naturally with the world and with human users, and be self-adaptive to changing situations and contexts. This includes assisting humans in the day-to-day work, as helpers in natural disasters or as entertainment robots such as soccer players. Under the hood, the robotic system are typically complex with sensors to gather data, processing systems which analyze the input data and enables the robot to made decisions followed by the motors and actuators to put the decision into actions.

Humanoid robots are designed to operate in little or unknown environments. A detailed understanding of these environments is required to navigate through them. Information about the environment could be provided by a **computer vision** system, acting as a vision sensor, providing high-level information about the environment. The data processing systems used in today's humanoid robots consists of several industrial PCs, dedicated DSPs/FPGAs, etc. Such a complex and elaborate system is necessary to handle the complex computer vision algorithms that exists today.

The availability of new processor designs with higher and higher computing power (computational density) has helped the robot designers to handle more complex tasks thereby improving the performance and behavior of robots. Take the clock rate, for instance: the Intel 4004 micro-processor ran at a clock frequency of 108KHz. In a mere eight years, the Intel 8088 could run at 5MHz, almost 50$\times$ speed boost. A modern 3GHz CPU has a clock rate around $30,000\times$ faster than the 4004 microprocessor. The increase in the transistor count, another factor indicating the CPU power, has been even more spectacular. The Intel 4004 contained just about 2,300; the 8088 with more than 29,000; and a modern high-end Intel Xeon chip with transistor count more than 2 billion: a million times more than that of the first CPU [15].

This trend is expected to continue, providing integration capacity of billions of transistors on to a single chip. Until now, Moore's Law continued with technology scaling, improving transistor performance to increase frequency, increasing transistor integration capacity to realize complex architectures and reducing energy consumed per logic operation to keep power dissipation within limit. However, the semiconductor technology has already hit the power wall and is not far away from hitting the utilization wall [16]. These effects are caused by shrinking technology, which continuously leads to higher energy densities. Thus, these days, energy efficiency has become more important than pure computing power. This means that in order to scale computing performance in the future, energy efficiency has to be significantly improved.

Moreover, the performance increase by micro-architecture alone is governed by Pollack's Rule, which states that performance increase is roughly proportional to square root of increase in complexity [17]. In other words, if the logic in a processor core is doubled, it can deliver only 40% more performance. A multi-core micro-architecture, on the other hand, has potential to provide near linear performance improvement with complexity and power. Two smaller processor cores, instead of a large monolithic processor core, can potentially provide 70-80% more performance, as compared to only 40% from a large monolithic core [18]. Multiprocessors have several other benefits as each processor core can be individually turned on or off, thereby saving power. New technology allows each processor core to be run at its own optimized supply voltage and frequency, thereby saving power. It is also easier to load balance among processor cores to distribute heat across the die and can potentially produce lower die temperatures improving reliability and leakage.

Computer-vision algorithms can highly benefit from processor architectures with numerous processing elements as they are compute-intensive and typically expose massive inherent parallelism. Multiprocessor system-on-chip (MPSoC) designs offer a lot of computational power assembled in a compact design. In mobile robotic applications, they offer the chance to replace several dedicated computing boards by a single processor, which typically leads to a significant acceleration of the computer-vision algorithms employed. This enables robots to perform more complex tasks at lower power budgets, less cooling overhead and, ultimately, smaller physical dimensions. MPSoC designs can achieve higher energy efficiency and performance scalability by increasing or decreasing the number of cores in use according to processing requirements. In addition to homogenous processors, the design of heterogeneous hardware with customized resources, such as accelerators dedicated for one application domain, is a promising solution to address the challenges in the field of robotics. Because of these features, MPSoCs are ideal for advanced embedded applications requiring minimal power consumption, including robotics.

1.1. Outline of the Thesis

This thesis is organized as follows. Chapter 1 presents the humanoid robot ARMAR, with focus on computer vision algorithms, compute units, etc. This chapter also describes the potential challenges and expected benefits in using a heterogeneous MPSoC like InvasIC, on a humanoid robot. Chapter 2 describes some of the existing homogeneous/heterogeneous MPSoCs and explores some of the conventional techniques used for application mapping on such architectures. This is followed by Chapter 3 which describes the proposed resource-aware programming model named Invasive Computing and the architecture of the heterogeneous MPSoC named InvasIC. The novel operating system for InvasIC is also described in this chapter. Chapter 4 describes the algorithms behind three widely used computer vision algorithms including Harris Corner detection, SIFT feature extraction and matching and optical flow. These algorithms are later used as candidates for evaluating the resource-aware programming model.

Chapter 5 presents two novel computer vision algorithms developed as part of this thesis, based on the resource-aware programming model. The results point to the benefits of using Invasive Computing on a homogeneous instance of InvasIC. Chapter 6 demonstrates how the algorithms described in Chapter 4 can be accelerated using specialized PEs available on InvasIC, followed by Chapter 7 describing the execution of an object recognition application on a heterogeneous MPSoC using the resource-aware model, presented in Chapter 3. Finally, Chapter 8 concludes the thesis with a outlook into the phase-II of the Invasive Computing project.

1.2. Image Processing on Humanoid Robots

Robots will have a broad impact across many existing and emerging markets, which belong to the industrial, service, domestic, security and space robotics sector [19]. However, for building robotic computer architectures, usually only off-the-shelf hardware components are used, which do not result in dedicated solutions in terms of efficiency, on-board computational power, memory organization, miniaturization, communication and multi-purpose I/Os.

Computer architectures in humanoid robots are often based on standard embedded industrial PCs, PC/104 systems for high-level tasks and dedicated DSP/FPGA units for low-level control (see, e. g., [20, 21, 22, 23, 24, 25, 26, 27, 28]). The work presented in [27, 28] described a control architecture for the humanoid robot ARMAR-III (Figure 1.1) developed at KIT and its hardware implementation with embedded Industrial PCs, PC/104 systems and DSP units. Using these compute units, several sophisticated interactive service tasks have been realized as described in [28, 29, 30].

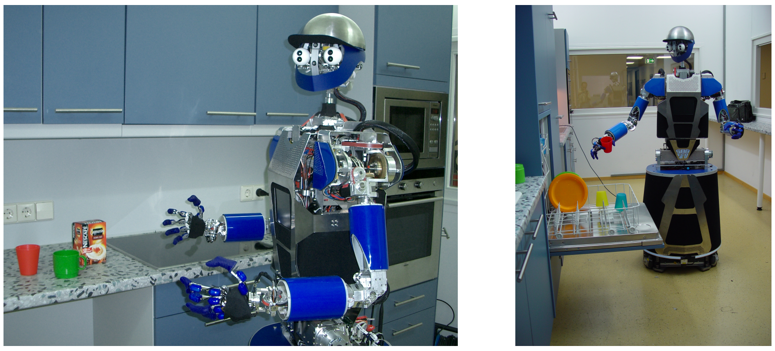


Figure 1.1.: The humanoid robot ARMAR-III with 43 degrees of freedom. The robot consists of an active head with cameras for vision, two arms and two five-fingered hands, a torso and a holonomic mobile platform [1]

Regarding computer vision, although a broad variety of computer vision algorithms exists today [31, 32, 33], they are mostly not optimized for real-time embedded systems. Many robotic platforms use FPGAs even though microcontrollers can successfully implement any digital logic function. FPGAs offers massive parallelism and can accelerate compute intensive tasks which cannot be efficiently handled by today's single/multi-core processors. On FPGAs, an algorithm can be implemented sequentially or completely in parallel, depending on the performance requirements. A completely parallel implementation is faster but consumes more FPGA resources. However, microcontrollers are easy to program and debug, well suited to control applications, especially with widely changing requirements.

Similar to the FPGAs, a lot of interest has been rising on Graphics Processing Units (GPU) [34, 35, 36]. While GPUs have been used traditionally to accelerate computer games, their massively parallel and light weight execution resources can be exploited in many computer vision applications, thus challenging the domain of FPGAs for hardware acceleration. In [34, 36], various case-studies have been presented comparing GPU vs. FPGA implementations. There seems to be no "one fits all" solution, but the benefits and drawbacks are manifold: GPUs benefit from higher clock frequencies than FPGAs, due to their custom layout. GPU programming does not require VHDL experience, but can be done by software developers, instead. On the other hand, programming GPUs efficiently can be as tedious as hardware development in VHDL. FPGAs offer a higher degree of freedom in optimizing memory architectures, data types and pipeline structures. However, the core problem remains for both GPU and FPGA: How to exploit parallelisms in the algorithm and distribute computations and data access on available resources?

In addition to FPGAs and GPUs, DSPs also offer certain benefits in the field of robotics. DSP is a type of flexible and reliable microprocessor - one that is incredibly fast and powerful, it can process data in real time, its real-time capability and high-speed computing architecture is an optimal alternative for sophisticated control algorithms. So, the integration of FPGA and DSP can perform a lot of computing-intensive and time-critical tasks such as information acquisition and data procession in robot control applications [37]. A high performance DSP/FPGA controller has been used in the implementation of the DLR dexterous robot hand [38].

In brief, the computer vision algorithm can benefit from various types of processing units including CPUs, FPGAs, GPUs and DSPs. In case of conventional robots, this leads to a complex processing hierarchy with separate boards for each type of compute unit. The use of multiple compute units results in high power consumption and low interconnect bandwidth, and occupies a large amount of space in the robot. The data exchange between compute units make the overall implementation complex and each unit with its own power supply also consume significant fraction of the power budget. Moreover, in the current implementation with multiple CPU-boards, it is not possible to migrate tasks from on CPU to the other. This means that the vision algorithm will not be able to use the processing units dedicated for motion planning or audio processing, even when the motion planning or audio processing systems are in an idle state. Such a mapping scheme often leads to under utilization of the available computing resources. Similar problems arise in modern computer architectures (Cell-Blade from IBM) that have been investigated for robotic vision applications by the proposers. Efficient low-level vision algorithms could be be implemented on such architectures. However, robotics applications include a wide variety of such low-level algorithms, which require higher degree of reconfigurability concerning load balancing, communication paths and bandwidth, as well as memory access.

1.3. Multiprocessor System-on-Chip (MPSoC) Designs

As technology scales further, transistor performance will not increase at the historic rates, due to excessive sub-threshold leakage current and supply voltage scaling slowing down [39]. This means that the path of multi-core evolution by integrating multiple complex cores on a die will not be as easy as it used to be. Integrating lots of smaller cores, where each small core delivers lower performance than a large complex core, can lead to a higher compute throughput for the entire system. The performance of a smaller core reduces as square-root of the size (inverse Pollack's rule), but power reduction is linear, resulting in smaller performance degradation with much larger power reduction. Overall, the compute throughput of the system, increases almost linearly with the larger number of small cores. Based on this idea, several multiprocessor systems-on-chips (MPSoCs) have started to enter the marketplace over the past several years and are expected to be available in even greater variety over the next few years. Examples include Tilera's 64-core processor [6], Ambric (with 336 32-bit RISC processors) [40], an 80-core Intel prototype processor [41], Xetal-II [6] (a SIMD processor with 320 processing elements). In MPSoCs, a spatial choice is added to the temporal choice of resources; applications can share a resource and wait on each other, or they can run simultaneously with independent resources.

An MPSoC system does not necessarily have to be symmetric or homogenous. An asymmetric system may have a few large cores that can deliver higher single-thread performance, but will predominantly have large number of small cores. Such a heterogeneous system can even integrate diverse special purpose cores for hardware acceleration like graphic engines (GPUs), CPUs with hardware extensions which can be accessed using special instructions, mixed-signal circuits, etc., to provide a complete integrated system. AMD's Accelerated Processing Unit (APU) [42] is an example for heterogeneous MPSoC which combine GPU and CPU on a single chip. The low-level pixel-processing algorithms (with regular loops operating on a stream of pixel data) can be mapped to the GPU while the CPU can post-process the results computed by the GPU. Static mapping of sub-algorithms on architectures like ADM APU seems feasible, but compilation support for future massively parallel MPSoCs is not in sight under hard performance/power/area constraints.

1.4. Benefits of using MPSoCs on Robots

Humanoid robots are required to handle various tasks like vision, motion planning, speech recognition, or speech synthesis. To be autonomous, humanoid robots should be able to learn to operate in the real world and to interact and communicate with humans. They have to model and reflectively reason about their perceptions and actions in order to learn, act, predict and react appropriately. Performing these tasks in the real world, especially in real-time, demands not only substantially high computing power but also concurrent hardware and software architectures that support situation-dependent context switching between different applications such as natural dialogue management, visual perception of the environment, situation interpretation and motion planning, as well as action execution. Such architectures should allow to process the large amount and variety of sensory data in parallel and allocate resources appropriately.

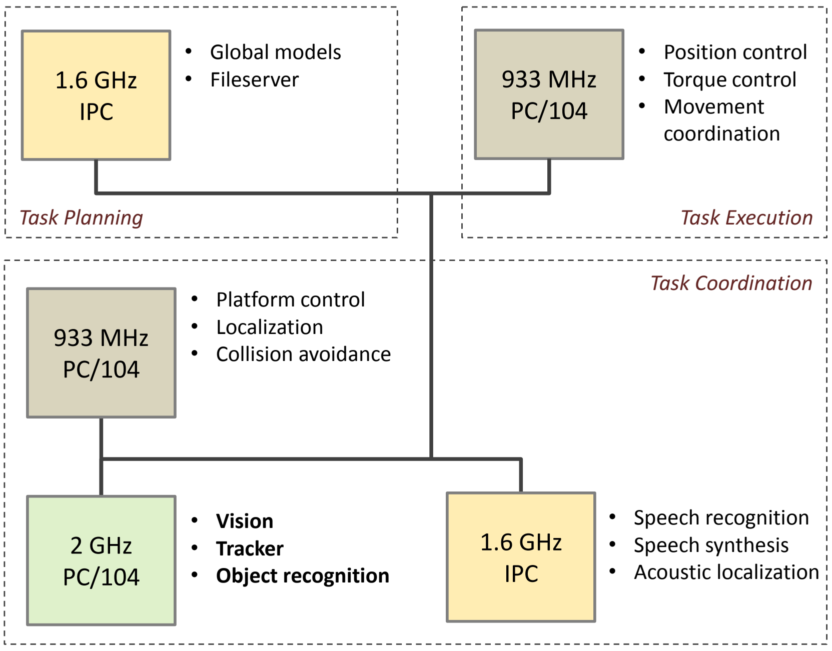


Figure 1.2.: Compute units on the ARMAR-III robot [1]

Usually, on a humanoid robot, the workload is spread across multiple industrial CPU boards. For example, on the ARMAR-III robot [43], data from the sensors flows into the respective processing system, each dedicated to a different task, like computer vision, low-level control, high-level control or speech processing, as shown in Figure 1.2.

Similar to the ARMAR-III design, there are three PC/104 computers in the ARMAR-4 robot (the next generation of the humanoid robot following ARMAR-3) [2]. Two are positioned in the torso and one in the head. The PC/104 in the head is equipped with an Intel Core2Duo processor and a combined audio and Firewire board, as shown in Figure 1.3. It is used for computer vision and sound processing. In the torso there are two PC/104s. One is equipped with an Intel Core i7 processor and is performing the high level planning and control, the other one is equipped with an Intel Core2Duo processor and a PCI/104 10 channel CAN interface board, running a real time operating system and is used for low level control and communication with the sensors and motor controllers [2]. Other examples with a similar architecture include the humanoid robot Asimo [44], HRP-4C [45] and the Hand Arm System from DLR [46].

Substituting the numerous processing systems with a single heterogeneous MPSoC can bring in several benefits. The heterogeneous MPSoC with its massive computing power can handle the entire workload previous handled by several independent processing units. This means the various applications that, on a conventional robot, run on dedicated compute units can now be combined and executed on a single-chip. The GPU friendly algorithms can be mapped to the on-board GPU while algorithms with huge number of mathematical operations can be mapped to special hardware with support for special instructions. For example, the specialized hardware units (like on board GPUs) can be used to accelerate low lower pixel processing algorithms like Harris corner detector and audio filtering. As all applications now run on a unified platform with several cores, load balancing becomes

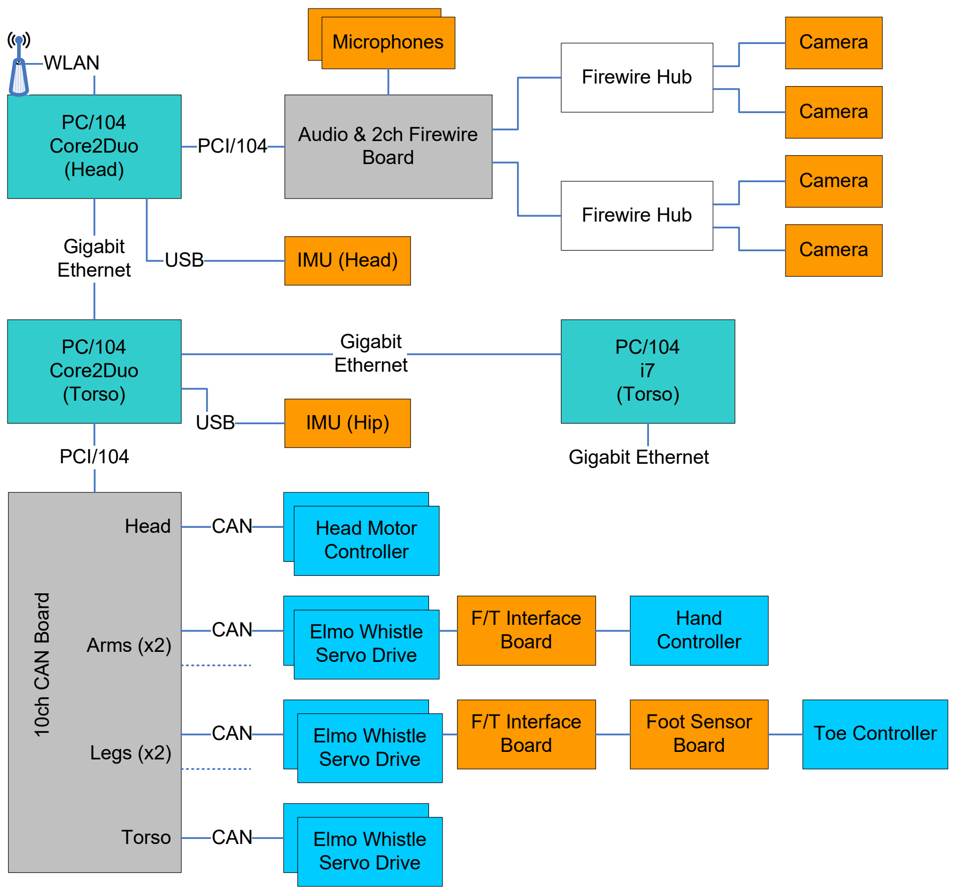


Figure 1.3.: Compute units used on the ARMAR-4 robot [2]

easier. The tasks can easily migrate from one core to the other. For example, the cores used for audio processing or motion planning can be used for vision algorithms when the audio or motion planning is in a idle state. Such a scheme also helps to save space on the robot as multiple CPU boards can be replaced with a single board.

Various compute units on the MPSoC are interconnected using a Network-on-Chip (NoC) interconnect, which offers significant bandwidth for data exchange. Such a high bandwidth, low latency interconnect structure is important to ensure that the data can flow easily and without delay from one algorithmic stage to the next. Moreover, an MPSoC with advanced power management scheme can shutdown its compute/communication/memory resources when its not in use. This helps to reduce power consumption of the entire processor and thereby reducing the cooling costs and saves valuable battery power in case of untethered robots.

1.5. Potential Challenges in using MPSoCs on Robots

On account of their immense computational power assembled in a compact design, the heterogeneous MPSoCs can bring in several benefits to the field of robotics, as described in

Section 1.4. However, in a heterogeneous systems the resource allocations becomes even more complex. Each application has a preferred type of resource on which it ideally should be mapped. But it can, if necessary, run with another type of resource as well. Hence, a full design-time exploration is not even possible, since not all software a user may run on a system is known at design-time. Moreover, the mapping of various tasks onto a unified MPSoC automatically results in sharing of physical resources like processing elements (PEs) and interconnect/memory bandwidth between the once separated tasks. Applications like pattern recognition, motion planning, speech synthesis, etc. have varying demands on computing power, depending on the situation the robot is facing or interacting with. These applications create dynamically changing load on the processor based on what the robot is doing at the current point in time. For instance, the speech-recognition application is activated whenever the user speaks to the robot and the motor control is activated when the robot has to move or grasp an object that it recognized.

As stated by the Intel fellow Shekhar Borkar, "The software has to also start following Moore's Law, software has to double the amount of parallelism that it can support every two years"', in order to make efficient use of the future multi-core designs. A conventional operating-system (OS) scheduler schedules the numerous threads of each application considering the overall system load. As a result, the resources available to each application may vary over time, leading to a further issue of *guaranteed execution quality*. For example, the object recognition algorithm is programmed to run at 25 frames per second (fps) and needs a minimum of 16 PEs on the MPSoC. However, at various points in time, fewer PEs were offered by the runtime system, leading to frame drops as the algorithm is running on real-time video input. Frame drops may reduce the overall accuracy of the algorithm.

Restricted power budgets have made researchers aware of the problem of *dark silicon* which implies that only a certain percentage of the available processors may be active at the same time. Hence, it is essential to make the best use of the available PEs on the heterogeneous processor. For example, the low level pixels processing algorithms like Harris corner detector may run more efficiently on a graphics processor when compared to its CPU counterpart, thereby reducing the overall power consumption of the entire chip. Another scenario arises if the graphics processor is heavily loaded and the Harris corner algorithm cannot wait indefinitely for the resources to be available. How to make these algorithm adapt at run-time so that they can choose the best available resources at any instance in time? Hence, programming such large-scale processor systems is also a nightmare if resource awareness is a must.

1.6. Resource-aware Programming

Robotic computer vision needs high performance computing power in low cost embedded systems. Future MPSoCs may offer 100s of CPUs and on-chip GPUs, but in order to exploit this computing power, central management concepts might meet their limits. In order to address the various challenges posed by the heterogeneous MPSoCs, related to resource-utilization, application mapping and to ensure certain quality of the obtained results, a resource-aware programming model has been proposed in [47]. In contract to the conventional programming model, on a resource-aware program can specify how many resources are essential for the application to meet its quality/throughout requirements, in addition to the degree of parallelism. Based on the above inputs, the runtime-system can allocate as many exclusive resources, in order to guarantee a certain quality of execution; for example in terms of processing speed. Such a programming model is currently out of reach in any type of available multi-core systems. Yet, *predictability* of execution qualities will be of utmost importance for all kinds of embedded, mixed-critical, and non-best-effort types of applications which shall benefit from the abundance of processors in future MPSoC designs.

Following the idea of resource-aware programming, the main goal of this work is to enhance the existing algorithms and develop resource-aware computer vision algorithms for the ARMAR robot, especially the algorithms with high computational requirements like computer vision. The goal is to explore techniques of self-organization to efficiently allocate available resources for the timely varying requirements of robotic applications. The programming model named *Invasive Computing* which is based on resource-aware programming model was explored. The benefits and restrictions of invasive architectures in challenging real-time embedded systems and in particular in humanoid robotics. Invasive Computing will provide the required self-organizing behavior to spread computations, variables and communications by the algorithms themselves. For example, planning collision-free motions for object grasping results in a varying demand on computing power, depending on the requested task. Since the search for collision-free trajectories in cluttered environments is known to be a PSPACE-hard problem, probabilistic approaches are commonly used for planning problems covering many degrees of freedom. The variety of tasks and setups, from footstep to reach and grasp planning, results in different sets of joints which are used for planning. For example, a grasping motion may cover the hip, the arm and the finger joints of a humanoid robot. An adaptive motion planning scheme dynamically changes the considered subsystems during the planning process (see [29]). Therefore, the complexity of the planning problem adaptively varies from three to up to twenty degrees of freedom. Due to the inherent flexibility and adaptivity as well as reconfigurability of Invasive Computing, the multi-level planning concepts can be implemented in a way that, depending on the current planning stage, a dynamic allocation of computing resources can be performed.

Using dynamically changing requirements of typical robot scenarios, as described above; this work demonstrate how such enhancements and adaptions can help the vision algorithms to achieve better quality results and higher throughput. Our experiments on a heterogeneous MPSoC named InvasIC, show that such a resource-aware methodology is often essential to satisfy the computing power requirements of robots and this also helps to reduce the interference between the various applications, so that algorithms in each step can produce consistent results regardless of the behavior of the other parts of the system.

2. Related Work

This chapter describes some of the well known multi-core/MPSoC systems available today. Section 2.1 describes the state-of-the-art MPSoC hardware while Section 2.2 describes various programming models developed to improve the resource utilization on the future MPSoCs.

2.1. Various MPSoC Platforms

With ongoing technology progress and as a consequence of the power wall, processors do not scale any more towards higher frequencies. Instead, the major trend goes towards the integration of more and more processor cores per chip. Miniaturization in the nano era makes it possible already now to implement billions of transistors, and hence, massively parallel computers on a single chip with typically 100s or 1000s of processing elements. Different many-core architectures have already been developed, including Tilera [48], Intel's TeraFlop [49] or Single-chip Cloud Computer (SCC) [50]. Some of these architectures are homogeneous comprising of uniform standard components, while the others are built using special hardware accelerators forming heterogeneous platforms.

Tilera TILE64 The Tile Processor Architecture consists of a 2D grid of identical compute elements, called tiles. Each tile is a powerful, full-featured computing system that can independently run an entire operating system, such as Linux. Likewise, multiple tiles can be combined to run a multiprocessor operating system such as SMP Linux. Figure 2.1 is a block diagram of the 64-tile TILE64 processor. The perimeters of the mesh networks in a Tile Processor is connect to I/O and memory controllers, which in turn connect to the respective off-chip I/O devices and DRAMs through the chip's pins. Each tile combines a processor and its associated cache hierarchy with a switch, which implements the Tile Processor's various interconnection networks. Specifically, each tile implements a three-way very long instruction word (VLIW) processor architecture with an independent program counter; a two-level cache hierarchy and a 2D direct memory access (DMA) subsystem.

Intel SCC is another such example of a homogeneous 48 cores architecture [50]. In this architecture 24 tiles comprising of pentium dual core processors are connected in a mesh network over a standard NoC. It supports formation of special voltage and frequency operation domains, comes with configurable memory access range, and inter-tile communication is performed using dedicated hardware message buffers. Applications from the areas of computer vision, high-performance computing (HPC) domain, etc., can make use of such homogeneous architectures and exploit their parallelism.

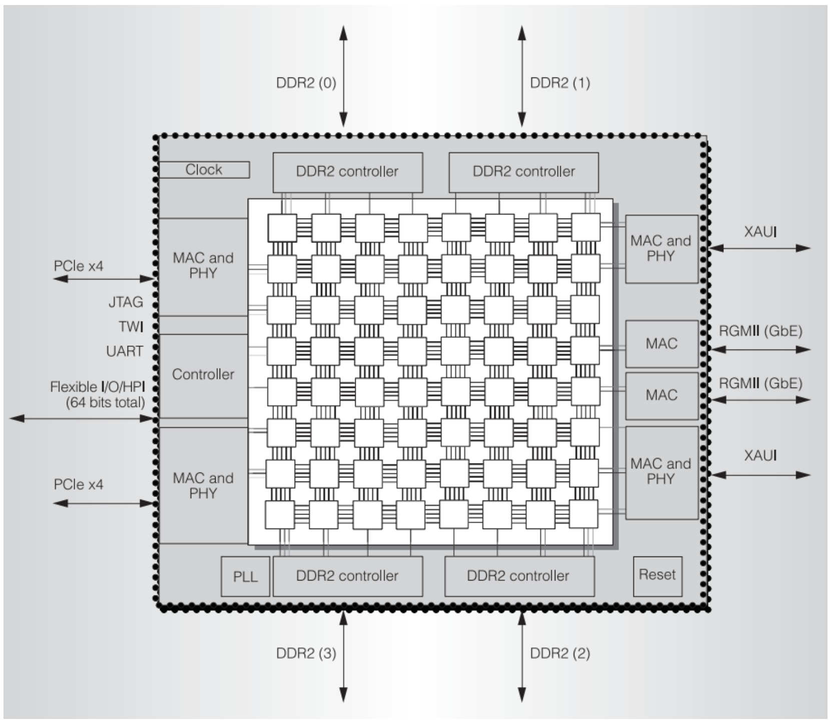


Figure 2.1.: Block diagram of the TILE64 processor [3]

IBM Cell Broadband Engine incorporates synergistic processing elements (SPE) and a master power processors (PPE) connected over a bus [51]. The first-generation Cell processor combines a dual-threaded, dual-issue, 64-bit PPE with eight newly architected SPEs, an on-chip memory controller, and a controller for a configurable I/O interface. These units are interconnected with a coherent on-chip element interconnect bus (EIB). The Cell processor uses a SIMD organization in the vector unit on the PPE and in the SPEs, as SIMD units have been demonstrated to be effective in accelerating multimedia applications.

AMD Accelerated Processing Units (APU) combines the general-purpose x86 cores of a CPU with programmable vector-processing engines of a GPU onto a single silicon die. Evaluation show that the fused CPU+GPU cores enable better performance than a discrete GPU and even traditional multi-core CPU processors by reducing the parallel overhead of PCIe data transfers. In this case, the compute unit is known as a SIMD engine and contains several thread processors, each containing four stream cores, along with a special-purpose core and a branch execution unit. The special-purpose core is designed to execute certain mathematical functions in hardware, for example, math functions like sin(), cos() and tan(). Since there is only one branch execution unit for every five processing cores, any branch in the program incurs some serialization to determine the path each thread should take. The execution of divergent branches is performed in lock-step manner for all the cores present in a compute unit. In addition, the processing cores are vector processors, which means that using vector types can produce material speedup on AMD GPUs.

AMD's Fusion APU, code-named Llano [5], is a heterogeneous multicore processor consisting of x86-64 CPU cores and a Radeon GPU, with an architecture as shown in Figure 2.3. The CPU and GPU cores have different virtual address spaces, but they can communicate

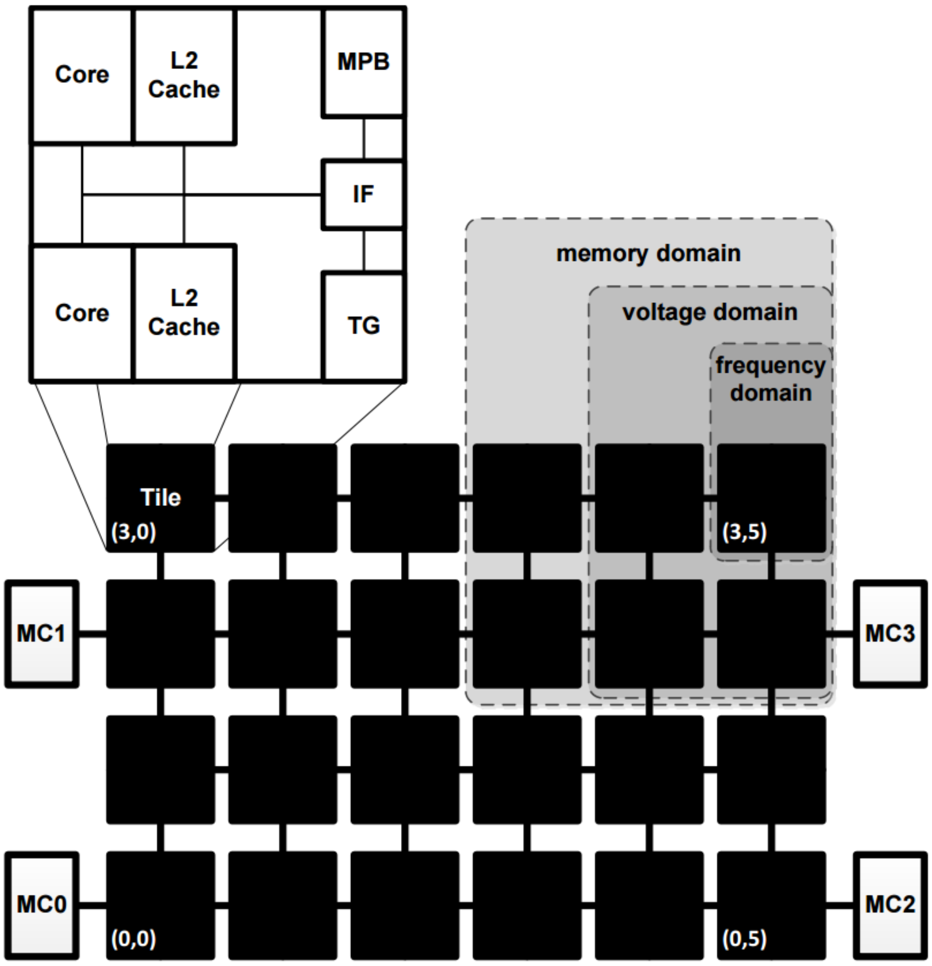


Figure 2.2.: Simplified hardware architecture of the SCC, displaying the arrangement of the cores and the contents of a tile [4]

via regions of physical memory that are pinned in known locations. The chip has a unified Northbridge that, under some circumstances, facilitates coherent communication across these virtual address spaces. Llano supports multiple communication paradigms. First, Llano permits the CPU cores and the GPU to perform coherent DMA between their virtual address spaces. In a typical OpenCL program, the CPU cores use DMA to transfer the input data to the GPU, and the GPU's driver uses DMA to transfer the output data back to the CPU cores. Although the DMA transfers are coherent with respect to the CPU's caches, the load/store accesses by CPU cores and GPU cores between the DMA accesses are not coherent. Second, Llano allows a CPU core to perform high-bandwidth, uncacheable writes directly into part of the GPU's virtual address space that is pinned in physical memory at boot. Third, Llano introduces the Fusion Control Link (FCL) that provides coherent communication over the Unified NorthBridge (UNB) but at a lower bandwidth. With FCL, the GPU driver can create a shared memory region in pinned physical memory that is mapped in both the CPU and GPU virtual address spaces. Assuming the GPU does not cache this memory space, then writes by the CPU cores and the GPU cores over the FCL are visible to

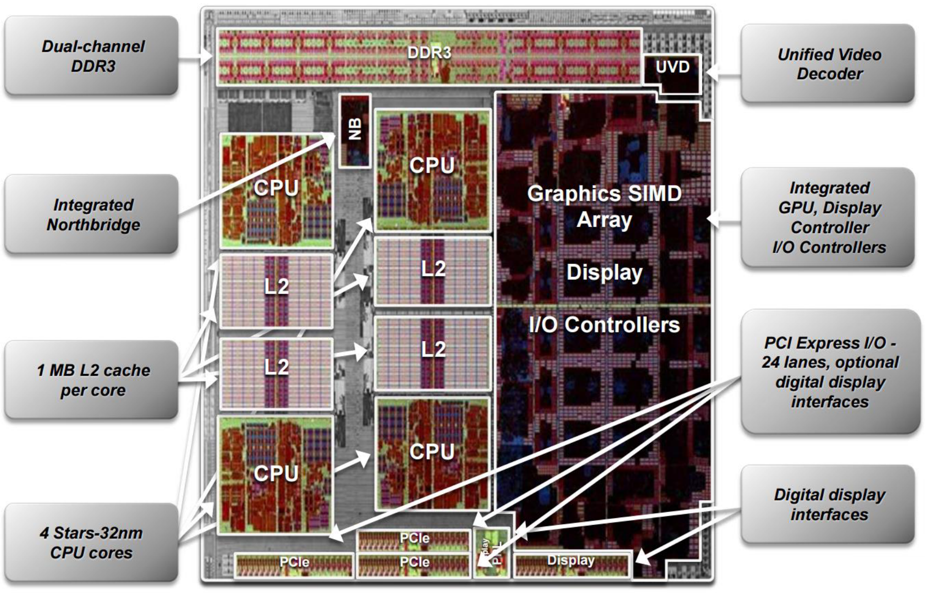


Figure 2.3.: AMD's fusion APU (Llano) [5]

each other, including GPU writes being visible to the CPU's caches. A GPU read over the FCL obtains coherent data that can reside in a CPU core's cache.

Ambric offers a highly unconventional architecture with over 300 processors, taking advantage of the large area now available to chip designers. The design consists of relatively simple processors, instantiating a large number of them in a 2D mesh. These processors are also very different to conventional CPUs, with only a few kilo-bytes of RAM per processor for instructions and data, and only basic integer addition and multiplication. The two types of processors used in the AM2045 [40] are the SR and SRD, both processors are very simple in-order 32-bit RISC processors, with the SR designed for extremely simple operations such as generating address streams and routing data around the device, while the SRD is more complex, with a large register set and an integer multiply-accumulate unit. Both processors execute most instructions with a throughput and latency of 1 cycle, with no stalls due to standard register usage. Ambric CPUs communicate with each other over channels, which are self-synchronizing uni-directional FIFOs, allowing producers and consumers to operate at different clock-rates in a GALS (globally asynchronous, locally synchronous) fashion. An implementation of the *optical flow* algorithm on Ambric [52] shows that massively parallel processor arrays (MPPA) can achieve real-time performance for computationally intensive algorithms with a physical and power footprint appropriate for embedding in autonomous systems such as unmanned vehicles, robots, etc., and that it competes well against a Virtex-4 FPGA implementation in both power consumption and throughput.

Xetal-II is a single-instruction multiple-data (SIMD) processor which delivers a peak performance of 107 GOPS on 16-bit data while dissipating 600 milliwatts (mw) of power [6]. As shown in Figure 2.4, the processing array (LPA) consisting of 320 PEs and a 10 Mbit on-chip frame memory handles the compute-intensive data processing. The data input processor (DIP) and data output processor (DOP) provide interfaces to three independent video channels, each with 10-bit resolution and carrying up to 80 Mpixels/s. A global control processor

(GCP) manages the operation of the IC. The chip is programmable in a subset of C extended with a vector data type. The adopted MP-SIMD processing paradigm provides high computational efficiency (MOPS/W) because of the good match with the parallelism inherently present in the algorithms and the data. Most image and video processing algorithms consist of many small kernels (such as convolution and edge detection) that operate on all data elements alike. This makes MP-SIMD processing a natural choice with computational efficiency originating from the reduced overhead of control operations. Since the architecture allows exploiting data-level parallelism present in most video processing applications and keeps the control and address decoding overhead low, a power-efficient operation is possible. In addition to facilitating inter and intra-frame processing for video scene analysis applications, the large on-chip frame memory enables low-power operation by avoiding high-speed off-chip memory access. This work also shows that utilizing the advantages of modern silicon technology, advanced video processing tasks, such as real-time full-frame video analysis, become feasible for portable consumer applications [6].

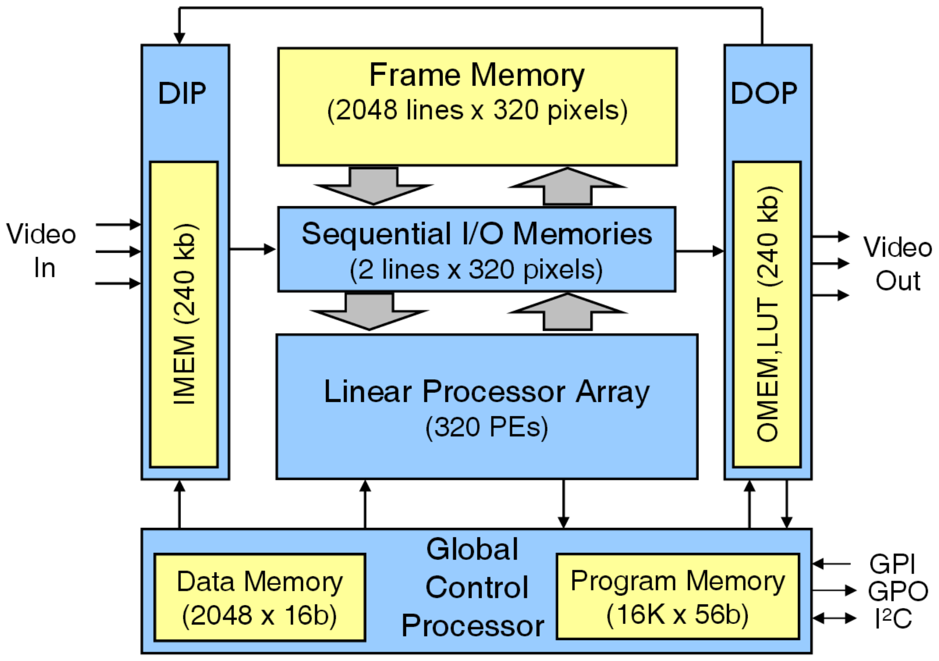


Figure 2.4.: Block diagram of the Xetal-II architecture [6]

Similarly, the **Xeon** is a brand of x86 microprocessors designed and manufactured by Intel Corporation, targeted at the non-consumer workstation, server, and embedded system markets. Primary advantages of the Xeon CPUs are their multi-socket capabilities, higher core counts, and support for ECC memory. Intel Xeon Phi features up to 61 cores each supporting 4 hardware threads with 512-bit wide SIMD registers achieving a peak theoretical performance of 1T op/s in double precision. In [53], the authors investigated the performance of the Xeon Phi coprocessor for sparse matrix multiplication kernels with very promising results.

Low power MPSoCs are highly suitable for mobile platforms. This includes the **Kalray** [54], an MPPA which integrates 256 user cores and 32 system cores on a chip with 28nm CMOS technology. Each core implements a 32-bit 5-issue VLIW architecture with the cores distributed across 16 compute clusters. The communication and synchronization between the clusters is ensured by data and control Networks-On-Chip (NoC). Each cluster contains 16 processing engine (PE) cores, one resource management (RM) core, a shared memory, a direct memory access (DMA) engine responsible for transferring data between the shared memories or within the shared memory. The compute cluster shared memory architecture optimizes a trade-off between area, power, bandwidth, and latency. The work presented in [54] demonstrates that this architecture is effective on two different classes of applications: embedded computing, with the implementation of a professional H.264 video encoder that runs in real-time at low power and in high-performance computing, with the acceleration of a financial option pricing application.

Tegra [55] is a system on a chip series developed by Nvidia for mobile devices such as smartphones, personal digital assistants, and mobile Internet devices. The Tegra integrates an ARM architecture central processing unit (CPU), graphics processing unit (GPU), and memory controller onto one package. Early Tegra SoCs are designed as efficient multimedia processors, while more recent models emphasize gaming performance without sacrificing power efficiency.

The **Snapdragon** [56] is a suite of system-on-chip semiconductor products designed and marketed by Qualcomm for mobile devices. The Snapdragon's CPU uses the ARM RISC instruction set, and a single SoC may include multiple CPU cores, a graphics processing unit (GPU), a wireless modem, camera, gesture recognition, etc. Snapdragon semiconductors are widely used in embedded systems (Google Android and Windows Phone devices). They are also used in cars, wearable devices, etc.

Similarly special DSP engines are incorporated in **TI's DaVinci** [57] architecture which comes along with an ARM core and peripherals. Such architectures are specially suited for multimedia and embedded applications. Designers can use the range of flexible IP blocks to tailor a SoC to meet the target video-equipment requirements.

2.2. Programming Models for MPSoCs

A major challenge associated with the MPSoCs and MPPAs described in Section 2.1 is the question on how to program such systems to make best use of their computing power. The work published in [58] compared different platforms for parallel computing like multi-core CPUs, GPUs, MPPAs, FPGAs, etc., and the results indicate that each platform require a different approach even for simple algorithms like random number generation. The case study also showed that the methods which are highly efficient in scalar oriented CPUs do not work well in wide issue GPUs, nor in memory limited MPPAs. Moreover, the emergence of many-core architectures necessitates a redesign of operating systems, including the interfaces they expose to an application. The new programming models and operating systems should be able to provide better support for parallel applications while the kernel itself should be scalable to hundreds of cores.

ROS - Resource-aware Operating System [59], has been developed for many-core hardware with support for parallel applications and a scalable kernel. ROS offers a resource-management scheme based on resource provisioning which enables system-wide, efficient accounting and utilization of resources. Resources, such as cores and memory, are explicitly granted and revoked. The kernel exposes information about a process's current resource allocation and the system's utilization, and allows the process to make requests based on this information. A process will not lose a core (or other granted resource) without being informed. Exposing information about a process's current resource allocation and the system's utilization, allows the application programs to make resource requests based on this information. These resources can be partitioned in both space and time, allowing processes to declare their resource needs in both dimensions. As an example, a process might indicate that it needs exclusive access to 25 cores 50% of the time, or that it requires 75% of the on-chip memory bandwidth 25% of the time. In this model, resource partitions serve to provision a set of resources to a group of processes rather than allocating them to a particular process. A process can always request more resources than have been provisioned for them, but there is no guarantee that they will be able to hold onto those resources if the system becomes over-provisioned. Treating resource partitions in this way leads to better utilization of system resources and reduces the hard problem of deciding when to revoke a resource from a process to the simpler problem of deciding which processes are allowed to create a resource partition in the first place. However, scheduling policy decisions such as which process to revoke a resource from, or how to discourage applications from hoarding resources is not explained in [59].

Tessellation [60] is another resource-aware many-core operating system that provides guaranteed fractions of system resources (such as processors, cache, network or memory bandwidth) to application programs. Tessellation uses an adaptive resource-allocation approach to provide quality of service (QoS) guarantees to applications while maximizing efficiency in the system. The resource allocation is handled by the Resource Allocation Broker (RAB), a service that distributes resources to cells while attempting to satisfy competing system-wide goals, such as deadlines met, energy efficiency, and throughput. The status of an application is monitored through performance counters and heartbeats. When an application is started, it provides its QoS requirements to the RAB Service in the form of target performance goals, such as desired frame-rates. The RAB Service continuously monitors the applications performance and compares it to target rates, adjusting resource allocations as required. To do this, the RAB Service utilizes two sources of information; the periodic performance reports (heartbeats) containing application-specific performance metrics and the system-wide performance counter values, such as cache-miss statistics and energy measurements. [61] shows how heartbeats can be used within an application to help a real-time H.264 video encoder maintain an acceptable frame rate by adjusting its encoding quality to increase performance. A heartbeat is registered after each frame is encoded and when the application checks its heart rate, it looks to see if the average over the last 40 frames was less than 30 beats per second (corresponding to 30 frames per second). If the heart rate is less than the target, the application adjusts its encoding algorithms to get more performance while possibly sacrificing the quality of the encoded image.

Autonomic Operating System project AcOS [62] enhances commodity operating systems with an autonomic layer that enables self-X properties through adaptive resource allocation. They aim at enabling users to state Service-Level Objectives (SLOs) and to automatically tune resource allocations in order to meet these user-specified SLOs, while enforcing system-level constraints, such as maximum processor temperature.

A further related approach is **SEEC** [63]. The goal of SEEC is to reduce the programming effort in multi-core systems. The idea is to (1) have application programmers in charge of specifying goals and progress and (2) enjoin on system programmers the specification of system-level actions that can affect application-level processes. Based on these specifications, the SEEC runtime system then attempts to optimize resource allocations using an *adaptive control system* that learns application and system models on-line. Regarding the constructive approach, SEEC is implemented as a runtime system and a set of libraries for Linux. In this sense, it shares the same approach with AcOS, as it cannot directly control the hardware to provide for the strict guarantees that are needed to separate different applications from each other.

Overall, we are moving from multi-core architectures towards multi-processor system on chip (MPSoC) designs. These devices will be heterogeneous in terms of on-chip processors, communication facilities and memory organization. In the future, shared and distributed memory will coexist on a single chip. At a certain level, cache coherence is no longer implemented in hardware analog to Intel's single-chip cloud computer (SCC). Future MPSoCs are expected to have numerous cores available on a single chip, so that every single thread will be able to run on his own private core. This calls for a radical change in the way operating systems manage processors; it also calls for programming and system paradigms suited for very fine-grained parallelism.

3. Invasive Computing

Decreasing transistor size have already led to a rethinking on how to make efficient use of multimillion transistor on system-on-a-chip designs. With ongoing technology progress and as a consequence of the power wall, processors do not scale any more towards higher frequencies. Instead, the major trend goes to the integration of more and more processor cores per chip. Different many-core architectures like Tilera's TILE64 processor, Intel's TeraFlop or SCC have already emerged. From a CMOS technology perspective, the integration of several hundreds of cores will be feasible in the foreseeable future. The most important problem, however, related to massively parallel MPSoCs is how to efficiently exploit the abundantly available processing power, is still unsolved. This thought has opened a new way of thinking about parallel algorithm design.

One way of how to manage and control the parallel execution on such MPSoCs would obviously be to give the power to manage resources, to the programs themselves and thus, have the running programs manage and coordinate the processing resources themselves to a certain degree and in context of the state of the underlying compute hardware. Through such techniques, the new programming paradigm named Invasive Computing will provide scalability, higher resource utilization numbers and, also performance gains by adjusting the amount of allocated resources to the temporal needs of a running application.

The main aim of this work is to demonstrate how Invasive Computing can solve many of the challenges faced by today's massively parallel and heterogeneous MPSoCs, by incorporating principles of self-organization various computer vision algorithms used on the ARMAR robot. The expected benefits includes a gain in computational efficiency and performance, self-adaptive applications which can optimize itself for optimum resource utilization.

3.1. Basic Principles

The name Invasive Computing stems from the notion that applications can dynamically *invade* (exclusively acquire) resources according to their needs for computation power, communication bandwidth/latency or memory size. The main idea behind invasion is to add to each application the ability to explore and claim resources in a certain neighborhood and to copy its configuration code, program, and possibly data to such places in a phase of invasion, and then to execute the given problem in parallel based on the available (invasible) region of processing resources. In such a system, applications compete with each other for a share of the hardware resources. Through invasion, an application will thus be able to spread its computations for parallel execution based on the availability and the actual state of processing resources. For execution phases with reduced degree of available application parallelism, the application may itself perform a retreat to free occupied resources so to optimally exploit all resources and make them available for other applications.

The chart depicted in Figure 3.1 shows the typical state transitions that occur during the execution of an invasive program. This process starts with the construction of an initial claim which holds a set of resources that the application can use for its parallel execution. The *claim* construction is performed by issuing a call to *invade*. Resource allocation is followed by an *infect* call which is used to start the actual application code on the previously allocated resources (within the *claim*). The actual application code that is spread onto infected resources for subsequent parallel execution is called *i*-let. Once the execution on all PEs finishes, the number of PEs inside the claim can be altered by calling *invade* or *retreat* to either expand or shrink the application's *claim*. In case of *retreat*, the processing elements are cleaned up from the *i*-let entities that have been setup by *infect*. Alternatively, if the degree of parallelism does not change, it is also feasible to dispatch a different program onto the same set of PEs by issuing another call to *infect*. If a call to *retreat* leaves the claim empty, there are no computing resources left for further execution of the program, hence it terminates its execution and exits.

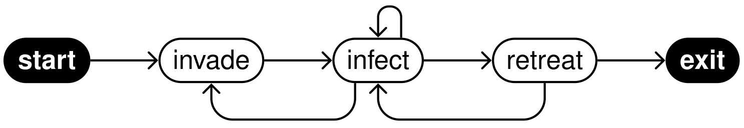


Figure 3.1.: Structure of an invasive program

In contrast to mere *allocation* of a fixed amount of resources, *invasion* can lead to over- or under-fulfillment of the application's requirements, depending on the current state of the system. As a consequence, applications are required to be adaptable, because the result of an *invasion* may not fully meet their original demand. However, once an application acquires a set of resources, it gains exclusive access to them. This allows the application to tune its algorithms and distribute its workload depending on the amount of resources actually received from the system. The resource guarantees in turn ensure that decisions taken during program tuning and workload distribution remain valid until the application releases the resources back to the system. The modified flow diagram with an additional stage to adapt the workload, is shown in Figure 3.2.

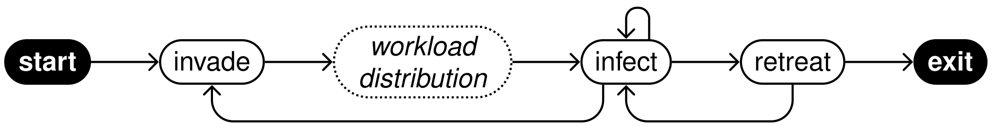


Figure 3.2.: Structure of an invasive program

3.2. InvasIC - Hardware Architecture

An incarnation of a hardware architecture supporting Invasive Computing is referred to as InvasIC. Heterogeneity is an important attribute of the InvasIC architecture, to exploit both loop level and thread level parallelism in applications. The heterogeneous tile-based MPSoC architecture consists of different types of processing elements, memory and input/output devices. The processing elements include loosely coupled RISC processors, Tightly-Coupled Processor Arrays (TCPAs) adapted to data driven applications and dynamically reconfigurable, application specific instruction set extensions(*i*-Cores). All tiles are connected by the invasive network on chip (*i*-NoC) in a 2D mesh topology. The invasive network adapter (i-NA) is the interface between the tile local bus and the *i*-NoC routers. All aspects of the architecture are designed with focus on scalability and decentralized resource management. The components within a RISC compute tile communicate using standard buses whereas the tile-external communication is performed via an *i*-NoC. Additionally on-chip memory tiles, off-chip memory tiles (accessible via memory controllers) and I/O tiles for interfacing with the peripherals are also provided.

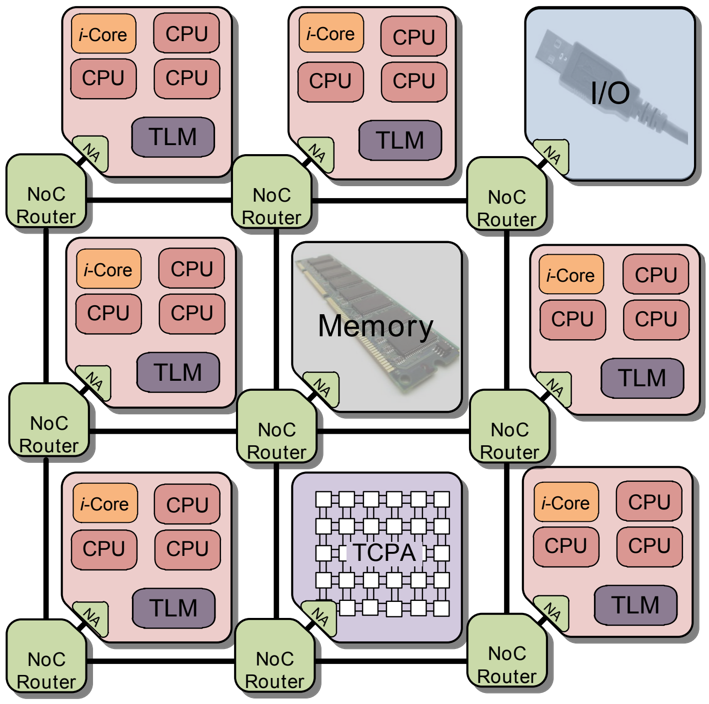


Figure 3.3.: InvasIC - a heterogeneous MPSoC with tiled architecture

3.2.1. InvasIC - Memory Hierarchy

The InvasIC offers a distributed shared memory architecture. Processes running on cores within one compute tile communicate using shared memory regions such as the Tile Local Memory (TLM) or external memory (DDR-II). These different memory sub-components are physically distributed and are accessed over bus and *i*-NoC, hence represent a non-uniform memory access (NUMA) architecture. Details of the InvasIC cache and memory hierarchy are provided below:

- Each PE has its own L1 cache and a shared L2 cache, within each tile.
- L1 instruction and data caches with cache coherence among cores within the same tile.
- No cache coherence between the tiles.
- Scratchpad memory (optional) is private to the core, provisions fast access and is not cached.
- The Tile Local Memory (TLM) is accessible by all cores within a tile over the bus.
- The TLM of other tiles can be addressed from each tile directly.
- Shared on-chip memory constitutes one or more memory tiles accessible over the *i*-NoC.
- External memory is accessible via memory controllers on I/O tiles, via the *i*-NoC.

Various components constituting the heterogeneous MPSoC architecture are described below.

3.2.2. RISC Processors (LEON3)

The LEON3 is a synthesisable VHDL model of a 32-bit processor compliant with the SPARC V8 architecture and are available as part of GRLIB [7]. LEON3 cores are used in the InvasIC architecture as they are light weight and suitable for general purpose computations. The LEON3 processor has the following features:

- SPARC V8 instruction set with V8e extensions
- Advanced 7-stage pipeline
- Hardware multiply, divide and MAC units
- High-performance, fully pipelined IEEE-754 FPU
- Separate instruction and data cache (Harvard architecture) with snooping
- Local instruction and data scratchpad RAM, 1 - 512 Kbytes
- Advanced on-chip debug support with instruction and data trace buffer
- Symmetric Multi-processor support (SMP)
- Power-down mode and clock gating

A block diagram of the LEON3 processor is shown in Figure 3.4.

LEON3 cores were selected for use within the InvasIC architecture due to their low complexity and open source availability as part of the GRLIB. The full source code is available under the GNU GPL license, allowing free and unlimited use for research and education. Moreover, it is highly configurable and particularly suitable for system-on-a-chip (SoC) designs. Conceptually speaking, other cores, such as ARM [64] or Intel cores [65] could also be employed. Another benefit from using the GRLIB is the availability of an AMBA bus system and numerous components, such as memory controllers or I/O interfaces, as well as debug support.

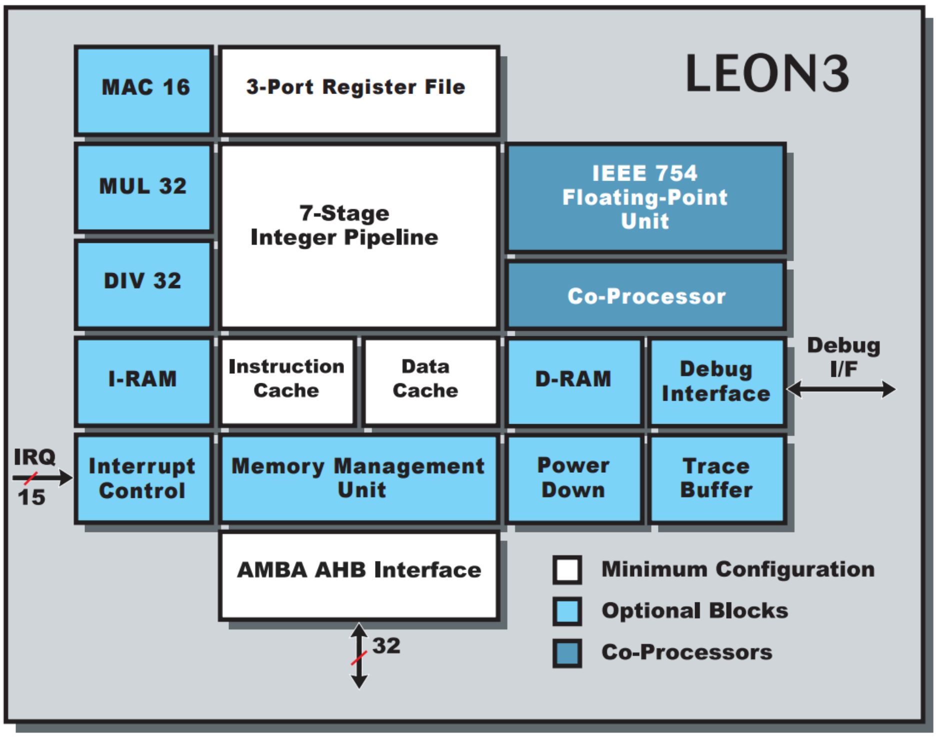


Figure 3.4.: LEON3 Processor [7]

3.2.3. Invasive Cores (*i*-Core)

Embedded processors are the key in rapidly growing application fields ranging from image processing, automotive, entertainment, to name just a few. In the early 1990s, the term ASIP (Application Specific Instruction-set Processors) has emerged denoting processors with an application-specific instruction set. They are more efficient in one or more design criteria like 'performance per unit area' and 'performance per watt' compared to mainstream processors.

Following this methodology, the *i*-Core (Invasive-Core) is a reconfigurable RISC core based on the LEON3 SPARC V8 architecture. It combines concepts for an adaptive micro-architecture and fine-grained reconfiguration to instantiate application-specific accelerators [66]. A reconfigurable processor consists of a regular processor core coupled with a reconfigurable fabric, allowing application-specific accelerators to be loaded at runtime. The processor core can be implemented as an ASIC and the fine-grained reconfigurable fabric as an embedded FPGA. This brings in a high degree of freedom in accelerator design and enables applications from different domains to be run at a high performance, on the same reconfigurable processor. To speed up execution of computationally intensive kernels, applications can use accelerators on the FPGA fabric via Special Instructions (SIs), which are extensions of the instruction set architecture (ISA) of the processor core.

In order to build an adaptive micro-architecture, the LEON3 design was extended to support reconfiguration of the processor pipeline, run-time configuration of cache-parameters and the branch prediction unit, as shown in Figure 3.5. This enables to adapt the micro-architecture to the requirements of the applications using it. The fine-grained reconfigurable

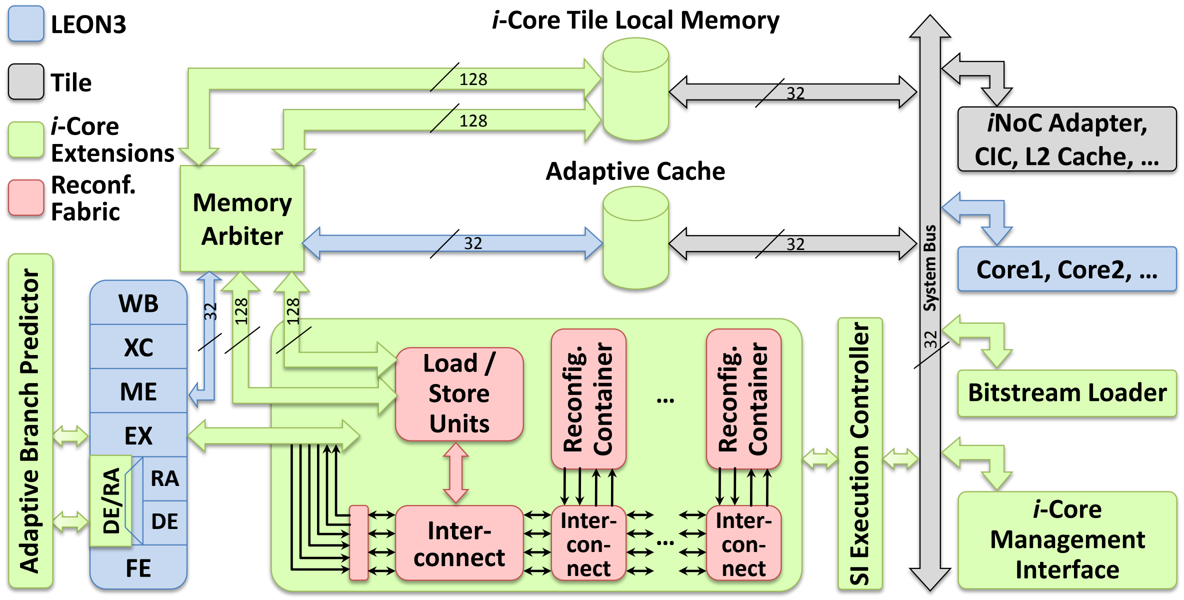


Figure 3.5.: i-Core consisting of adaptive Microarchitecture and reconfigurable fabric [8]

accelerator of the *i*-Core relies on an FPGA fabric that is loosely connected to the processor pipeline. This allows low-latency access to the fabric and therefore the ability to attain speedup even for applications consisting of kernels with short durations (e.g. in the range of 10 to 100 cycles when executed on the fabric) [67].

The reconfigurable fabric has a dedicated high-bandwidth connection to a fast on-chip memory. In order to use the fabric for an accelerator, it is configured using a partial configuration bitstream from a library. This reconfiguration is triggered by an additional instruction implemented in the adapted LEON3 design. The accelerator is used after configuration by executing special instructions on the LEON3 core. This approach exceeds the concept of state-of-the-art ASIPs, as it adds flexibility and additionally enables dynamic run-time adaptation towards the executing application to increase the performance.

However, in contrast to the adaptive micro-architecture, the reconfigurable fabric can only be used if this is supported by the binary of the application. The speedup that can be attained by executing an SI on the fabric instead of executing the equivalent ISA instructions on the pipeline, depends on the inherent parallelism of the respective kernel and the amount of fabric used for its implementation. The *i*-Core uses a dedicated connection to its tile local memory, using two 128-bit ports to provide a high memory bandwidth to expedite the execution of special instructions.

The implementation of a particular ISE may adapt during run-time. These adaptations correspond to ISA-dependent adaptations at the micro-architecture level. Depending on the executing *i*-lets and their specific ISE requirements, the reconfigurable fabric is invaded to realize a certain subset of the requested ISE. Note that it is typically not possible to fulfill all ISE requests, depending on the size of the available reconfigurable fabric and the requirements of the *i*-lets. The concepts of Invasive Computing are used to manage the competing requests of different tasks. In the scope of reconfigurable processors this means that each task specifies the SIs that it uses. Additionally, each task provides information on what performance improvement (speedup) can be expected, depending on the size of the reconfigurable fabric that is assigned to it. A run-time system then decides which task obtains what share of the reconfigurable fabric.

The reconfigurable fabric is shared among several tasks and thus, the ISE of a particular task has to cope with an at compile-time unknown size of the reconfigurable fabric. The approach of so-called modular SIs allows for providing different trade-offs between the amount of required hardware and the achieved performance by breaking SIs into elementary reconfigurable data paths (DPs) that are connected to implement an SI. A SI may be realized by multiple SI implementations, typically differing in their hardware requirements and their performance. Modular SIs are composed of DPs, where typically multiple DPs are combined to implement an SI. Depending on the provided amount of DPs, the SI can execute in a more or less parallel way. When more DPs are available, then a faster execution is possible (that corresponds to a different implementation of same SI). Dynamically changing between these different performance-levels of an SI implementation allows reacting on changing application requirements or changing availability of the reconfigurable fabric.

To request a share of the reconfigurable fabric, the task issues an *invade* call. This selects a share of a resource and grants it to the task. A part of reconfigurable fabric of the i-Core is assigned to the task. The size of the assigned fabric depends on the speedup that the application is expected to exhibit. Generally, the more fabric is available to the task, the higher the speedup, but the expected speedup for a given amount of fabric is task-specific. During execution of invade, the run-time system will use the speed-up information provided, to decide which share of the fabric will be granted to the task. In the worst case, the application will receive no fabric at all (due to heavy load from other tasks).

After the run-time system has decided which SI implementations to use, the application can start using the accelerators on the reconfigurable fabric by issuing the *infect* call. Reconfiguration of the FPGA fabric occurs in parallel to the task execution, allowing the application to continue processing without waiting for the fabric to finish loading. After completing execution of a particular kernel, an entirely different set of SIs may be executed on the same share of the fabric attained by the task during its initial *invade* call. The fabric must, however, be prepared for execution of the new SIs. The presence of an adaptive instruction set architecture and adaptive micro-architecture allows optimizing performance relevant characteristics of the *i*-Core to task-specific requirements. This makes the *i*-Core especially beneficial in multi-tasking scenarios, where different tasks compete for the available resources.

3.2.4. Tightly Coupled Processor Array (TCPA)

Computer vision applications require high computing power to deal with the increasing complexity of the algorithms and the high pixel rates [68]. Moreover, it is important to process the input data and analyze the results in real-time, due to an uncontrolled and continuously changing environments, for e.g. robotic maneuver. One way to achieve better performance is to implement them on Massively Parallel Processor Arrays (MPPAs) rather than on conventional general-purpose cores like LEON3. Low level pixel processing algorithms with massive inherent parallelism highly benefit from such massively parallel architectures and consequently the processing time can be significantly reduced [52].

It is expected that the number of cores on an MPPA will continue to double every 18 months. It can be foreseen that MPPAs in the year 2020 and beyond can incorporate 1000s of processing elements (PE) on a single chip [69]. Tightly coupled processors arrays (TCPA) are highly parametrized coarse-grained processor array templates, which can be used in multimedia and wireless applications that require real-time or near real-time processing speeds [70]. TCPAs consist of an array of tightly-coupled lightweight processor elements [71]. The main advantage of the TCPA architectures is the possibility of partial and differential reconfiguration. Instruction level parallelism is employed by a VLIW architecture as shown in Figure 3.6. The next level of parallelism consists of multiple processing elements (PEs) working together. Such architectures are well suited as domain-specific companions in an MPSoC for acceleration of nested loop programs from digital signal processing and multimedia applications. Different configurations can be loaded (both program and interconnect) in the configuration memory and a reconfiguration manager can configure the array at runtime. Different application can be executed on the same array.

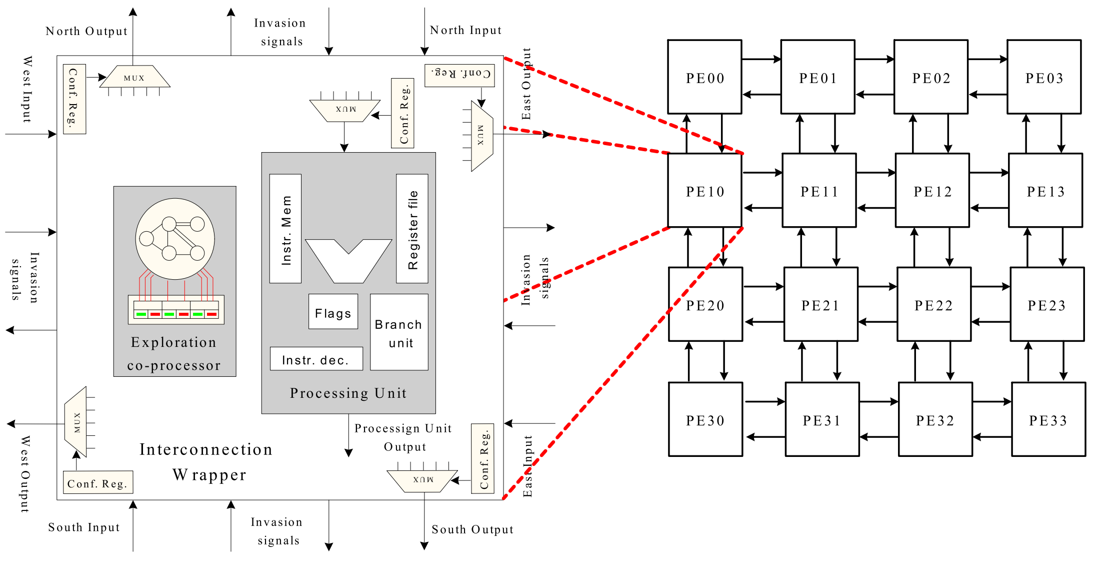


Figure 3.6.: Structure of a processing element within the TCPA

The TCPA may consist of heterogeneous PEs. For instance, some of the processors at the borders might include extra functionality for the purpose of address generation. The PEs in the array are interconnected by a circuit-switched mesh-like interconnect with a very low latency, which allows data produced in one PE to be used already in the next cycle by a neighboring PE.

An interconnect wrapper encapsulates each PE and is used to describe and parameterize the capabilities of switching in the network. The wrappers are arranged in a grid fashion and may be customized at compile time to have multiple input/output ports in the four directions, that is, north, east, south, and west. Using these wrappers, different topologies between the PEs like grid and other systolic topologies, but also (higher-dimensional) topologies such as torus or 4D hypercube can be implemented and changed dynamically. To define all possible interconnect topologies, an adjacency matrix is given for each interconnect wrapper in the array at compile time. Each matrix explains how the input ports of its corresponding wrapper and the output ports of the encapsulated PE are connected to the wrapper output ports and the PE input ports, respectively. If multiple source ports are allowed to drive a single destination port, then a multiplexer with an appropriate number of input signals is generated. The select signals for such generated multiplexers are stored in configuration registers and can therefore be changed dynamically. By changing the values of the configuration registers in an interconnect wrapper component, different interconnect topologies can be implemented and changed at runtime [71]. Two different networks, one for data and one for control signals, can be defined by their data width and number of dedicated channels in each direction. For instance, two 16-bit channels and one 1-bit channel might be chosen as data and control network respectively. Note that the data and control path width for the other architectural components such as functional units and registers is deduced from the selected channel bit widths.

As shown in Figure 3.6, a PE itself is again a highly parameterizable component with a VLIW (Very Long Instruction Word) structure. Here, different types and numbers of functional units (e.g., adders, multipliers, shifters, logical operations) can be instantiated as separate functional units which can work in parallel. The size of the instruction memory and register file is also parameterizable. The processing elements are weakly programmable since the functional units have only a reduced instruction set that is domain specific, tailored for one field of application. Additionally, the control path is kept very simple (no interrupt handling, multithreading, instruction caching, etc.), and only single-cycle instructions and integer arithmetic are considered. Consequently, no interrupt, exception, data or instruction cashes, and multi-threaded execution is supported by PEs. A highly parameterizable register file for each PE is provided to store data and control signals and also the results from various functional units. Furthermore, the register file contains input FIFO buffers in order to store incoming data and control signals to the PE. Using the run-time reconfigurable interconnect structure, different user-defined interconnection topologies can be implemented between PEs. The interconnection structure is made of point-to-point connections among PEs which allows them to pass data and control signals without communication delay. In order to optimize the configuration periods of an array, a set PEs sharing a same property for their program and interconnection configuration may be grouped into one domain.

In order to feed the data, array consists of in/out buffers/FIFOs at the four sides of the array, as shown in Figure 3.7. Address generators are used to fetch the correct data from the main memory and load it to the buffers. A Global controller takes care of the execution of the loop program by sending different control signals to the array.

Typically, applications are statically mapped in a spatially or temporally partitioned manner on such array processors. To overcome this rigidity, the TCPA support at hardware level the ideas of Invasive Computing such as resource exploration and management. For this purpose, new controllers were integrated in each PE of a TCPA to enable extremely fast and decentralized resource allocation [72]. Additionally, these invasion controllers enable hierarchical power management in TCPAs [73].

In a typical invasion scenario, an application which has started on an MPSoC system can use the TCPA to execute loop programs. During this process, the application can claim (*invade*) available PEs and execute (*infect*) the program and release (*retreat*) the PEs after execution.

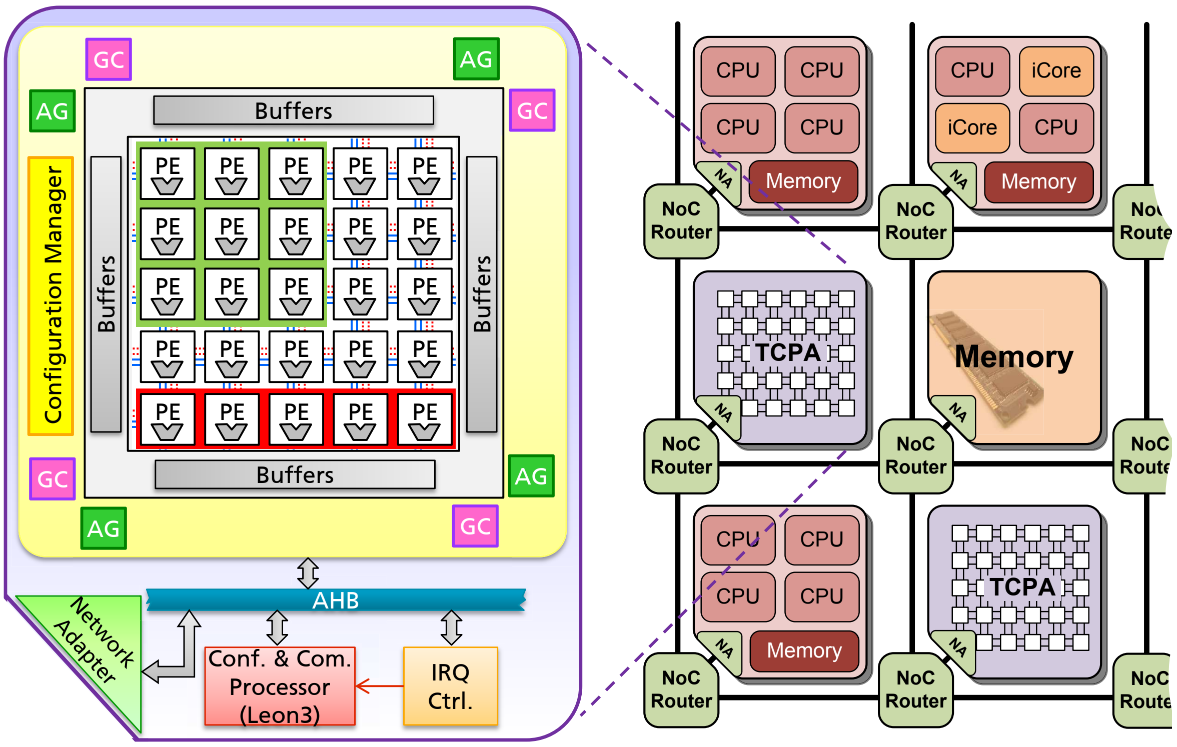


Figure 3.7.: Tightly Coupled Processor Array (TCPA) Architecture [9]

3.2.4.1. Application mapping on TCPA

TCPA can be shared by various applications as shown in Figure 3.8. In this example an image filter application shares the processor array with three other applications. The PEs free for invasion are colored in white. The image filter (green colored region) can choose different regions within the TCPA as shown in Figure 3.8, based on its computing requirements.

Any application running on the TCPA, begins its execution on a single PE as in a single-core processor. As it reaches a stage for parallel execution, the application starts looking for more resources in the neighborhood with the help of resource exploration controller, embedded within its PE (master PE). Based on the resource requirements a certain amount of resources will be acquired (invaded) by this PE. It then proceeds execution in parallel on the invaded PEs. Finally when the operation completes the master PE releases (retreat) the invaded resources allowing other applications to use those PEs, if required. The idea of Invasive Computing is demonstrated using Figure 3.9. Figure 3.9(a) represents an invasion carried out by an application running on a single PE, which is in need for 8 other PEs to spread its computations. This figure also represents a 2nd application which was already running on the TCPA.

Figure 3.9(b) shows the resources invaded by the first application and a retreat operation by the 2nd application, as no more parallelism is available within that application. Later the first application performs a retreat upon completing its parallel execution, as shown in Figure 3.9(c). Subsequently, this application once more requests for more resources as shown in Figure 3.9(d), but with greater resource requirements this time. Thus through Invasive

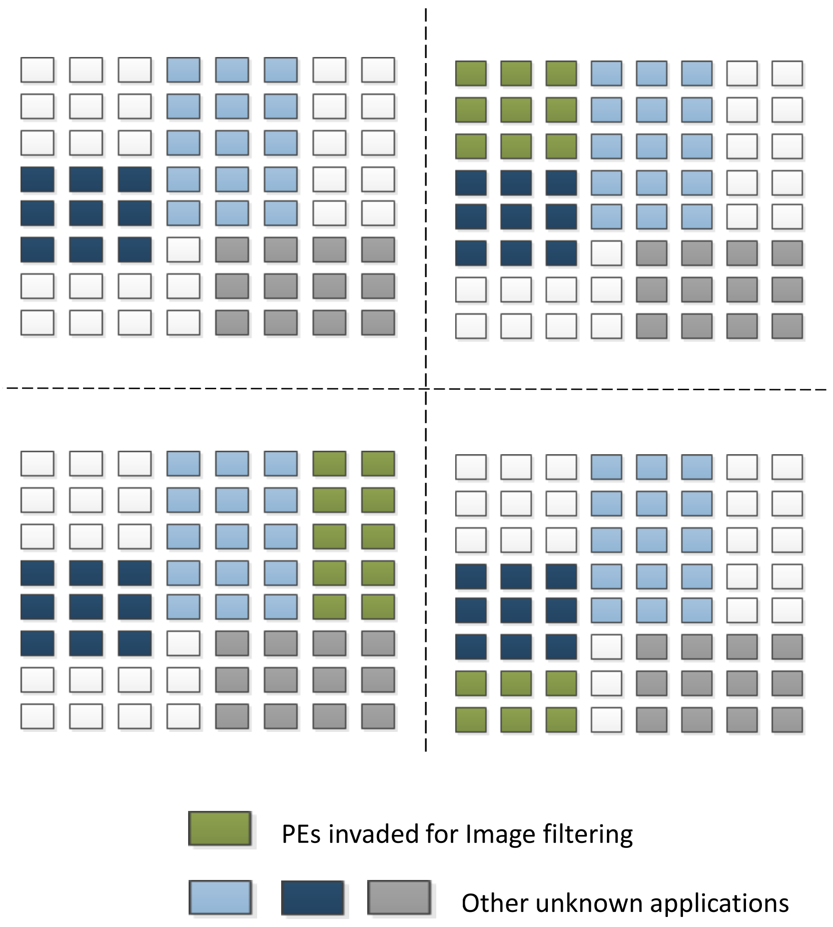
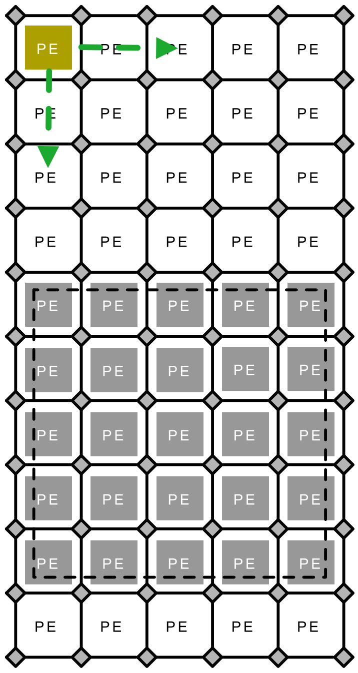


Figure 3.8.: Various possible invasions within a shared TCPA for image filtering application

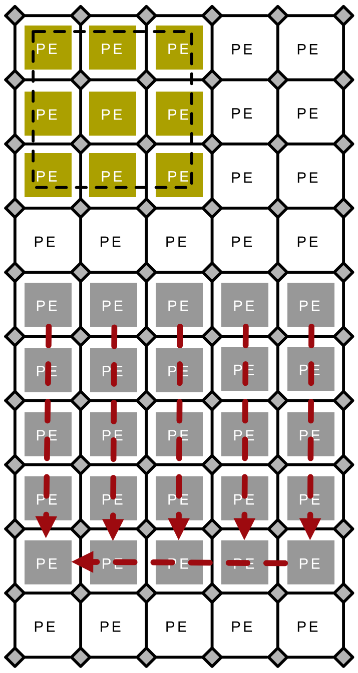
Computing the applications release resources immediately after parallel execution, enabling resource sharing across various applications running on the TCPA.

3.3. OctoPOS - A Resource-aware Operating System

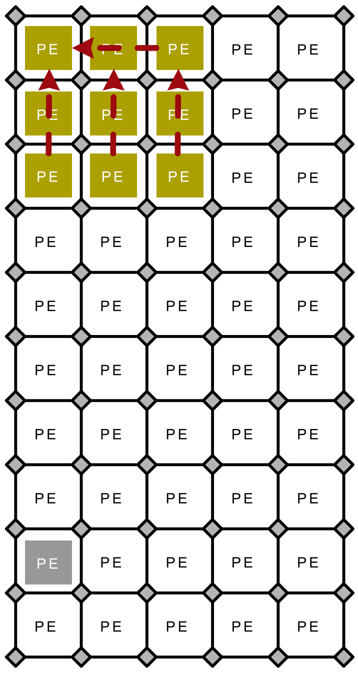
To support this idea of *self-adaptive* and *resource-aware programming*, not only new programming concepts, languages, compilers, and operating systems need to be developed, but also revolutionary architectural changes in the design of MPSoCs (multiprocessor systems-on-a-chip) to efficiently support invasion, infection, and retreat operations. OctoPOS [74] is an operating system (see Figure 3.10) specifically designed to support Invasive Computing, is presented in this section.



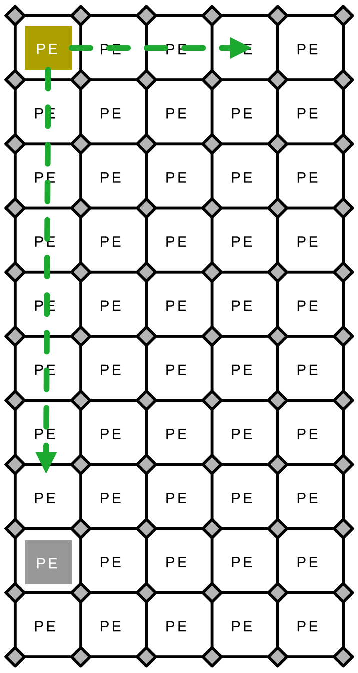
(a) Invasion stage-1



(b) Invasion stage-2



(c) Invasion stage-3



(d) Invasion stage-4

Figure 3.9.: Invade and retreat operations inside the TCPA

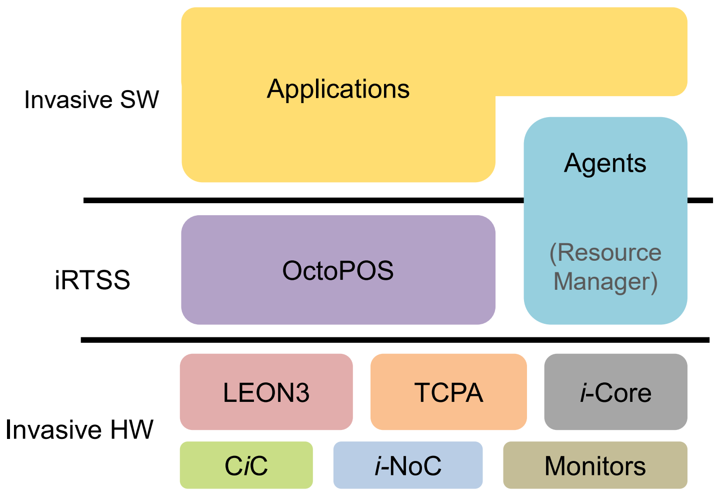
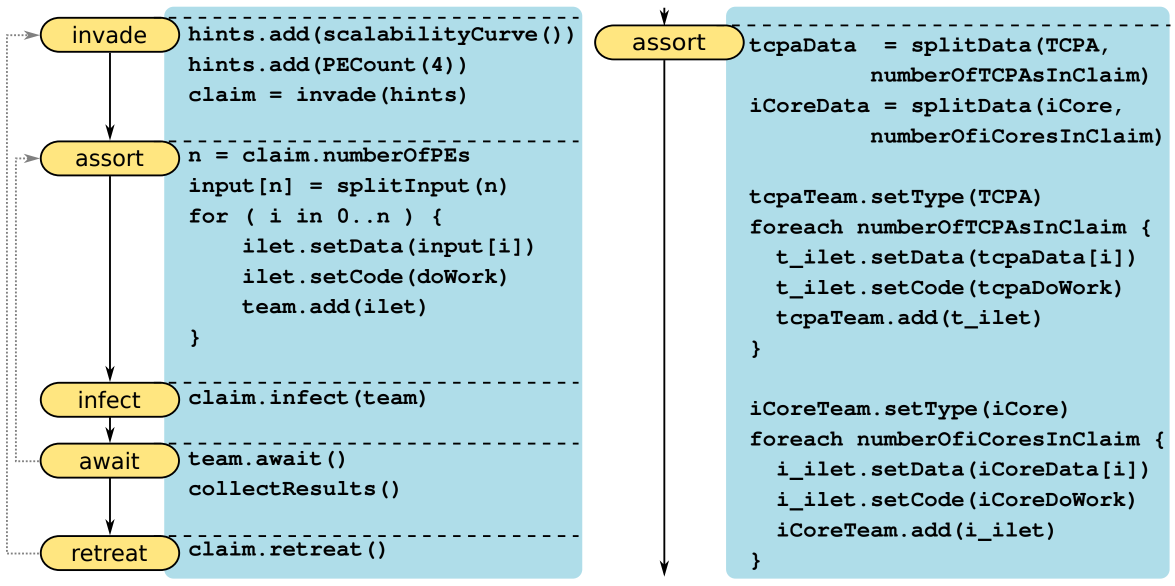


Figure 3.10.: Various layers forming Invasive Computing programming model

3.3.1. The Invasive Programming Model

In Invasive Computing, parallelism is expressed in the form of *i*-lets: light-weight units of execution which consist of a snippet of code that can be executed in parallel to other *i*-lets, and the data on which it operates. Compared to classical threads, *i*-lets are significantly more efficient to create and dispatch, and can implement parallelism in a far more fine-grained way.



```
hints.add(scalabilityCurve())
hints.add(PECount(4))
claim = invade(hints)
```

```
n = claim.numberOfPEs
input[n] = splitInput(n)
for ( i in 0..n ) {
    ilet.setData(input[i])
    ilet.setCode(doWork)
    team.add(ilet)
}
```

```
claim.infect(team)
```

```
team.await()
collectResults()
```

```
claim.retreat()
```

```
tcpaData  = splitData(TCPA,
                numberOfTCPAsInClaim)
iCoreData = splitData(iCore,
                numberOfiCoresInClaim)

tcpaTeam.setType(TCPA)
foreach numberOfTCPAsInClaim {
  t_ilet.setData(tcpaData[i])
  t_ilet.setCode(tcpaDoWork)
  tcpaTeam.add(t_ilet)
}

iCoreTeam.setType(iCore)
foreach numberOfiCoresInClaim {
  i_ilet.setData(iCoreData[i])
  i_ilet.setCode(iCoreDoWork)
  iCoreTeam.add(i_ilet)
}
```

Figure 3.11.: Application flow in an invasive program (left) and assort phase with a heterogeneous claim (right) [8]

Figure 3.11 shows an exemplary application flow in an invasive program. First, during the invade phase, the application expresses a request for a set of resources through *hints* given to the runtime system. In the example, it specifies that it would like to acquire four processing elements, and give an estimate as to how well it would scale if it were to receive fewer or more processing elements. The system then decides, according to the global system state, which resources to assign to the application, and returns a *claim* describing these resources.

In the *assort* phase that follows, the application adapts itself according to the contents of the claim by assorting a so-called *team* of *i*-lets. In a simple scenario, the application creates at least one *i*-let for each reserved processing element to provide maximal system utilization.

The resources of the claim can then be *infected* with the assorted team, leading to the execution of the team's *i*-lets. Once the execution has finished, the results can be collected and merged. The application may subsequently either reuse and adapt the claim for further computations or *retreat* from it, releasing the associated resources.

The application can request different types of resources, such as LEON3, *i*-Core or TCPA, in the *invade* phase. Depending on the structure of the application and its hints, a claim can consist of one or more resource types. With multiple resource types assigned to the application's claim, the *assort* phase becomes more complex. In order to deal with the claim's heterogeneity, the application assorts different teams for each resource type. It splits the input data depending on the proportion of each resource type in the claim and the expected performance gain.

3.3.2. OctoPOS - Design Principles

OctoPOS [74] provides the necessary OS-level primitives and a scalable, efficient, low-overhead execution environment for invasive-parallel applications. The primary *raison d'être* of OctoPOS is the insight that, in practice, the multi-threading model implemented by contemporary operating systems is far from light-weight: Performing a context switch already takes thousands of CPU cycles, and creating a new thread even wastes tens of thousands of cycles. In contrast, the goal of OctoPOS is to build a tailored operating system that cuts down on these costs.

OctoPOS is designed for maximum scalability. On the one hand, this means being scalable to systems with hundreds or even thousands of processing elements, on the other hand scalability also has to encompass the application itself: OctoPOS has to be able to execute applications that dynamically create lots of potentially parallel tasks on large many-core systems. We tackle these requirements by employing the following design principles.

On a coarse level, scalability is achieved by composing the overall system out of several independent OS instances. With respect to the invasive hardware architecture described in Section 3.2, each tile of the many-core system runs its own OctoPOS instance. This approach somewhat resembles the Multikernel presented in [75], but OctoPOS goes one step further and does not rely on global cache coherency between instances. This increases scalability especially for large many-core systems, where cache coherency is getting increasingly difficult to implement, becoming a bottleneck in the whole system. Instead, the operating-system instances communicate with each other using special operations provided by the network-on-chip that connects the different tiles.

Inside the OS instances, OctoPOS strives to improve scalability by relying solely on non-blocking synchronization for implementing the key components of the application runtime

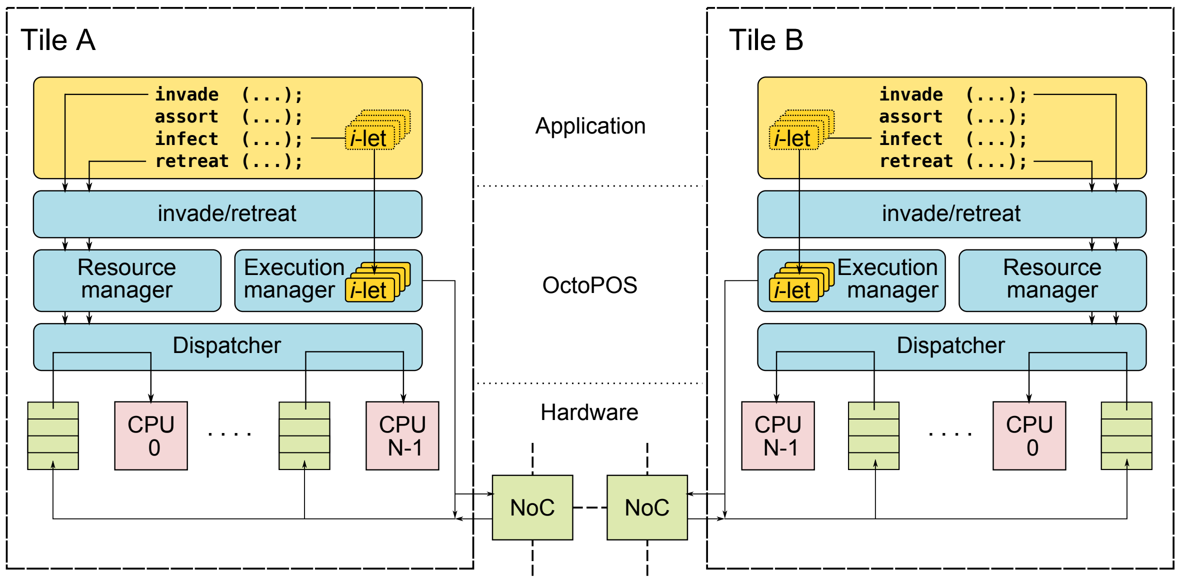


Figure 3.12.: Architectural overview of the invasive hardware/software stack [10]

environment. This includes, but is not limited to, all facilities that deal with resource allocation, task creation, execution and their synchronization.

Another constructive measure towards scalability is the tailoring of operating-system functionality to the needs of the application, since providing superfluous features often leads to unnecessary overhead during execution. One example of this is the preemption of running control flows: Invasive applications can have exclusive access to the hardware they reserve, for example the processing elements. If this is the case, temporal isolation through preemption mechanisms (for example, in order to prevent CPU monopolization by an application) becomes superfluous.

3.3.3. OctoPOS - Architectural Overview

The overall system design of OctoPOS is depicted in Figure 3.12, exemplarily showing two instances of OctoPOS running on two compute tiles.

The *resource manager* implements the basic invasive functions *invade* and *retreat* that are used to create, manage and destroy claims containing processing elements on local or remote tiles. The claims and the actual user program supplied to the *execution manager* as a team of *i*-lets are then used as an input to *infect* to start execution. To provide an efficient execution environment, *i*-lets are regarded as mostly run-to-completion functions that can run in parallel. They are represented as a pair of pointers, one for the function to be executed and one for the input data. Due to its run-to-completion semantics, an *i*-let leaves no runtime state on the stack after it has terminated. The *i*-let descriptors are stored in processor-local *i*-let queues and can be executed by the *dispatcher* like normal functions one after another on the same execution context, without the need for an expensive context switch in between. However, in the exceptional case that an *i*-let executes a blocking OS function (i.e., does not run to completion), OctoPOS moves the current context to a waiting queue and transparently provides a new execution context for the following *i*-lets.

Data dependencies between *i*-lets are expressed implicitly by means of synchronization primitives in the code. These synchronization operations come in two variants:

- *Blocking* primitives use the above mechanism to stall the calling *i*-let as long as the waiting condition holds.
- *Infecting* primitives schedule a new *i*-let as soon as the waiting condition is no longer satisfied.

Programming errors or malicious code can lead to deadlocks or infinite loops, and there are no guarantees that an application will terminate and release its resources. This can be countered by assigning time budgets to applications and enforcing them using a watchdog timer. However, dealing with misbehaving applications is not considered in the first phase of the Invasive Computing project.

To cross tile boundaries, the network-on-chip provides dedicated operations which were co-designed with the operating system. These operations allow OctoPOS to start *i*-lets on a different OctoPOS instance residing on another tile, or to copy a block of data from one tile to another. Applications can use these services via specific system calls to communicate across cache-coherency boundaries.

In summary, OctoPOS is a scalable operating system that provides an efficient execution environment for applications following the paradigm of Invasive Computing. OctoPOS was used as a basis for the implementation of the resource-aware image-processing algorithms described in Chapter 4.

4. Computer Vision Algorithms for Humanoid Robots

In order to make robot autonomous, it is important that the robots can perceive the world around them. Computer vision is interesting in this context as it provides a large set of information about the changing environment. Through computer vision, the robot control systems generate plans for the robot to evolve in the environment and search or manipulate objects. Two different computer vision algorithm were used in this work to demonstrate the benefits and restrictions of Invasive Computing. The first algorithm used is the object recognition (described in Section 4.1) followed by optical flow algorithm (described in Section 4.2).

4.1. Object Recognition

Figure 4.1.: Humanoid robot ARMAR recognizing objects around it

Object recognition is the task within computer vision of finding and identifying objects in an image or video sequence. Object recognition in unstructured scenes is a challenging area of ongoing research in computer vision. One important application lies in robotics, where the ability to quickly and accurately identify objects of interest, is crucial for general-purpose robots to perform tasks in unstructured everyday environments such as households and offices. Real world environments are highly cluttered, contain many occlusions and frequently contain five or ten different objects in the same scene. Robots must often manipulate objects in their environment, for example, grasping a juice bottle. This means that a robotic perception system needs to accurately localize objects after detecting them. Figure 4.1 shows the humanoid robot ARMAR robot operating in such a cluttered environment. Additionally, for a robot to react quickly to changes in its environment, a robotic perception system needs to operate in real-time.

The development of image matching by using a set of local key-points can be traced back to the work of Moravec [76]. He defined the concept of "points of interest" as being distinct regions in images that can be used to find matching regions in consecutive image frames. The Moravec operator was further developed by Harris and Stephens [77] who made it more repeatable under small image variations and near edges. Schmid and Mohr [78] used Harris corners to show that invariant local features matching could be extended to the general image recognition problem. They used a rotationally invariant descriptor for the local image regions in order to allow feature matching under arbitrary orientation variations. Although it is rotational invariant, the Harris corner detector is however very sensitive to changes in image scale so it does not provide a good basis for matching images of different sizes. Lowe [79, 80, 81] overcome such problems by detecting the points of interest over the image and its scales through the location of the local extrema in a pyramidal Difference of Gaussians (DOG). The Lowe's descriptor, which is based on selecting stable features in the scale space, is named the Scale Invariant Feature Transform (SIFT). Mikolajczyk and Schmid [82] experimentally compared the performances of several currently used local descriptors and they found that the SIFT descriptors to be the most effective, as they yielded the best matching results. Due to the high effectiveness of the SIFT descriptor, it has been used on the ARMAR robot for object recognition. Experiments shows that the SIFT key-points features are highly distinctive and invariant to image scale and rotation providing correct matching in images subject to noise, viewpoint and illumination changes.

A technique to minimize the computational time of the SIFT algorithm was presented in [11]. The algorithm in [11] computes features with high repeatability and very good matching properties in real-time. Object recognition is performed by comparing the features from the reference image with real-time input from a camera. The objects to be recognized at run time are known beforehand and their features are stored in a database. At runtime, the algorithm looks for matching features in the input frame to detect objects as shown in Figure 4.2.

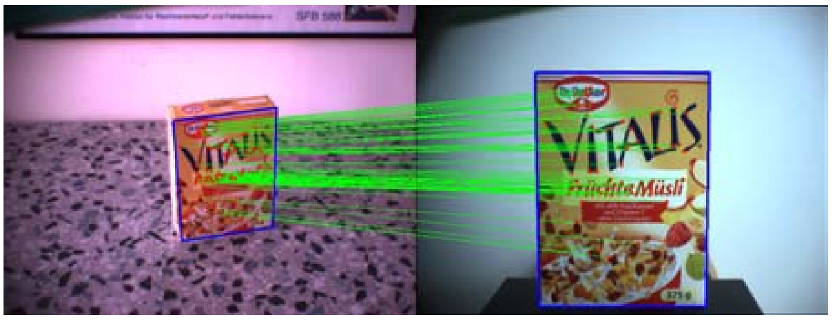


Figure 4.2.: Harris SIFT object recognition [11]

The left sub-image is the real-time input from the camera mounted on the ARMAR-III robot head [83], while the template stored in the database is shown on the right. In this work, the object recognition algorithm (based on Harris corner detector and SIFT) presented in [11] was used to analyze the performance benefits offered and challenges posed by the heterogeneous MPSoC presented in Section 3.2.

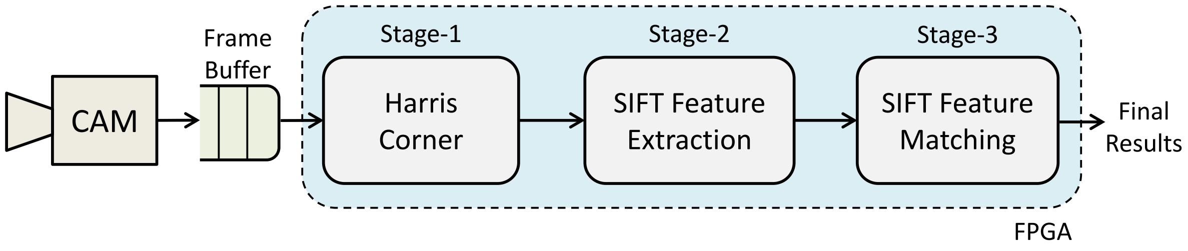


Figure 4.3.: Data Flow

As shown in Figure 4.3, the algorithm in [11] consists of three main stages. The Harris corner detection is used as the first stage, where a set of corner points are detected in the input image (captured by the camera mounted on the robot). The coordinates of the detected corners are then passed on to the second stage of the algorithm to extract SIFT features. In this step, a SIFT feature is computed for every corner point in the input image. Finally the SIFT features extracted from the image and compared to the SIFT features from the training image. In order to increase the speed of SIFT feature matching, a method based on K-Dimensional Tree (kd-tree) has been used. A match count above a threshold value indicates that the target object has been detected within the input image. The remainder of this chapter explains the details of each stage within the object recognition algorithm, used on the ARMAR robot.

4.1.1. Harris Corner Detection

Corner detection is an approach used within computer vision systems to extract certain kinds of features and infer the contents of an image. A corner is the intersection of two edges in an image. Since it is the intersection of two edges, it represents a point in which the directions of these two edges change. Hence, the gradient of the image (in both directions) have a high variation, which can be used to detect it. Corner detection is frequently used in motion detection, image registration, video tracking, image mosaicing, panorama stitching, pattern recognition, etc. Corners are also used as robust image representation when combined with feature descriptors for object recognition.

A fundamental step in these applications is the detection of corners which represent identifiable anchor points in the image. Several corner detectors exist today in literature and comparative evaluations have shown that the Harris [77] (see Figure 4.4) and Shi-Tomasi [84] detectors achieve some of the best results. Also, a wide range of real-time applications [85] [86] [87] [88] have used Harris and Shi-Tomasi detectors.

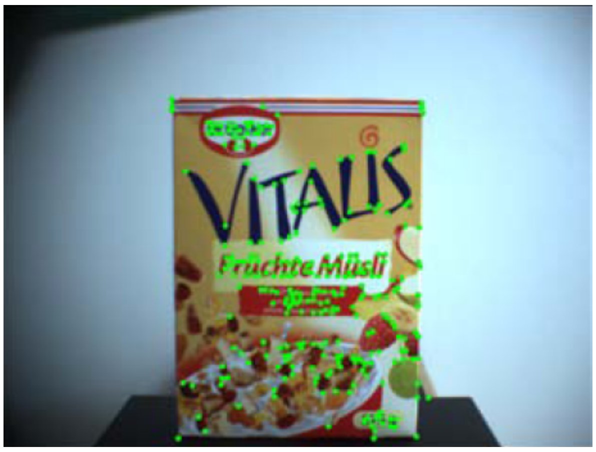


Figure 4.4.: Output from Harris Corner detection algorithm (detected corners are highlighted using green dots)

4.1.1.1. Basics of Corner Detection

This section provides a brief overview of the Harris and Shi-Tomasi corner detectors. To detect corners, both algorithms test each pixel in the image by considering how similar a patch centered on the pixel is to nearby, largely overlapping patches. If the pixel is in a region of uniform intensity, then the nearby patches will look similar. If the pixel is on an edge, nearby patches in a direction perpendicular to the edge will look different, but parallel to the edge will result only in a small change. If the pixel is on a feature with variation in all directions, then none of the nearby patches will look similar. The calculation is based on the local auto-correlation function that is approximated by matrix M over a small window w for each pixel $p(x,y)$:

$$M = \begin{bmatrix} \sum_w W(x)I_x^2 & \sum_w W(x)I_xI_y \\ \sum_w W(x)I_xI_y & \sum_w W(x)I_y^2 \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \tag{4.1}$$

I_x and I_y are horizontal and vertical intensity gradients, respectively, and $W(x)$ is an averaging filter that can be a box or a Gaussian filter. The eigenvalues λ_1 and λ_2 (where $\lambda_1 \geq \lambda_2$) indicate the type of intensity change in the window w around $p(x,y)$:

1. If both λ_1 and λ_2 are small, $p(x,y)$ is a point in a flat region.
2. If λ_1 is large and λ_2 is small, $p(x,y)$ is an edge point.
3. If both λ_1 and λ_2 are large, $p(x,y)$ represents a corner point.

Shi-Tomasi directly computes the smaller eigenvalue λ_2 as its corner measure C as shown in Equation (4.2):

$$C = \lambda_2 = \frac{(a+c) - \sqrt{(a-c)^2 + 4b^2}}{2} \tag{4.2}$$

Harris combines the eigenvalues into a single corner measure R as shown in Equation (4.3), which avoids the explicit computation of eigenvalues (k is an empirical constant with value 0.04 to 0.06).

$$R = \lambda_1\lambda_2 - k \cdot (\lambda_1 + \lambda_2)^2 = (ac - b^2) - k \cdot (a + c)^2 \tag{4.3}$$

Once the corner measure is computed for every pixel, a threshold is applied on the corner measures to discard the obvious non-corners. The rest of the pixels are then ranked in the descending order of the corner measure. After applying the non-maximal suppression, the pixels with the highest corner measures are then selected as corners.

4.1.2. SIFT Feature Extraction

This section describes the SIFT Feature Extraction algorithm used on the ARMAR robot. As stated in [11], the SIFT descriptor is a very robust and reliable representation for the local neighborhood of an image point. However, the scale-space analysis required for the calculation of the SIFT feature point positions is too slow for visual servoing applications. One of the main strengths of the SIFT features are their scale-invariance. This is achieved by analyzing and processing the images at different scales. For this, a combination of Gaussian smoothing and a re-size operation is used. Between two so-called octaves, the image size is halved, i.e. re-sized to half width and half height. The different scales within an octave are produced by applying a Gaussian smoothing operator, and the variance of the Gaussian kernel is chosen in a way that the last scale of one octave and the first scale of the next octave correspond to each other. Since the scale space analysis performed by the SIFT features for calculating the feature point positions is the by far most time-consuming part, the idea presented in [11] is to replace this step by a faster method, namely an appropriate corner detector.

The Harris corner detector is a suitable starting point for the computation of positions of scale and affine invariant features. At each interest point as obtained above, an image descriptor is computed. The SIFT descriptor proposed by Lowe (1999, 2004) can be seen as a position-dependent histogram of local gradient directions around the interest point. To obtain scale invariance of the descriptor, the size of this local neighborhood needs to be normalized in a scale-invariant manner. Now an orientation is assigned to each key-point to achieve invariance to image rotation. A neighborhood is taken around the key-point location depending on the scale, and the gradient magnitude and direction is calculated in that region. An orientation histogram with 36 bins covering 360 degrees is created. The highest peak in the histogram is taken and any peak above 80% of it is also considered to calculate the orientation. It creates key-points with same location and scale, but different directions.

Figure 4.5 illustration of how the SIFT descriptor is computed from sampled values of the gradient orientation and the gradient magnitude over a locally adapted grid around each interest point, with the scale factor determined from the detection scales of the interest point and the orientation determined from the dominant peak in a gradient orientation histogram around the interest point. In this example, a 16x16 neighborhood around the key-point is

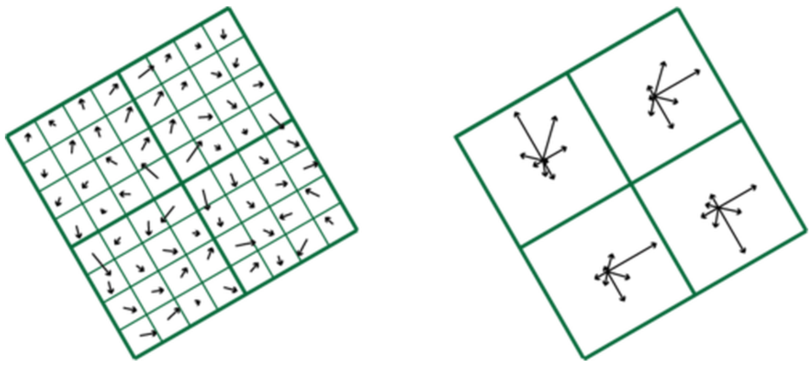


Figure 4.5.: Computation of SIFT descriptor from sampled pixels

taken. It is divided into 16 sub-blocks of 4×4 size. For each sub-block, 8 bin orientation histogram is created. So a total of 128 bin values are available. It is represented as a vector to form key-point descriptor.

To obtain contrast invariance, the SIFT descriptor is normalized to unit sum. In this way, the weighted entries in the histogram will be invariant under local affine transformations of the image intensities around the interest point, which improves the robustness of the image descriptor under illumination variations.

The SIFT descriptors computed using the above technique are invariant to image scaling, rotation, change in illumination and camera viewpoint. They are well localized in both the spatial and frequency domains, reducing the probability of disruption by occlusion, clutter, or noise. SIFT descriptors have been shown to be well suited to image matching and recognition as well as to many applications as they are robust to occlusion, background clutter and other content changes.

4.1.3. SIFT Feature Matching

For object recognition, every SIFT feature extracted from the real-time input image (captured by the camera mounted on the head of the robot) has to be compared with the SIFT features extracted from the training image. The number of SIFT features in the real-time input image may vary from time to time based on the nature of the objects present in the frame. In a typical application, a large number of SIFT descriptors extracted from one or many images are stored in a database. Generally it varies from few hundred features to thousand or more features in a cluttered scene. A query usually involves finding the best matched descriptor vector(s) in the database to a SIFT descriptor extracted from a query image. Comparing each SIFT feature (from the real-time input) with every feature in the training image is an extremely compute intensive process. In order to increase the speed of SIFT feature matching, a method based on KD-tree has been presented in [89]. KD-tree (which is a form of balanced binary search tree) is a useful data-structure for finding nearest-neighbors of various image descriptors.

A kd-tree is a binary tree in which every node is a k-dimensional point. Every non-leaf node generates a splitting hyperplane that divides the space into two subspaces. Points left

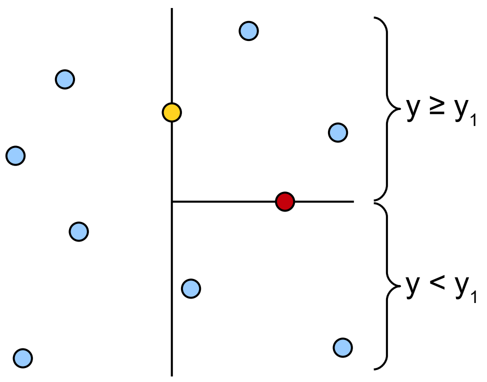


Figure 4.6.: Construction of KD-Tree (recursively build a kd-tree in the right half-space by picking the some point and splitting the data horizontally through it)

to the hyperplane represent the left sub-tree of that node and the points right to the hyperplane by the right sub-tree. The hyperplane direction is chosen as follows: Every node split to sub-trees is associated with one of the k-dimensions, such that the hyperplane is perpendicular to that dimension vector. For example, if a particular split the `x-axis` is chosen, all points in the sub-tree with a smaller `x` value than the node will appear in the left sub-tree and all points with larger `x` value will be in the right sub-tree (see the example shown in Figure 4.6). The canonical method of kd-tree construction has the following constraints:

1. As one moves down the tree, one cycles through the axes used to select the splitting planes.
2. Points are inserted by selecting the median of the points being put into the sub-tree, with respect to their coordinates in the axis being used to create the splitting plane.

This method leads to a balanced KD-tree (as shown in Figure 4.7), in which each leaf node is about the same distance from the root.

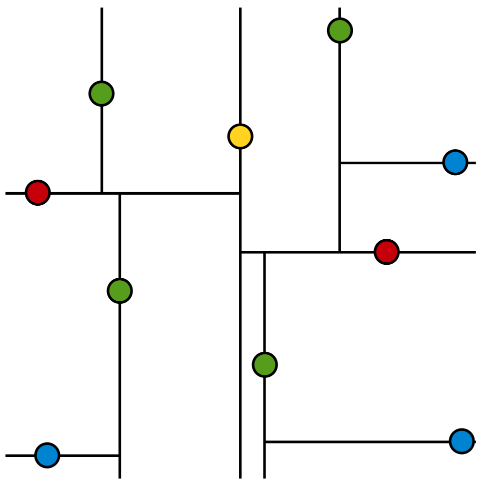


Figure 4.7.: A completely constructed KD-Tree

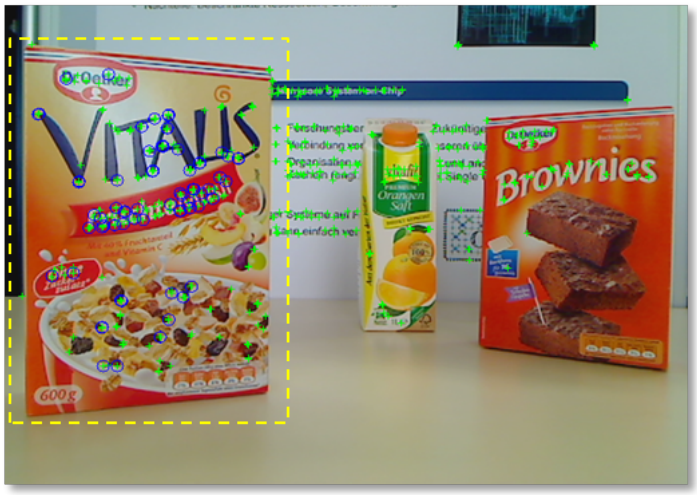


Figure 4.8.: Result from Harris-SIFT based object recognition technique

4.1.3.1. Nearest Neighbor Search Algorithm

In order to find matching SIFT features a technique known as Nearest neighbor search is used. Nearest neighbor search is an important component in many computer vision applications. At the first level (root) of the tree, the data is split into two halves by a hyperplane orthogonal to a chosen dimension at a threshold value. Generally, this split is made at median in the dimension with the greatest variance in the data set. By comparing the query vector with the partitioning value, it is easy to determine to which half of the data the query vector belongs. Each of the two halves of the data is then recursively split in the same way to create a fully balanced binary tree. At the bottom of the tree, each tree node corresponds to a single point in the data set; though in some implementation, the leaf nodes may contain more than one point. The height of the tree will be $\log_2 N$, where N is the number of points in the data set. Given a query vector, a descent down the tree requires $\log_2 N$ comparisons and leads to a single leaf node. The data point associated with this first node is the first candidate for the nearest neighbor. The first candidate will not necessarily be the nearest neighbor to the query vector; it must be followed by a process of backtracking, or iterative search, in which other cells are searched for better candidates. The recommended method is priority search in which the cells are searched in the order of their distance from the query point. This may be accomplished efficiently using a priority tree for ordering the cells; in which the cells are numbered in the order of their distance from the query vector. The search terminates when there are no more cells within the distance defined by the best point found so far. In high dimensions to find the nearest neighbor may require searching a very large number of nodes. This problem may be overcome at the expense of an approximate answer by terminating the search after a specified number of nodes are searched (or earlier if the search terminates).

A sample output from the algorithm mentioned above is shown in Figure 4.8. The input image contains three different object and a cluttered background resulting in numerous Harris corners (marked in green). SIFT descriptors were computed at every corner point and matched with the descriptors extract from the training image. The matched descriptors are circled in blue. A match count above the threshold value leads to a positive result and the detected object is marked using a bounding box.

In addition to the object recognition algorithm, this work also utilize another algorithm called optical flow to demonstrate the befits of Invasive Computing. The details of the optical flow algorithm is presented in Section 4.2.

4.2. Optical Flow

Optical flow or optic flow is the pattern of apparent motion of objects, surfaces, and edges in a visual scene caused by the relative motion between an observer (an eye or a camera) and the scene. Currently, optical flow plays an increasingly important role in the robotics domain where Optical flow techniques can be used for motion detection, object segmentation, time-to-collision and focus of expansion calculations, motion compensated encoding, obstacle avoidance [90], etc.

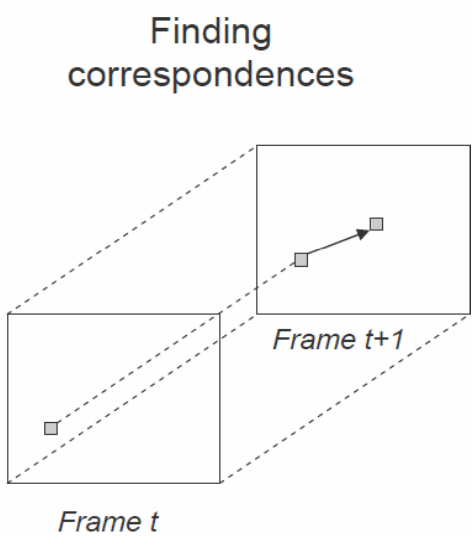


Figure 4.9.: Optical flow definition

Optical flow determine the correspondences between consecutive images as shown in Figure 4.9. In other words, pixels from frame (t) have to be identified in frame (t + 1) and simultaneously where they are changed. From the original coordinates (x_a, y_a) to the changed coordinates (x_b, y_b) a vector can be drawn, defined as motion vector.

The optical flow computation can be done in different ways. One of them is using census transformation as described in [91]. The optical flow computation using census transformation involves two stages. To obtains the information about pixel motion between two consecutive frames, a signature is generated for every pixel in frame (t). The signatures are generated by comparing the pixel value of the current pixel to every other pixel in the neighborhood, as shown in Figure 4.10.

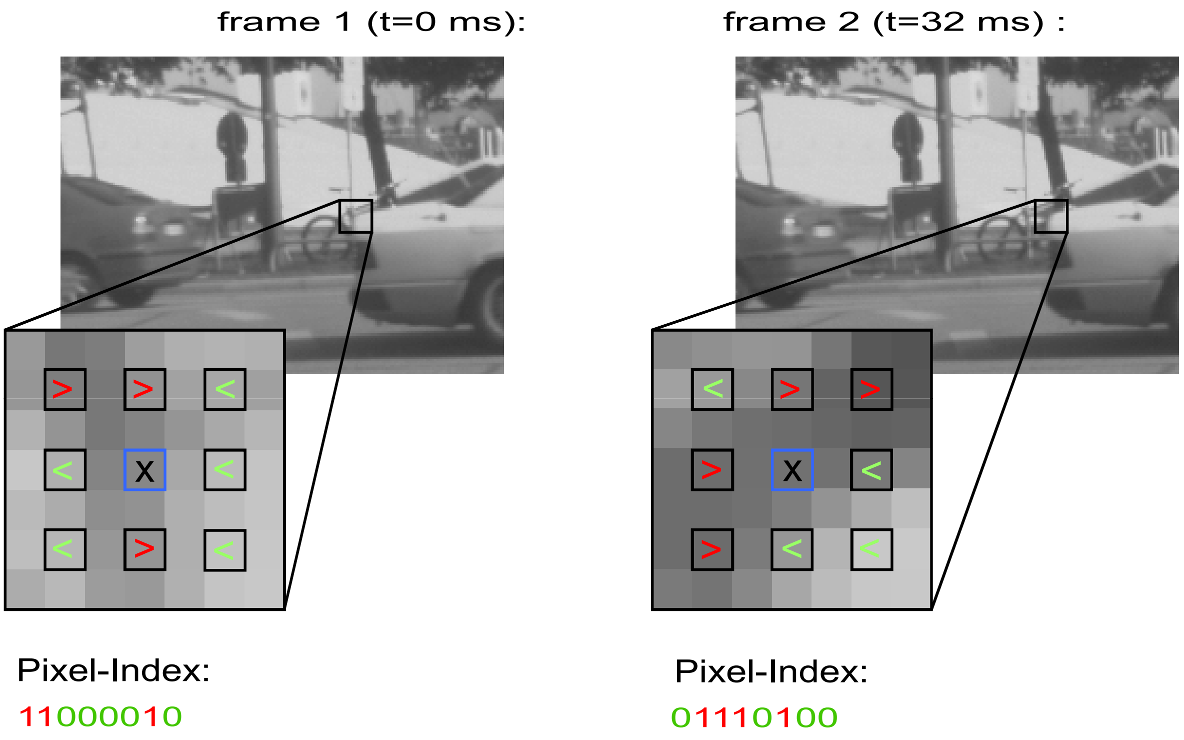


Figure 4.10.: Census transformation

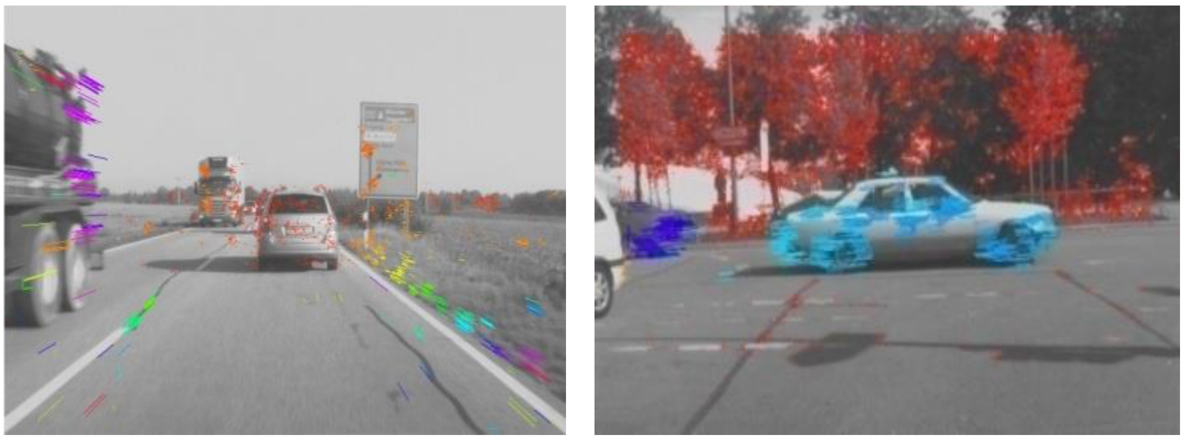


Figure 4.11.: Output from Optical flow algorithm

Then the signatures of pixel p in frame (t) is compared to signature of pixels in frame (t+1). Two pixels (one from frame (t) and another from frame (t+1)) correspond to each other if their signatures match. In that case, one pixel is known as the correspondence of the another one. Figure 4.11 shows a sample output from optical flow algorithm. The flow vectors around the moving car indicates its velocity. The flow vectors on the background are just dots representing zero motion (static camera).

5. Resource-Aware Algorithms

Humanoid robots fulfill many different tasks during operation including recognizing objects, navigating towards a goal region, avoiding collisions, maintaining balance, planning motions, having conversations, and many more. Additionally they have to work in a highly dynamic and ever changing human centered environments. Taking a closer look, all of these tasks have different requirements depending on their objectives. Some tasks are computationally or data-flow intensive and some are control flow intensive. Some are continuously running at a specific frequency while others are doing their work asynchronously triggered by external events.

Robot-specific tasks need to be executed on the hardware built into the robot. Current humanoid robot control architectures share similar hardware setups consisting of one or

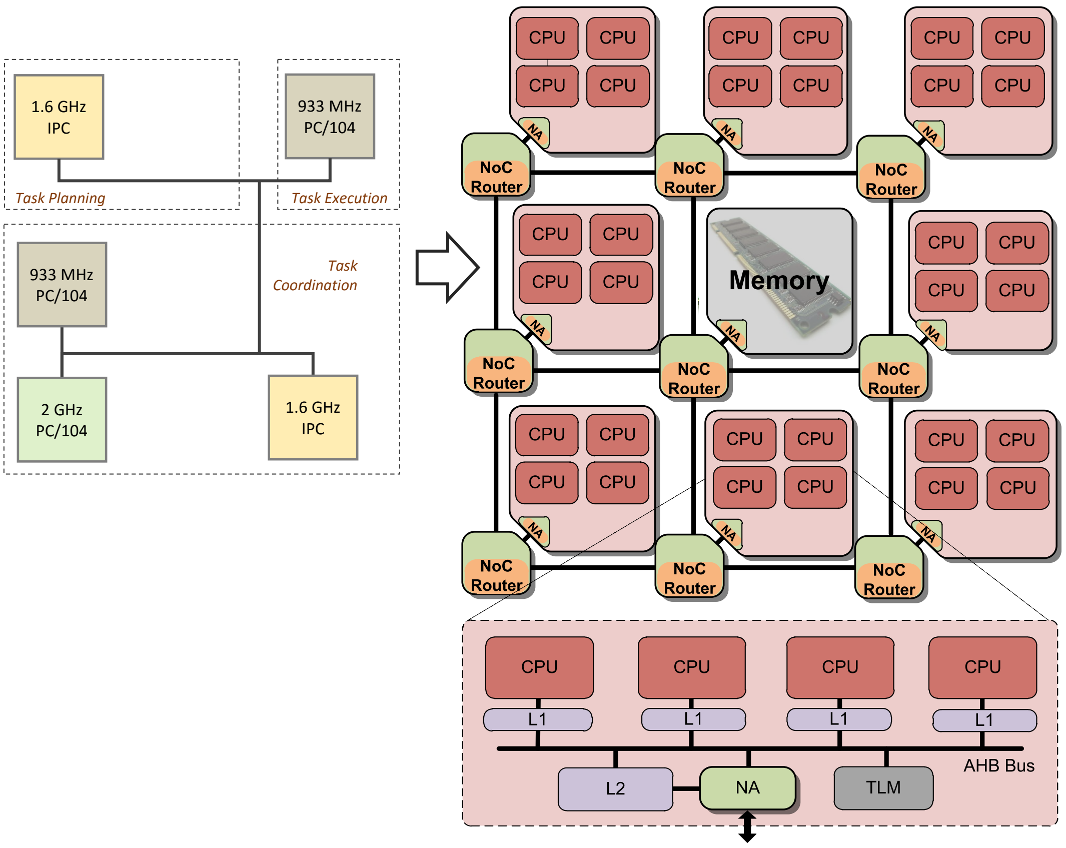


Figure 5.1.: Conventional compute units used on the ARMAR robot (left) and the newly proposed MPSoC architecture (right)

more industrial PCs equipped with single- or dual-core processors, as shown in Figure 5.1. Timely execution of tasks is guaranteed as the underlying hardware is selected to be able to cope with worst case requirements. Usually tasks are distributed and mapped onto a number of PCs, making migration of tasks between PCs difficult. Each compute unit is used only when the robot performs a task mapped to that compute unit or CPU.

Merging the various tasks on to a unified computing platform like an MPSoC (as shown in Figure 5.1) can offer several advantages including easy task migration, better communication bandwidth between processing units, reduced power consumption by avoiding off-chip interconnects, etc. Furthermore, in order to leverage the capabilities of today's robots it is desirable to make use of the abundant computing power offered by MPSoCs by executing many independent tasks in parallel. On the other hand, however, the mapping of various tasks onto a unified MPSoC automatically results in sharing of physical resources like PEs and interconnect/memory bandwidth between the once separated tasks. A common practice as of now is to do static resource allocation at compile time as described in [92]. However, static allocation schemes does not achieve the best resource utilization as the applications cannot share compute resources. Here, often changing requirements lead to underutilization since resources are often occupied while not performing any work.

Another approach to distribute the workload onto the numerous available resources is to use an operating systems with support for symmetric multiprocessing (SMP), like on the Intel's SCC. However, they need very complex and sophisticated application mapping and thread scheduling algorithms in order to achieve good performance figures and tend to face scalability issues [93]. Another disadvantage is the required hardware homogeneity (no specialized hardware possible).

For a humanoid robot like ARMAR, various applications have to run on the MPSoC in order to achieve a complex task. For example, the various algorithms combined to perform an object grasping is shown in Figure 5.2.

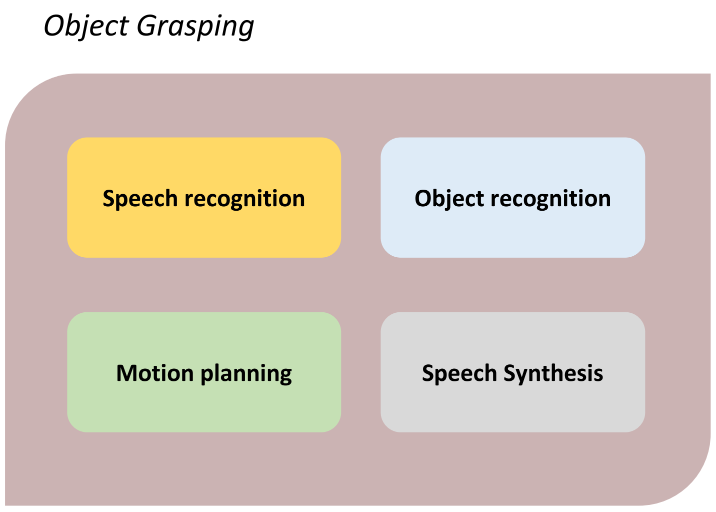


Figure 5.2.: Various algorithms combined to perform object grasping

Each algorithm has its own computing power requirements to produce results within a specified time and to ensure a smooth operation of the robot. Sharing of resources amoung various applciation running concurrently may lead to unpredictable execution time and poor quality results. This issue is demonstrated below using the example of Harris Corner detection algorithm.

Corner detection is often employed as the first step in computer-vision applications with real-time video input. Hence, the application has to maintain a steady throughput and good response time to ensure quality results. However, the presence of other high-priority tasks may alter the behavior of the corner-detection algorithm. To evaluate such a dynamically changing situation, we analyzed the behavior of the conventional Harris detector on an MP-SoC with 32 PEs. In the first model (Figure 5.3, left) the Harris Corner detection algorithm was scheduled statically on 9 PEs. A video input with 640×480 pixels at 10 frames per second was used, with the test running for 20 seconds.

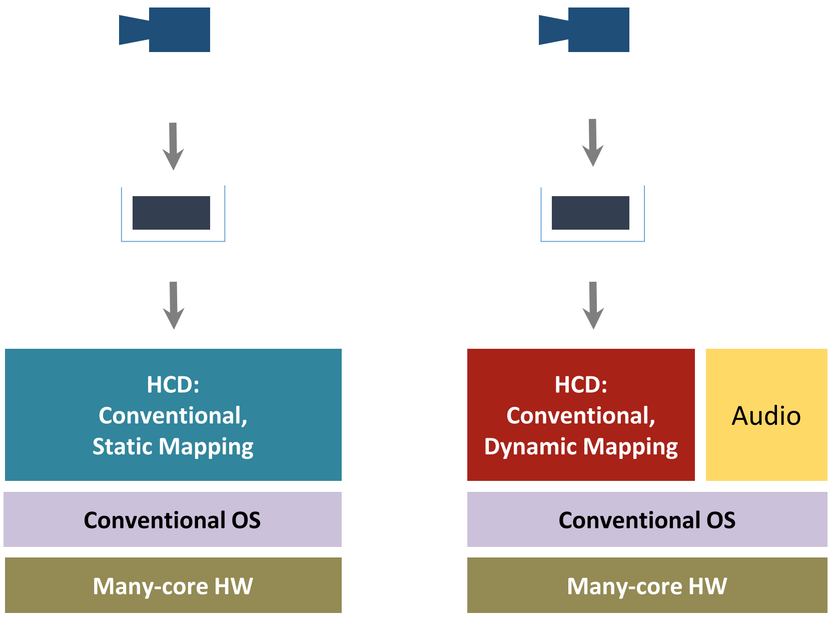

HCD: Conventional, Static Mapping
Conventional OS
Many-core HW
HCD: Conventional, Dynamic Mapping
Audio
Conventional OS
Many-core HW

Figure 5.3.: Evaluation model for Harris Corner detection with dedicated HW (left) vs. shared HW (right)

In order to evaluate the impact of other applications running concurrently on the MP-SoC system, an audio processing application was used, as shown on the right hand side in Figure 5.3. The audio processing application creates dynamically changing load on the processor based on what the robot is doing at that point in time. For instance, the speech-recognition application is activated when the user speaks to the robot. The conventional OS scheduler schedules the threads of the applications based on the overall system load. Figure 5.4 shows the number of PEs allocated to the Harris Corner detection application at runtime. Sharing of available resources resulted in the execution-time profile shown in Figure 5.5. It can be seen that the execution time varies from 0 to 430 milliseconds, based on the load condition. A lack of sufficient resources leads to very high processing intervals or frame drops (a processing interval of zero represents a frame drop). The number of frames dropped during this evaluation is as high as **20%** and the worst-case latency increased by 4.3× (100 milliseconds to 430 milliseconds). Frame drops reduce the quality of the results and the robot may lose track of the object if too many consecutive frames are dropped. Figure 5.5 shows the execution-time profile based on the resource allocation scheme in Figure 5.4.

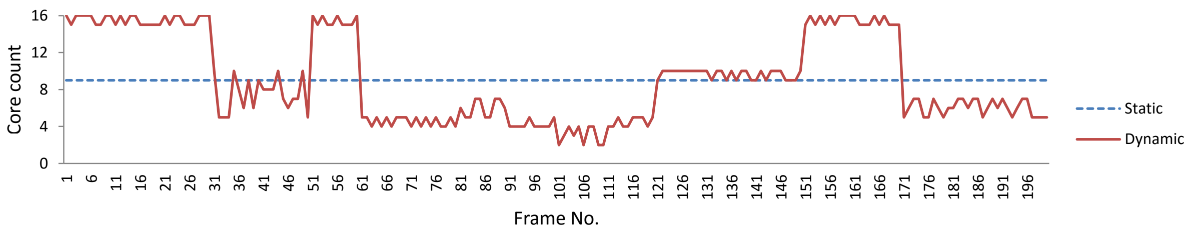


Figure 5.4.: Variation in available resources for Harris Corner detection [12]

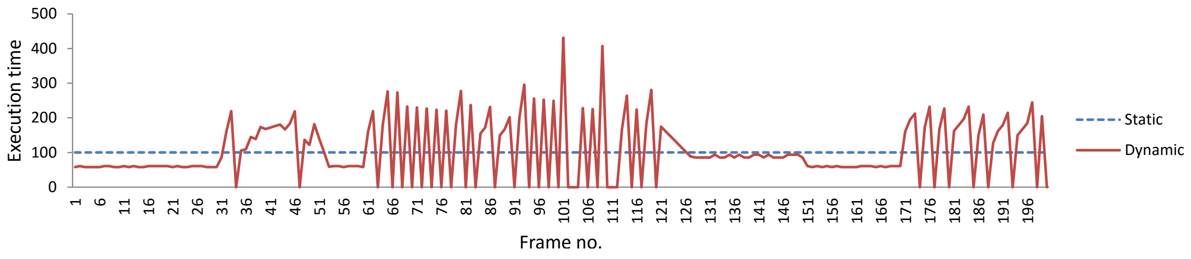


Figure 5.5.: Variation in processing interval based on available resources [12]

Within the resource-aware application model, every application program perform the steps as shown in Figure 5.6 in order to spread its computations to neighboring PEs and to achieve its throughput constrains like frames-processed/second (fps). As a results every application program knows in advance (after the *invade* phase and before the *infect* phase), the amount of computing resources (for example the number of PEs) allocated by the OctoPOS for that *invade* request. This information may be used by the application program to adapt its workload at runtime. The rest of this section explores the possibilities, benefits and restrictions of extending conventional algorithms like Harris Corner detection, SIFT feature matching, etc., into resource-aware models. The following sections also demonstrate how the resources are claimed and how the processing interval can be constrained to guarantee consistent throughput and processing intervals.

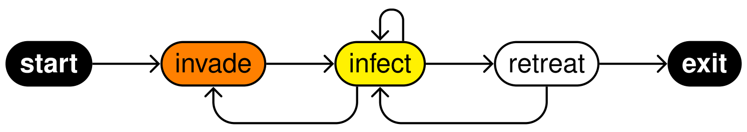


Figure 5.6.: Structure of an invasive program

5.1. Resource-Aware Corner Detectors

Corner detection is a fundamental step in various computer vision applications as the detected corners represent identifiable anchor points for further processing. Corners are also used as robust image representation when combined with feature descriptors for object recognition. Several corner detectors exist today in literature and comparative evaluations have shown that the Harris [77] and Shi-Tomasi [84] detectors achieve some of the best results. Also, a wide range of real-time applications [85] [86] [87] [88] have used Harris and Shi-Tomasi detectors. A brief overview of the corner detection algorithm is available under 4.1.1.1.

5.1.1. Conventional Approach Towards Faster Corner Detection

Independent of the algorithm used, corner detection is a compute-intensive step. Two different approaches have been used to increase the throughput of corner detection. The first approach focuses on hardware accelerators or graphics processing units (GPUs) to accelerate the conventional algorithm while the second employs algorithmic techniques to reduce the computational complexity. A high throughput can be guaranteed using hardware accelerators based on field-programmable gate arrays (FPGAs). However, FPGAs offer rather little flexibility and require very high effort in terms of design, implementation and verification. GPUs, on the other hand, are very powerful and provide significant acceleration over small multi-core processors, but they consume very high power and require data transfers between the processor and the hardware accelerator, which increases overall latency.

Hence, a more suitable approach is algorithmic optimization, which includes a pruning technique based on gradient magnitude that selects pixels with a high gradient magnitude as corner candidates for the Shi-Tomasi and Harris algorithms [94]. Another pruning technique to reduce the computational complexity of the conventional Harris detector is described in [95]. This technique relies on the fact that in most situations, the obvious non-corners constitute a large majority of the image. Hence the corner detectors incur a lot of redundant computations as they evaluate the entire image for a high corner response. Such a scenario is depicted in Figure 5.7 where the background wall does not contain an corners while the foreground has several corners. The conventional Harris Corner detection algorithms does not distinguish between plain and cluttered region and hence they encounter redundant computations. Section 5.1.2 describes how the conventional corner detectors were extended to resource-aware models making use of the Invasive Computing methodology.

5.1.2. Extension to a Resource-Aware Model

The threshold-based pruning technique in [94] was extended to a resource-aware pruning technique in order to control the workload for the corner detection based on available resources on the MPSoC. In the first step, the algorithm considers the corner point as the junction of two or more edge lines. One important property of such a point is that it has a high gradient change in more than one direction. A corner response (CR) function can be defined from the above property as:

$$CR = \left(|I_x| \cdot |I_y|\right) \tag{5.1}$$

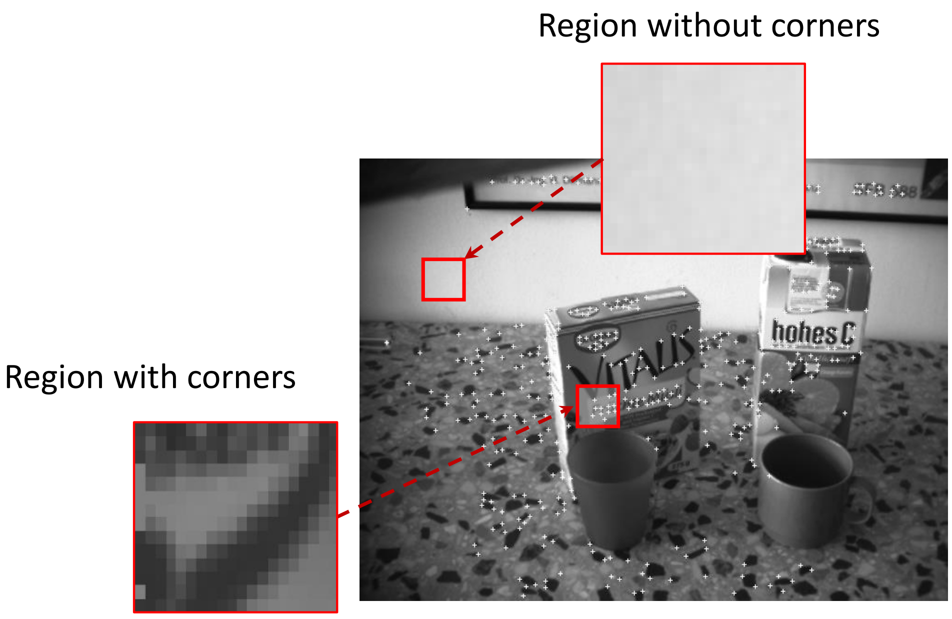


Figure 5.7.: Harris corner detection (foreground and background pixels)

where I_x and I_y are the horizontal and vertical pixel-intensity derivatives. If CR is greater than a predefined gradient threshold, the pixel is a corner candidate and should be retained for processing in the subsequent steps. This pruning technique is visualized in Figure 5.8, where the image on the left is a preprocessed image where every pixel is computed using Equation (5.1). Applying a threshold on the *CR* values results in the image on the right (Figure 5.8). Any pixel with value above the threshold is set to a value of "255" (white) and the remaining pixels gets the value "0" (black). The resulting image can now be used as a mask for the rest of the *harris-map* computations.

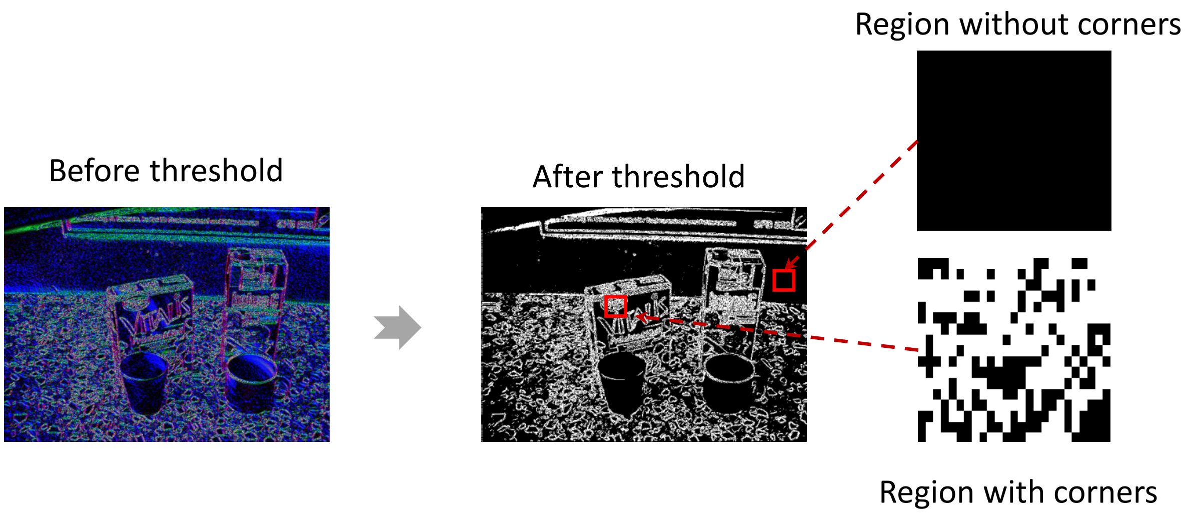


Figure 5.8.: Pruning technique used in resource-aware Harris Corner detection

This technique ensures that the non-corner pixels are removed prior to more intensive processing. From (4.3), R is most influenced by the term $(ac - b^2)$ as the two $(a + c)$ terms cancel out. For a good corner, R needs to be a large value. Therefore, maximizing $(ac - b^2)$ can select good corners without explicit eigenvalue computation. Hence, the new algorithm successively reduces the number of candidate corners at every step to minimize the computational effort. In the next step, an intermediate corner non-maxima suppression is applied to reduce the effect of false corners caused by multiple responses of the gradient operator. All candidate corners from the previous steps are further assessed through computing the eigenvalues as in the conventional Harris (4.3) or Shi-Tomasi detector (4.2). Finally, a non-maxima suppression is applied to suppress the corners that are close to each other.

Some of the challenges posed by the conventional corner detector on an MPSoC can be resolved using the pruning technique described above. The main idea and novelty of the resource-aware corner detector is that the threshold for pruning is based on the available resources. This means that the new detector can control the workload by changing the threshold. In situations where the system is under-utilized, the threshold can be reduced, whereby more pixels will be processed and a higher accuracy will be achieved. On the other hand, increasing the threshold can prune away more pixels when the processing system is heavily loaded by other high-priority tasks.

Figure 5.10 shows the relation between the threshold and the processing interval (pixels processed). The results were captured by applying the pruning technique on six different video sequences as shown in Figure 5.9 (each video sequence consists of 200 frames).

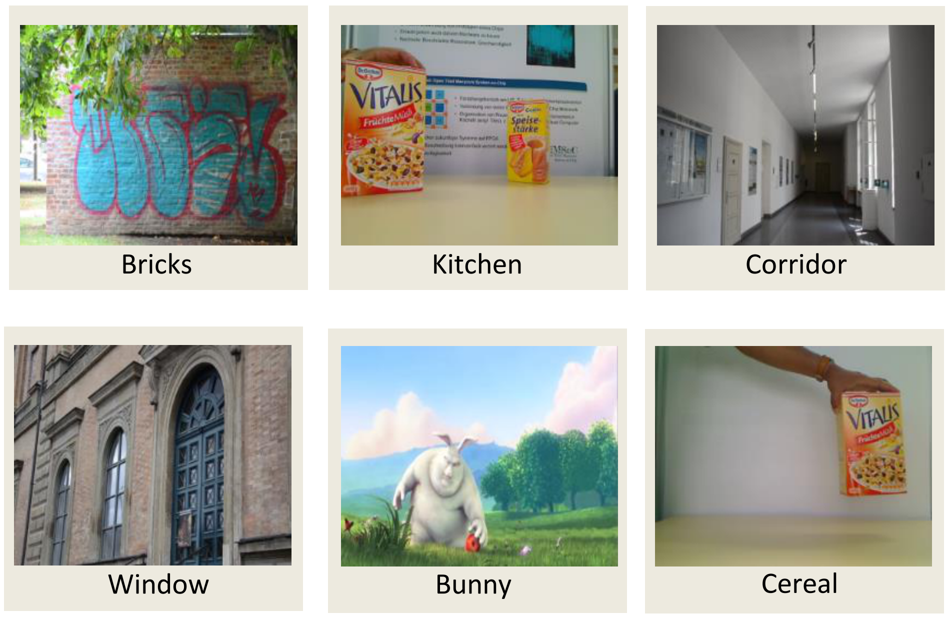


Figure 5.9.: Snapshots of the video sequences used for evaluation [10]

In order to evaluate the impact of pruning on the accuracy of detected corners, we use the metrics named *precision* and *recall* as proposed in [96]. Results indicate that the effects of pruning vary based on the scene. For instance, the speedup achieved (using the same threshold) is low for cluttered scenes like *Bricks* while the majority of the pixels can be pruned away for scenes with plain backgrounds (e.g., the moving cereal box). Figure 5.10 also shows the relation between threshold and accuracy (average of precision and recall rates) for all six video sequences, plotted independently of each other. Hence, the amount of computing resources required to perform the corner detection will vary from one scene to another based on the nature of the foreground, background, and so on. Therefore, the resources have to be allocated on a frame-to-frame basis, based on the scene captured. Section 5.1.3 explains the technique used to estimate resource demands based on application requirements, how to allocate and release these resources at runtime, and how to adapt to available resources at runtime.

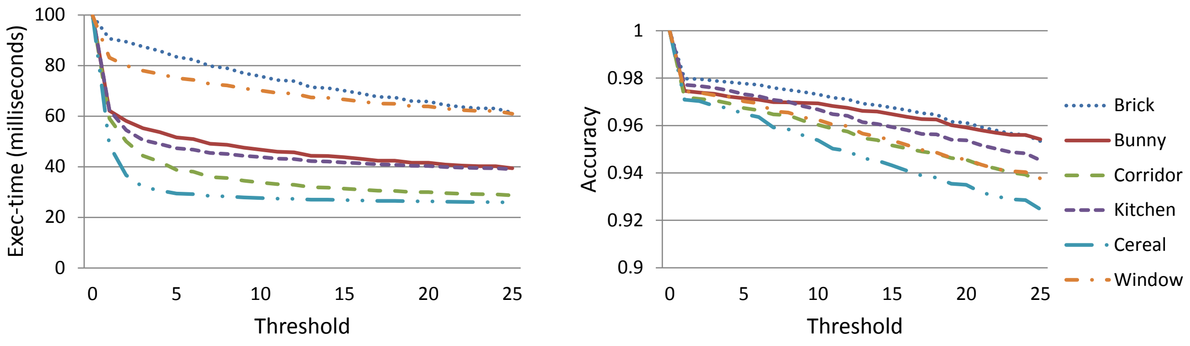


Figure 5.10.: Effects of pruning on execution time and accuracy for Harris detector [10]

5.1.3. Resource Allocation and Workload Distribution

The first step within the algorithm is to allocate sufficient resources to perform corner detection within the interval specified by the user. As shown in Figure 5.10, a threshold around 4 offers a significant speedup without much loss in accuracy. This region is interesting for most mobile platforms as the number of computations and hence power consumption can be reduced by operating in this range.

Choosing a threshold of 4 means that the computing resources (PEs) needed by the algorithm will vary over time. Figure 5.11 shows the PEs required for Harris Corner detection with a pruning threshold of 4. As the percentage of pixels that can be pruned depends on the nature of the scene, each scene type requires a different set of PEs. In some cases the PE count varies within the scene type due to minor changes within the video sequences. For example, as the object in the *Kitchen* scene moves towards the camera, the scene becomes more cluttered, the algorithm cannot prune away as many pixels as in the previous frame and hence to meet the deadline of 10 fps, the application has to *invade* more PEs.

The resource-aware corner-detection algorithm is structured as follows. The core algorithm consists of five stages as shown in Figure 5.12. The first stage is called a resource-estimation stage, where the image is pre-processed (differential-image and integral-histogram computation) to estimate the PEs needed based on the scene type and the user-specified deadline.

The analysis starts with the generation of a differential image, where each pixel is computed using (5.1). In order to speed up the pruning logic within the algorithm, an integral histogram is computed from the differential image as described in Algorithms 5.1 and 5.2,

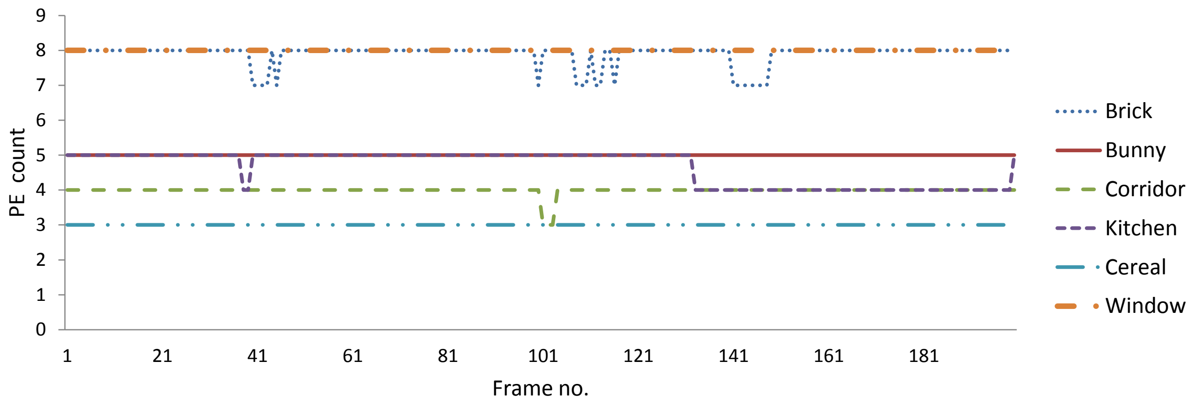


Figure 5.11.: Resource usage based on scene type (Harris detector using a threshold of 4) [10]

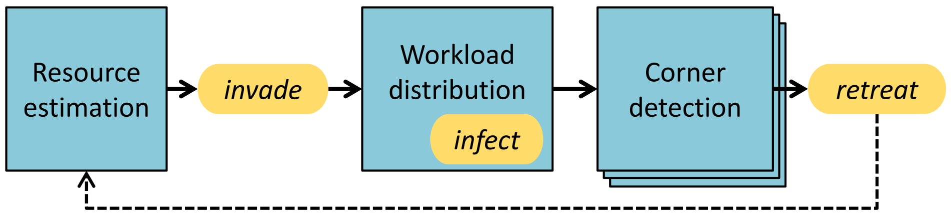


Figure 5.12.: Flow diagram for the resource-aware corner detector [10]

where n is the total number of pixels to be processed, I_{diff} is the differential image, dx and dy are the horizontal and vertical pixel-intensity derivatives, $limit$ is the maximum possible value generated by (5.1) and H is the integral histogram computed from differential image.

Algorithm 5.1 Differential image

```
1: $i \leftarrow 0$
2: $h \leftarrow 0$
3: **while** $i < n$ **do**
4:     $I_{diff}(i) \leftarrow |dx(i) \cdot dy(i)|$
5:     $h(I_{diff}(i)) \leftarrow h(I_{diff}(i)) + 1$
6:     $i \leftarrow i + 1$
7: **end while**
```

Algorithm 5.2 Integral histogram

```
1: $i \leftarrow 0$
2: $H \leftarrow 0$
3: **while** $i < limit$ **do**
4:     $k \leftarrow i$
5:     **while** $k < limit$ **do**
6:         $H(i) \leftarrow H(i) + h(k)$
7:         $k \leftarrow k + 1$
8:     **end while**
9:     $i \leftarrow i + 1$
10: **end while**
```

Once the integral histogram is computed, the values in the bin represent the number of pixels to be processed by the algorithm when the threshold is set to histogram-bin-index.

In the next stage, an *invade* request is raised to allocate sufficient PEs using Equation (5.2):

$$N_{pe} \geq \frac{(n \cdot T_{prn}) + (P_{pix}(th) \cdot T_{cd})}{T_{exe} \cdot \eta(N_{pe})} \tag{5.2}$$

where N_{pe} is the number of PEs needed for the computation, T_{prn} is the processing time per pixel until the end of the pre-processing stage, P_{pix} is the number of pixels to be processed by the upcoming stages (as computed by the pruning algorithm as a function of the threshold value th), T_{cd} is the time to compute the final corner measure for pixels with corner response (CR) above threshold and T_{exe} is the processing interval. $\eta(N_{pe})$ represents the algorithm's efficiency as a function of degree-of-parallelism or available resources. This term helps to increase the accuracy of the resource-estimation model, as the execution time for corner detection application does not decrease linearly with increasing degree of parallelism. This is due to the fact that some parts of the algorithm (workload distribution, merging final results, etc.) are performed by a single *i*-let, in a sequential manner. Moreover, additional *i*-lets created by the application also creates additional load on the external memory and shared communication interfaces, limiting the scalability.

An analysis conducted on the proposed InvasIC hardware resulted in an efficiency graph as depicted in Figure 5.13. From the graph it can be seen that when the number of *i*-lets is increased from 1 to 2, the execution time does not improve by 2x, instead by 2×0.98 (98%), that is, 1.96x. Using this graph, the efficiency factor for various levels of parallelism can be computed. The values shown here are applicable only for the Harris Corner detection implementation used in this work and may vary based on how the original algorithm is implemented.

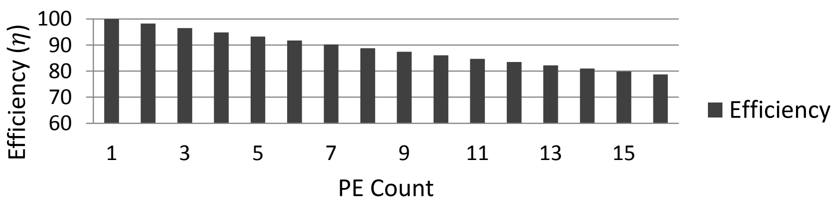


Figure 5.13.: Efficiency map for resource-aware Harris Corner detection on target HW [10]

For best results, the threshold value th can be set to zero, so that the algorithm will attempt to process all pixels in the image. It should be noted that T_{prn} and T_{cd} may vary depending on the actual implementation and the processor architecture. Hence these values were estimated by profiling the application on the target platform. Once sufficient resources are *invaded*, the workload-distribution stage can split the total workload into N *i*-lets (where $N = N_{pe}$) and move the individual workload data to the target tiles, through DMA operations over the *i*-NoC. This is followed by the *infect* operation which send the *i*-lets to perform the corner detection based on the pre-calculated threshold value. The *i*-lets are scheduled on the *invaded* PEs by OctoPOS. Each *i*-let computes the corner measure using (4.2) or (4.3), depending on which type of corner detection algorithm is being used. Only those pixels with corner response (CR) above the threshold value are processed by *i*-lets. The results computed by each *i*-let is stored in local TLM and copied back to the source tile's TLM, once the entire workload is processed. Upon completion, the application can release the allocated PEs (*retreat*) and retain execution on a single PE or perform a new *invade* request to allocate PEs for the next frame.

Considering the current system load, OctoPOS makes a final decision on the number of PEs to be allocated to the application. The PE count may vary from zero (if the system is too heavily loaded and no further resources can be allocated at that point in time) to the total number of PEs requested (provided that a sufficient number of idle PEs exist in the system and the current power mode offers sufficient power budget to enable the selected PEs). This means that under numerous circumstances the application may end up with fewer PEs and has to adapt itself to the limited resources offered by the runtime system. This is achieved by increasing the threshold value *th* until the condition in (5.3) is satisfied.

$$P_{pix}(th) \leq \frac{N_{pe} \cdot T_{exe} \cdot \eta(N_{pe}) - (n \cdot T_{prn})}{T_{cd}} \tag{5.3}$$

Figure 5.14 shows the effect of increasing the threshold on a sample image (top left corner). The pixels pruned away are represented using black color and the processed pixels in white. As the pruning threshold is increased, more pixels are pruned away and hence more and more regions in the resultant image turn black. This example also shows that the pruning technique eliminates regions without corners (e.g., wall regions) while the regions with high gradient values are retained for further processing.

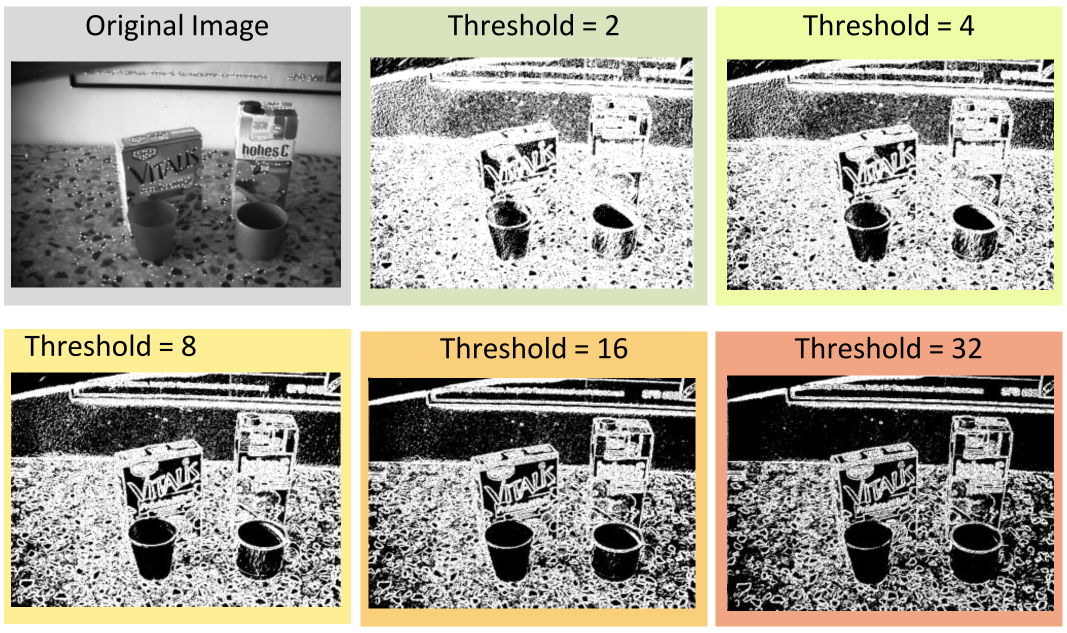


Figure 5.14.: Image samples demonstrating the pruning effect [10]

The modified flow diagram is depicted in Figure 5.15, where a new stage (adaptive threshold calculation) is added after the resource-allocation stage. This stage helps the algorithm to adapt to the available resource so that the probability of frame drops can be reduced even when sufficient resources are not available.

The results from the resource-aware model are presented in Section 5.1.4, where the conventional algorithm is compared against the resource-aware model.

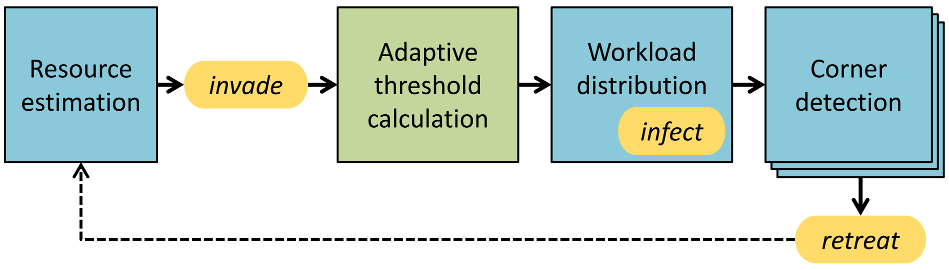


Figure 5.15.: Flow diagram for threshold estimation based on available resources [10]

5.1.4. Results

This section presents the results obtained using the resource-aware corner-detection algorithm and also provides a comparison with the conventional model. The conventional and the resource-aware models were evaluated on the MPSoC hardware described in Section 3.2. To analyze the problems arising from resource sharing, we examined the behavior of the conventional Harris [77] and Shi-Tomasi [84] corner detectors on a MPSoC with a total of 16 PEs. A sequence of 200 VGA frames (640 x 480 grayscale) were processed by the application. To evaluate the impact of other applications running concurrently on the MPSoC, we used applications like audio processing or motor control. These applications create dynamically changing load on the processor based on what the robot is doing at the current point in time. For instance, the speech-recognition application is activated whenever the user speaks to the robot and the motor control is activated when the robot has to move or grasp a recognized object. A conventional operating-system (OS) scheduler schedules the threads of each application considering the overall system load. As a result, the resources available to each application may vary over time.

A fixed resource-allocation scheme was used, as shown in Figure 5.4, to ensure a fair comparison. The corner-detection algorithm is programmed to run at 10 frames per second (fps) and needs a minimum of 9 PEs. However, at various points in time, fewer PEs were offered by the runtime system, leading to frame drops as the algorithm is running on real-time video input. Frame drops reduce the overall accuracy of the corner-detection algorithm. In order to evaluate the impact of frame drops on the accuracy of detected corners, we use the metrics named *precision* and *recall* as proposed in [96]. The value of recall measures the number of correct matches out of the total number of possible matches, and the value of precision measures the number of correct matches out of all matches returned by the algorithm. Results were captured for both Harris and Shi-Tomasi detectors and the variation in accuracy values due to sharing of available resources is shown in Figure 5.16, with the accuracy values computed using (5.4):

$$Ac(n) = \frac{\sum_{i=1}^{n} Pr(n) + \sum_{i=1}^{n} Re(n)}{2n} \tag{5.4}$$

Ac represents the average accuracy for n frames, Pr represents the precision and Re represents the recall values. The value of accuracy at any point is the average of the precision and recall until that point in time.

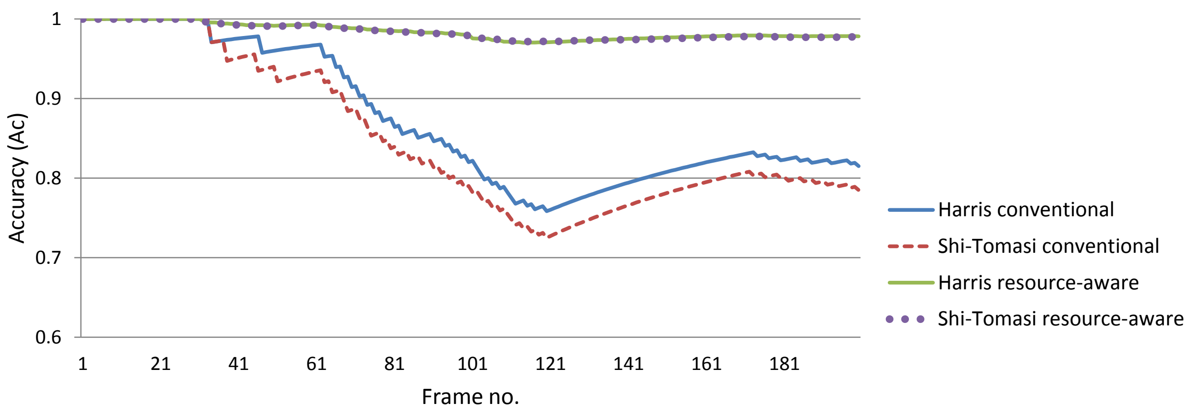


Figure 5.16.: Comparison between the conventional and resource-aware models [10]

For the initial few frames, the accuracy stays at the maximum value of 1. However, for the conventional mode, lack of sufficient resources results in an accuracy drop as shown in Figure 5.16. At some instances, the overall accuracy reaches a very low value of 0.73 for the Shi-Tomasi and 0.76 for the Harris detector. The lack of sufficient resources also resulted in an overshoot in execution time, where, based on the load conditions, the execution time increased by a factor of up to 4.3, as the processing of a single frame now takes 430 milliseconds (ms) instead of 100 ms. The number of frames dropped during the evaluation period was as high as 22 percent. The overall accuracy dropped to 79 and 81 percent for Shi-Tomasi and Harris detectors, respectively. This is a significant loss in accuracy for many real-world applications of corner detection, like object recognition. Object recognition heavily relies on the results of corner detection; the robot may lose track of the object if too many consecutive frames are dropped.

This evaluation also showed that the adaptations made by the resource-aware model helped to avoid frame drops. No frames were dropped during the evaluation and the accuracy values improved significantly over the conventional approach. The resource-aware and the conventional algorithms resulted in the same accuracy value when sufficient resources were available. In the case of resource shortage, however, the resource-aware model adapted its workload by increasing the pruning threshold. A slight drop in accuracy caused by the pruning of pixels is visible in Figure 5.16. However, the overall accuracy has improved significantly over the conventional approach. Moreover, this helped to avoid an overshoot in execution time and thus to eliminate frame drops, so that the results are consistently available within the predefined deadline.

An overall comparison between the two models is available in Table 5.1, where *throughput* is represented by the percentage of frames processed and *accuracy* is measured using Ac presented in Equation (5.4).

In brief, the resource-aware corner detector can operate very well under dynamically changing conditions by adapting the workload and thus avoiding frame drops and improving the detection accuracy [10]. When compared to the conventional detectors, the overall accuracy values for the Shi-Tomasi detector remain below the Harris detector as the Shi-Tomasi algorithm involves additional computations (additional square-root and division operation in Equation (4.2)) while the resource-allocation pattern remained unchanged. However, the

Algorithm	Throughput	Accuracy
Harris conventional	81%	0.81
Harris resource-aware	**100%**	**0.98**
Shi-Tomasi conventional	78%	0.78
Shi-Tomasi resource-aware	**100%**	**0.98**

Table 5.1.: Comparison between conventional and resource-aware corner detectors [10]

resource-aware versions of both algorithms resulted in an almost equal accuracy as the pruning was more effective for the Shi-Tomasi detector due to the higher computation demand after the pruning stage.

Table 5.2 shows the profiling results for each stage within the resource-aware Harris corner detector. These experiments involved an *invade* request for 8 PEs spread across the 4 tiles on the evaluation platform.

Stage	Time (ms)
Resource estimation	12
Resource allocation (*invade*)	0.26
Adaptive threshold calculation	0.8
Workload distribution	2.4
Corner detection	97
Resources de-allocation (*retreat*)	0.01
Total	**112.5**

Table 5.2.: Time taken for various stages in the resource-aware corner detection [10]

As the execution time for corner detection varies based on the actual workload, it is difficult to provide a precise number for this stage. The value provided in Table 5.2 is for zero threshold, meaning that all pixels were processed. It can be seen that the time spent for resource allocation and release is below 0.5 percent, while the additional logic for resource awareness resulted in approximately 12 percent overhead. However, results computed within this stage can be reused in the corner-detection stage, resulting in a reduced overhead for the corner detection. The time for *infect* is only a small fraction of the workload-distribution stage, as most of the time is consumed by loading and transferring the large image blocks across tiles.

Regarding scalability towards larger architectures, two effects come into play. First, with an increasing number of tiles, communication latency over the *i*-NoC rises, as packets have to traverse more hops to reach their destination. For our prototype's $N \times N$ meshed grid network, the maximum number of traversed hops equals $\sqrt{N}$, which means that maximum latency scales better than linearly with an increasing number of tiles. Second, when targeting larger platforms with more tiles and PEs, more messages need to be send both for acquiring and releasing resources via *invade* and *retreat*, and for dispatching *i*-lets via *infect* to do the actual computation. For *invade* and *retreat*, it is necessary to send more messages to address more tiles. However, once the request messages are out, the operations complete in parallel on all affected tiles. The same holds for dispatching *i*-lets via *infect*. Therefore, on the one hand with increasing architecture sizes we incur an additional overhead for initiating *invade*, *infect* and *retreat* operations, but on the other hand we leverage the increased parallel processing power to complete all operations in parallel.

5.2. Resource-Aware KD-Tree Search

In computer science, a *k-dimensional tree* or *kd-tree* is a space-partitioning data structure for organizing points in a k-dimensional space. In other words, kd-trees are a special case of binary space-partitioning trees, where every node is a k-dimensional point. Kd-trees are useful data structures for several applications such as range searches and nearest-neighbor searches (*NN-searches*). The NN-search algorithm aims to find the point in the tree that is nearest to a given input point. An efficient search can be implemented by taking advantage of the kd-tree properties leading to a quick search-space reduction. Further speedups can be achieved by using an approximation algorithm. For example, an approximate NN-search can be achieved by simply setting an upper bound on the number of points to examine in the tree, or by interrupting the search process based on a real-time clock (which may be more appropriate in HW implementations). Approximate nearest-neighbor search is useful in real-time applications such as robotics due to the significant speedup achieved by not searching for the best point exhaustively.

The NN-search problem arises in numerous fields of application including computer vision, pattern recognition, statistical classification, computational geometry, data compression, DNA sequencing, cluster analysis, etc. In the context of 3D vision, NN-search is frequently used in 3D point-cloud registration. A comparison between these techniques is available in [97]. However, the registration of large data sets is computationally expensive. As an example, the humanoid robot ARMAR-III [1] is capable of recognizing and tracking textured objects. The recognition algorithm uses a combination of Harris Interest Points and SIFT feature descriptors as described in [11]. Features extracted from the scene are matched against a pre-computed object database using a heuristic NN-search. This enables the processing of large numbers of features on every object to be recognized and tracked. A high recognition frame rate is achieved by using the best-bin-first search algorithm [98].

The compute-intensive nature and the high degree of inherent parallelism in the NN-search algorithm makes it suitable for implementation on an MPSoC. However, the available resources on an MPSoC (processing elements (PE), memories, interconnects, etc.) have to be shared among various applications running concurrently, which leads to unpredictable execution time or frame drops during NN-search. To address these challenges, a novel resource-aware NN-search algorithm for kd-trees has been developed. Based on the new algorithm, this chapter describes how to distribute the huge workload during the NN-search on the massively parallel PEs for best performance, and how to generate results on time (avoiding frame drops) even under varying load conditions.

5.2.1. Conventional Approach for Accelerating NN-Search on KD-Trees

NN-search on kd-trees is a compute-intensive task with a high degree of inherent parallelism, and can be accelerated using multi-core CPUs or GPUs. Studies conducted in the past suggested performing NN-search on the GPU using the basic brute-force technique [99]. The brute-force search is a simple search technique that compares one element with every other element in the database. GPUs have impressive brute-force search performance. However, GPU architecture makes efficient data-structure design quite difficult. In particular, GPUs are vector-style processors with limited branching ability. Hence, conditional computation typically under-utilizes these devices seriously [100]. Brute-force search on the GPU is still much faster than it is on the CPU, but not much faster than a kd-tree-based NN-search on a CPU [101]. Because of the ubiquity of the NN-search problem, a huge variety of data structures and algorithms have been developed to accelerate this process. Cayton et al. [101] introduced a simple data structure for NN-search on the GPU, with search and build algorithms that are efficient on parallel systems. However, the authors state that a significant effort was required to develop the GPU software in comparison to the well-known and simple NN-search on a CPU.

Novel architectures like Intel's Single-Chip Cloud Computer [50] or Tilera's 64-core processor [102] offers massive computing power and are suitable for implementing highly parallel algorithms like NN-search. Such architectures can overcome the limitations imposed by multi-core platforms with a limited number of cores, and the high degree of parallelism within the HW can lead to a significant acceleration of the conventional and simple NN-search algorithm for kd-trees [100].

5.2.2. Challenges with conventional kd-tree search on shared MPSoC

For applications like real-time object recognition and tracking, the NN-search algorithm has to complete the search process within a predefined time. For fast-moving objects the search interval has to be reduced so that the object can be tracked accurately. This can be achieved by using more PEs on the MPSoC. The actual duration of the search depends on the size of the data set to be processed and the size of data set depends on the number of objects present in the scene, the nature of the background, lighting conditions, etc. Figure 5.17 shows the variation in execution time when the NN-search is performed on a kd-tree of SIFT features (used by the ARMAR robot to recognize objects).

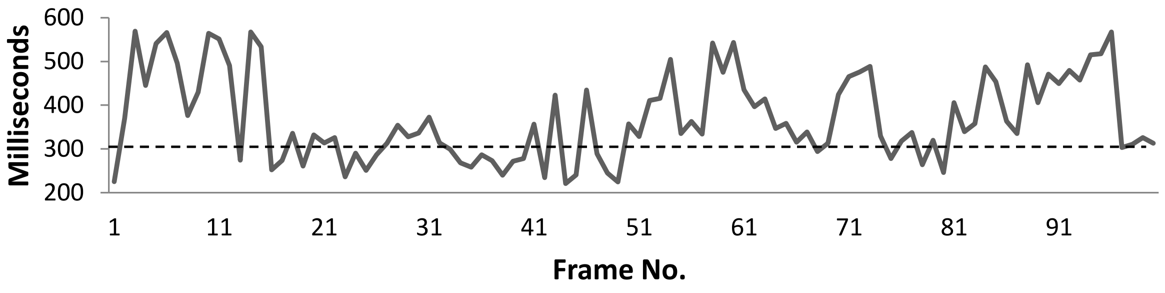


Figure 5.17.: Execution time for NN-search (static allocation) [13]

In this case the application is statically scheduled on 16 concurrent PEs to ensure that the available computing power does not vary over time. A sequence of hundred different scenes were processed by the algorithm and the size of the data set for kd-tree search varies in every scene. It can be seen that the execution time varies between 200 and 600 milliseconds, based on the size of the input data set. However, this evaluation is not complete as the static resource allocation is not a recommended approach, and results in poor resource usage. This is evident from Figure 5.17 where the resources were allocated such that the application can process one frame every 300 milliseconds (represented by the dotted line). However, the NN-search duration falls below or above the deadline based on the number of features to be processed. Points where the execution time falls below the deadline represent under-utilization of allocated resources, while those above the line indicate a lack of sufficient resources.

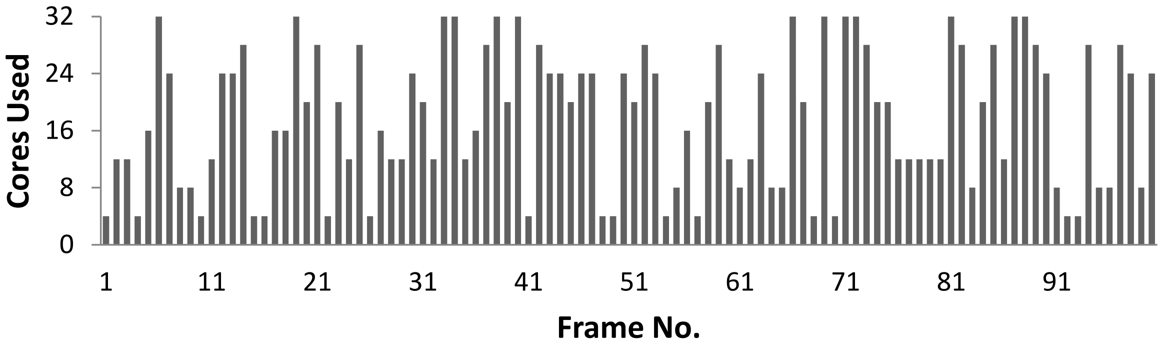


Figure 5.18.: Dynamic resource usage during NN-search [13]

The execution time can be equalized by adding more cores to the application when higher computing power is required and vice versa. Furthermore, the impact of other applications running concurrently on the MPSoC system (audio processing, robot control, etc.) has to be considered. These applications create dynamically changing load on the processor based on what the robot is doing at that point in time. For instance, the motion planning gets scheduled when the robot is moving or when the robot has to grasp an object. The conventional OS scheduler partition the available resources to different application considering the overall system load. As a result, the PEs allocated to each application may vary from time to time, leading to an overshoot in the execution time. Such a situation is depicted in Figure 5.18, where the y-axis represents the number of PEs allocated to the NN-search application for each frame, for a total of 100 frames.

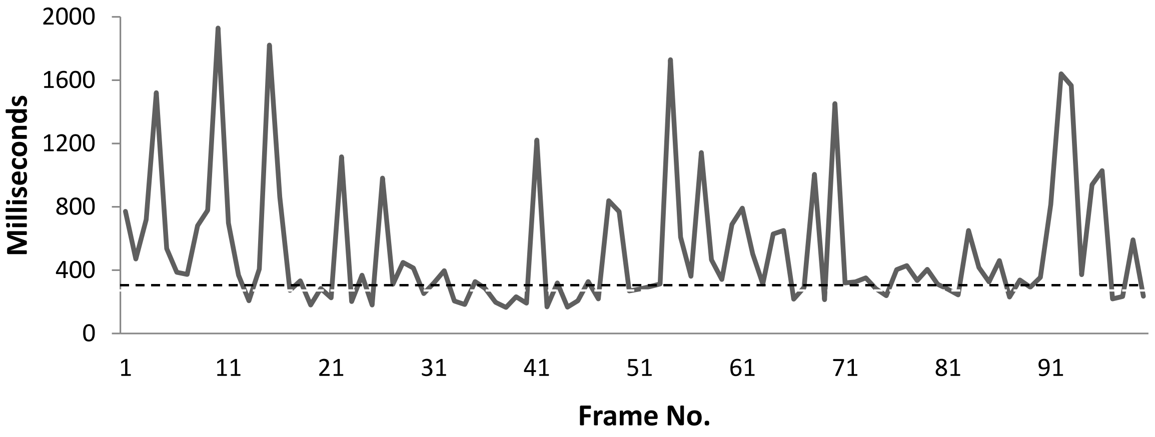


Figure 5.19.: Execution time for NN-search (dynamic allocation) [13]

The resulting execution time is depicted in Figure 5.19 and the results indicate very high jitter in the execution time. This evaluation reveals the highly unpredictable search duration for NN-search on today's MPSoC. Prolonged search durations would lead to frame drops and tracking errors. The number of frames dropped during this evaluation was as high as **44%**. The robot may even lose track of the object if too many consecutive frames are dropped.

In order to overcome these challenges, a novel algorithm for the NN-search was developed, where the application program can request for resources on an MPSoC and adapt the current workload based on the available resources. The upcoming sections demonstrate how the resources are claimed and how the search interval is constrained to guarantee better response time, compared to conventional MPSoC systems. The results indicate a significant improvement in overall performance of the recognition process when the NN-search is performed on the new resource-aware platform.

5.2.3. Nearest-Neighbor Search on KD-Trees

The object-recognition process used on the ARMAR robot consists of two steps. In the first step, the robot is trained to recognize the object. A training data set consisting of SIFT features is created for every object to be recognized. To speed up the nearest-neighbor computation, a kd-tree is used to partition the search space; one kd-tree is built for each object. The second step in the recognition process has real-time requirements as it helps the robot to interact with its surroundings (by recognizing and localizing various objects) in a continuous fashion. In this step, a set of SIFT features, extracted from the real-time image, is compared (using NN-search) with the pre-loaded data set (kd-tree).

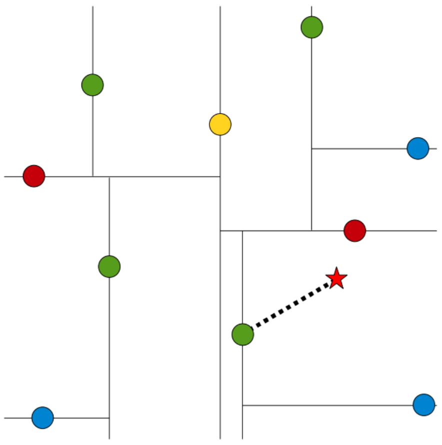


Figure 5.20.: Kd-tree search algorithm

The nearest-neighbor look-up works as follows. In Figure 5.20 assume that the star indicates a SIFT feature (called the test point) from the real-time input image, which has to be compared with the SIFT features from the training image (arranged in the form of a kd-tree). Given a kd-tree and a point in space (the test point), the aim of the nearest neighbor search algorithm is to find which point in the kd-tree is closest to the test point. The point in the data set closest to the test point is called its nearest neighbor. For example, suppose for the test point (indicated by the star) the nearest neighbor could be the point connected to the star by the dashed line. In general, if there is a point in this data set that is closer to the test point that the current guess, it must lie in the circle centered at the test point that passes through the current guess. This circle is shown in Figure 5.21

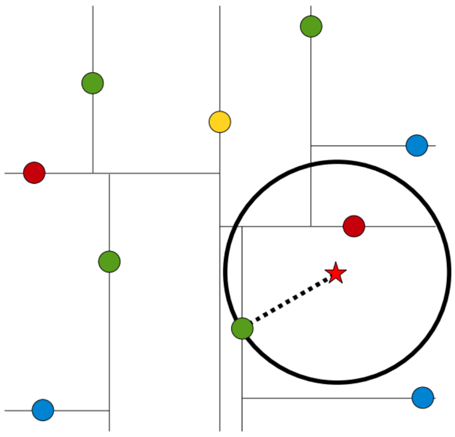


Figure 5.21.: Match candidates for kd-tree search

Although in this example this region is a circle, in three dimensions it would be a sphere, and in general it is called a candidate hypersphere. The circle helps to prune which parts of the tree might hold the true nearest neighbor.

The computation of the nearest neighbor for the purpose of feature matching is the most time-consuming part of the complete recognition and localization algorithm. This algorithm performs a heuristic search and only visits a fixed number of leaves resulting in an actual nearest neighbor, or a data-point close to it. Figure 5.22 shows the result of the recognition process, where the real-time image captured by the robot head is on the left and the training image on the right.

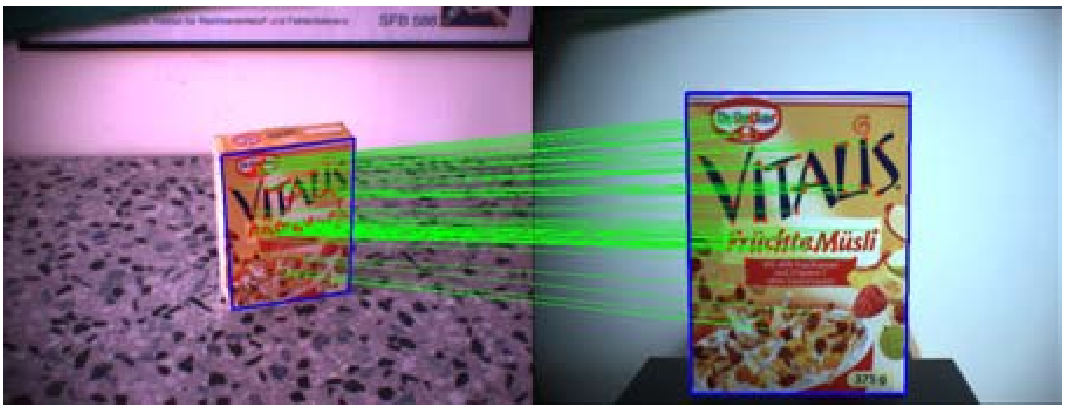


Figure 5.22.: Feature correspondences after the NN-search [11]

For the NN-search algorithm, the number of kd-tree leaves visited during the search process determines the overall quality of the search process. Visiting more leaf nodes during the search leads to a higher execution time. The search duration per SIFT feature can be calculated from Figure 5.23. The values were captured by running the NN-search application on a single PE using a library of input images covering various situations encountered by the robot.

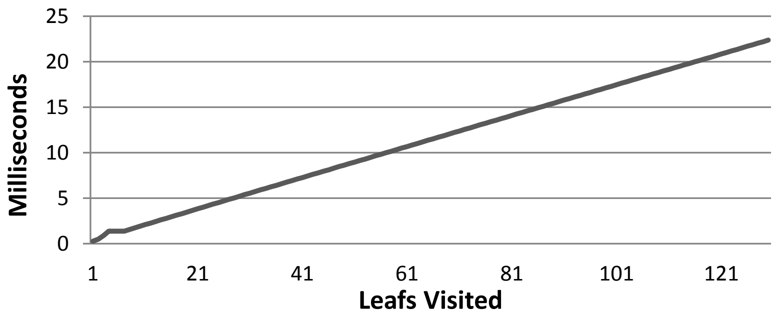


Figure 5.23.: Variation in execution time vs. leaf nodes visited for NN-search [13]

From this graph it is clear that the search interval varies linearly with the number of leaf nodes visited during the search. Moreover, the relation between quality (i.e. the number of features recognized) and leaf nodes is shown in Figure 5.24. The quality of detection falls rapidly when the number of leaf nodes is reduced below 20 and increases linearly in the range between 20 and 120.

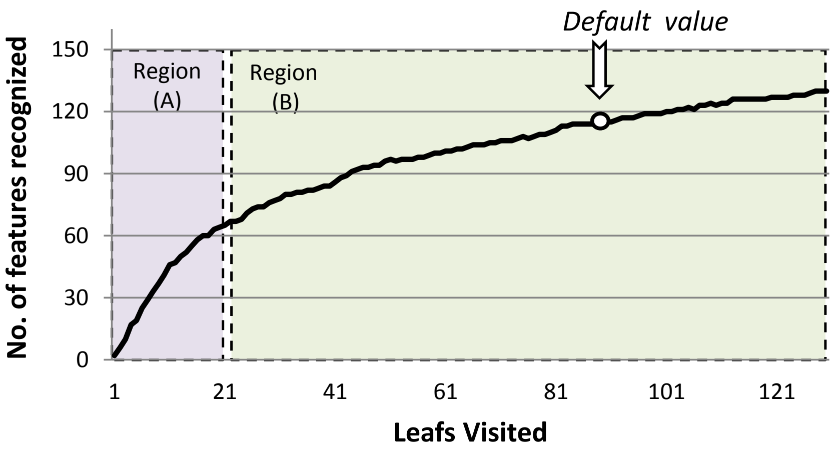


Figure 5.24.: Search quality vs. leaf nodes visited for NN-search [13]

At a further higher leaf count, the quality does not improve significantly as the best possible matches were already found. In the conventional algorithm used on CPUs, the number of leaf nodes visited is set statically such that the search process delivers results with sufficient quality for the specific application scenario. Using the results from this evaluation, the overall search duration can be predicted based on the object to be recognized, the number of features to be processed and the number of PEs available for NN-search. The first two parameters are decided by the application scenario while the PE count is decided by the runtime system based on the current load situation. As shown in Figure 5.18, the resources allocated to the application may vary from time to time, leading to highly unpredictable search durations and frame drops. Two different techniques can be applied to the conventional algorithm to constrain the execution time, as described below.

5.2.3.1. Threshold-Based Search

In order to avoid frame drops and to improve the tracking accuracy, the conventional NN-search can be modified to process the SIFT features based on their quality. The number of leaf nodes visited is fixed (to the default value) and once the deadline is hit, the algorithm can drop the remaining low-quality features and move on to the next frame. This technique is relatively simple, easy to implement and works on any single-/many-core platform. The algorithm would perform well in scenarios where the scene contains only the object to be recognized and all the detected features belong to the same object. The results deteriorate when there are more objects in the frame and also in scenes with cluttered background; this is because some of the high-quality SIFT features may belong to other objects or to the background. Therefore, the features dropped by the algorithm may belong to the target object, leaving it undetected.

Such a scenario is show in Figure 5.25 where the image on top shows the results of SIFT feature matching based on the conventional search algorithm. In this case, sufficient resources were available and the algorithm could process all features within the specified time frame. The processed features are marked in green and the features which were successfully matched are circled in blue. The image at the bottom (Figure 5.25) shows the results from the threshold-based search operation. Sufficient amount of resources (PEs) were not available in this case and the algorithm dropped significant number of features (marked in red) when the deadline was hit. It could be seen that the total number of matches (circled in blue) has decreased when the threshold-based search is used without sufficient resources in a cluttered scene.

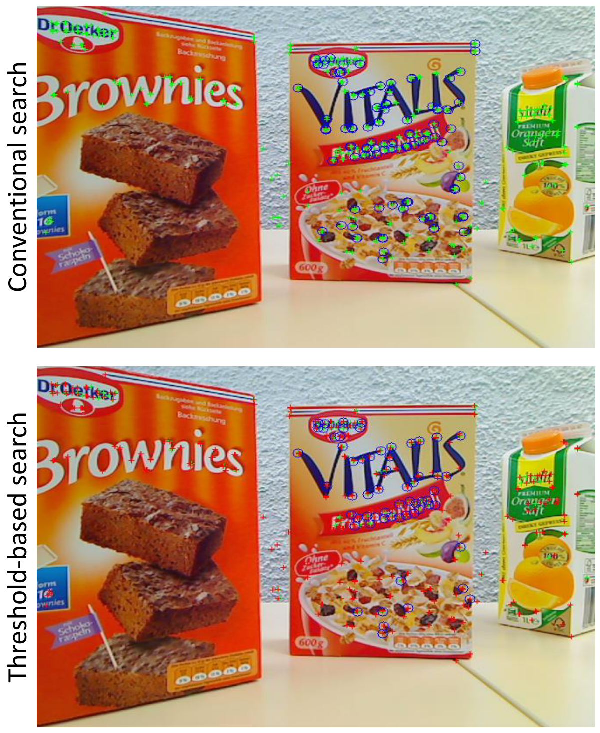


Figure 5.25.: Conventional vs. threshold-based kd-tree search

5.2.3.2. Iterative Search

An alternative approach to overcome the problems in the threshold-based search is to modify the conventional NN-search to proceed in an iterative manner. This means that the search process starts with the first feature and performs a search until the first leaf node is reached. The results are saved and the algorithm moves on to the next feature and repeats the same process again. Thus, every feature in the data set is processed once (to the first leaf node) and then the algorithm returns to first feature again to continue the search from the first leaf node to the second. This process continues until a default number of leaf nodes is visited or until the next frame is available, whichever occurs first. In this manner, when the deadline is hit, the algorithm will have performed an equal search for all the features in the data set and none of the features will be dropped. However, the overall quality of the NN-search will be significantly affected, if the search process stops half-way without reaching the default leaf count, as shown in Figure 5.24. Detailed evaluation were performed to compare the results from the threshold-based and iterative NN-search and the findings are provided in Section 5.2.5.

5.2.4. Resource-Aware Nearest-Neighbor Search Algorithm

Sections 5.2.2 and 5.2.3 described the conventional algorithm, its limitations, and also presented two modified algorithms for object tracking on conventional MPSoC platforms. This section describes the novel approach towards NN-search using the idea of resource-aware programming based on Invasive Computing on the MPSoC architecture InvasIC. The main idea and novelty of the new algorithm is that the workload is calculated and distributed taking into account the available resources (PEs) on the MPSoC.

5.2.4.1. Adaptive workload distribution

The first step in the resource-aware nearest neighbor search algorithm is to allocate sufficient resources to perform a parallel search. The resource-allocation is based on various aspects as listed below:

- The number of SIFT features to be processed, which in turn depends on:
  - The nature and number of objects present in the frame.
  - The nature of the background, foreground, etc.
- The size of the kd-tree.
- The available search interval.

The number of SIFT features varies from frame to frame based on:

The size of the kd-tree is decided by the texture pattern on the object to be recognized and tracked. The search interval or the frame rate is decided by the context where the NN-search is employed (for example, if the robot wants to track a fast-moving object, the frame rate has to be increased or the execution time has to be reduced).

Equation (5.5) represents this relation and can be used to compute the number of PEs (N_{pe}) required to perform the NN-search on any frame within the specified interval T_{search}. N_{fp} is the number of SIFT features to be processed and T_{fp} is the search duration per SIFT feature, a function of the number of leaf nodes visited, as described in Figure 5.23. The initial resource estimate is based on the default leaf count (N_{leaf_best}), as described in Section 5.2.3.

$$N_{pe} \geq \frac{N_{fp} \times T_{fp}(N_{leaf\_best})}{T_{search}} \tag{5.5}$$

Note that the function $T_{fp}(N_{leaf\_best})$ is different for every object to be recognized and tracked by the robot, as this is dependent on the number of features forming the kd-tree, the shape of the tree, etc.

5.2.4.2. Efficiency Graphs

Equation (5.5) assumes that the search interval decreases linearly with increasing PEs. Such an assumption does not hold, considering the limited parallelism within the application program. For example, the NN-search algorithm is highly parallel during the search process. However, the overall execution time also includes the time to load the kd-tree to on-chip memory (TLM) via *i*-NoC, combine the results from individual *i*-lets, filter the best matches, etc. Furthermore, every additional *i*-let created by the NN-search algorithm also creates an additional load on the external memory and shared communication interfaces like AHB Bus, *i*-NoC, TLM, etc., limiting the scalability. Our analysis of NN-search on the proposed MPSoC hardware resulted in an efficiency graph as shown in Figure 5.26. From the graph it can be seen that when the number of *i*-lets is increased from 1 to 2, the execution time does not improve by 2x, instead by 2×0.98 (98%), i.e. 1.96x. Using this graph, the efficiency factor for various levels of parallelism can be computed. The values shown here are applicable only for the NN-search implementation used in this work and may vary based on how the original algorithm is implemented.

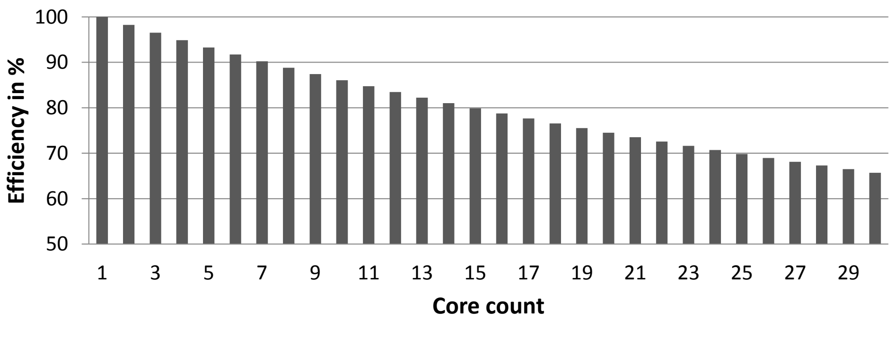


Figure 5.26.: Efficiency graph for NN-Search on MPSoC hardware [13]

In order to increase the accuracy of the resource-estimation model, an application-specific efficiency factor or scalability information can be added to (5.5). The enhanced model is represented by (5.6), where $\eta(N_{pe})$ represents the algorithm's efficiency as a function of degree-of-parallelism or available resources (N_{pe}).

$$N_{pe} \geq \frac{N_{fp} \times T_{fp}(N_{leaf\_best})}{T_{search} \times \eta(N_{pe})} \tag{5.6}$$

Using the new model, the application raises a request to allocate PEs (N_{pe}), which is then processed by the OctoPOS. Considering the current system load, the OctoPOS makes a final decision on the number of PEs to be allocated to the NN-search algorithm. The PE count may vary from zero (if the system is too heavily loaded and no further resources can be allocated at that point in time) to the total number of PEs requested (provided that there exists a sufficient number of idle PEs in the system and the current power mode offers sufficient power budget to enable the selected PEs). This means that under numerous circumstances the application may end up with fewer PEs and has to adapt itself to the limited resources offered by the runtime system.

5.2.4.3. Resource-Aware Workload Distribution

In constrained scenarios as explained above, the application has to re-balance the workload in order to complete the NN-search within the search interval specified by T_{search}. This is achieved by recalculating the number of leaf nodes (N_{leaf_adap}) to be visited during the NN-search such that the condition in (5.7) is satisfied.

$$T_{fp}(N_{leaf\_adap}) \leq \frac{N_{pe} \times T_{search} \times \eta(N_{pe})}{N_{fp}} \tag{5.7}$$

The algorithm can use the new leaf count for the entire search process on the current image. However, it can be seen from Figure 5.24 that the quality drops significantly if the number of leaf nodes calculated in (5.7) is too low (between 0 and 20, for the particular object used in our evaluation). This region is marked as region(A) in the figure. This issue can be resolved by preventing the leaf count from falling below the minimum leaf count (N_{leaf_min}) or the estimated leaf count should fall within the region(B) as shown in Figure 5.24. Under most circumstances the application can process all the features by adapting the leaf count. However, if there are a large number of features to be processed using too few PEs, the leaf count calculated may fall below the minimum limit. In this case, the NN-search will drop a few low-quality SIFT features to complete the search within the predefined search interval.

It should be noted that the resource-allocation process operates once for every frame. Upon completion, the application releases the resources and waits for the next frame to arrive. Therefore, the core algorithm for NN-search retains its simple structure in contrast to the complex iterative search algorithm described in Section 5.2.3.2. The flow diagram in Figure 5.27 describes the entire process of resource allocation and workload calculation for resource-aware NN-search.

In contrast to the threshold-based search described in Section 5.2.3.1, the resource-aware NN-search can process all features, in most circumstances and hence overcome the challenges raised by scenes with multiple objects or cluttered backgrounds. As the available resources are known in advance, the application can make an early decision on the num-

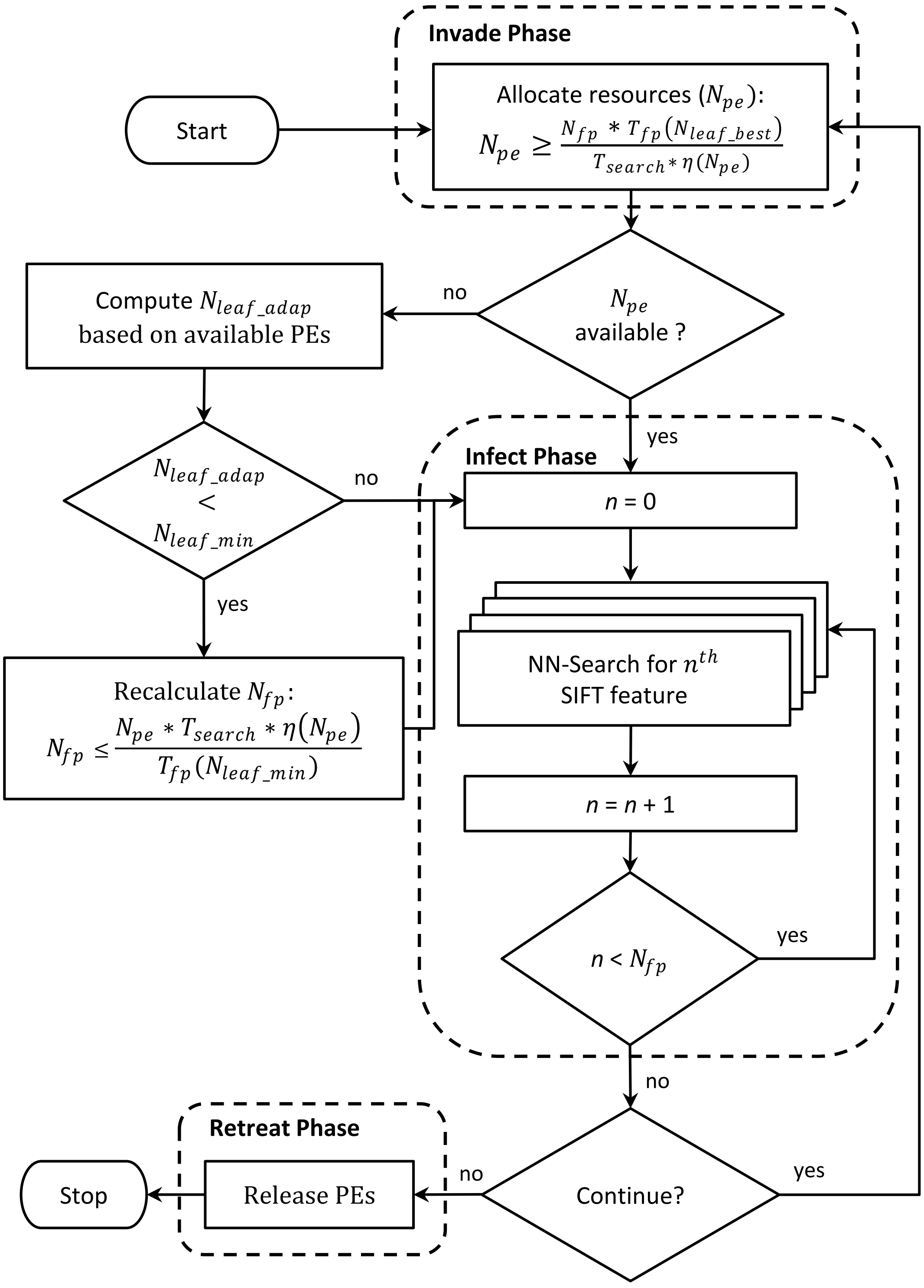

Invade Phase
Start
Allocate resources (N_{pe}):
$N_{pe} \geq \frac{N_{fp} * T_{fp}(N_{leaf\_best})}{T_{search} * \eta(N_{pe})}$
N_{pe} available ?
no
Compute N_{leaf_adap} based on available PEs
$N_{leaf\_adap} < N_{leaf\_min}$
no
yes
Recalculate N_{fp}:
$N_{fp} \leq \frac{N_{pe} * T_{search} * \eta(N_{pe})}{T_{fp}(N_{leaf\_min})}$
yes
Infect Phase
$n = 0$
NN-Search for n^{th} SIFT feature
$n = n + 1$
$n < N_{fp}$
yes
no
Retreat Phase
Stop
Release PEs
no
Continue?
yes

Figure 5.27.: Flow diagram for resource-aware NN-search [13]

ber of leaf nodes to be visited during NN-search and can avoid the additional complexity encountered by the iterative approach described in Section 5.2.3.2.

5.2.5. Evaluation & Results

This section describes the results obtained from the resource-aware NN-search along with a comparison with the conventional search techniques like threshold-based and iterative search, on a massively parallel MPSoC. A set of 100 different scenes was used for evaluation, where each frame contains the object to be recognized and localized along with few other objects and changing backgrounds. The position of the objects and their distance from the robot were varied from frame to frame to cover different possible scenarios. Evaluations were conducted on the FPGA-based hardware prototype of InvasIC. As a single FPGA cannot hold the large MPSoC design (with 32 LEON3 cores, external memory controllers, multiple debug and I/O interfaces, on-chip memories, NoCs, etc.) a multi-FPGA prototyping platform from Synopsys called CHIPit System [103] was used. This system consists of six Xilinx FPGAs (Virtex-5 XC5VLX330) with a total capacity of 12 million ASIC gates. The design operates at a frequency of 25 MHz. Although the operating frequency of the hardware prototype is relatively low compared to an ASIC implementation, it does not affect the evaluation process as all three versions of the algorithm were tested on the same FPGA platform and the results are compared in terms of the quality of the detection process with a fixed search interval.

All three flavors of the NN-search algorithm were tested using the same set of input images for a search interval of 300 milliseconds per frame (3.3 frames per second). Figure 5.28 shows a comparison between the resource-aware and the threshold-based NN-search, with the number of features recognized (quality of detection) on the y-axis and the frame number on the x-axis. In order to maintain equality in the evaluation process, the number of PEs allocated to the applications was equalized. The PE distribution varies from frame to frame as shown in Figure 5.18. The remaining resources were either idle or allocated to other audio/video or motion-control applications running on the robot.

It is clear from Figure 5.28 that the resource-aware NN-search algorithm outperforms the conventional algorithm using the same amount of resources. This is because the resource-aware model is capable of adapting the search algorithm based on the available resources compared to the conventional algorithm with fixed thresholds. However, the resource-aware algorithm results in the same number of matched features as the conventional algorithm in some frames. This is because there were a sufficient number of idle PEs and the runtime system allocated sufficient resources to meet the computing requirements of the conventional algorithm and hence the conventional algorithm did not drop any SIFT feature. On the contrary, when a frame contains large number of SIFT features and the processing system is heavily loaded by other applications, the conventional algorithm dropped too many SIFT features, thereby resulting in a low overall detection rate (matched features). This behavior is depicted in Figure 5.29 and Figure 5.30. Figure 5.29 contains the object to be recognized along with a tiny cup and plain background. Hence the features dropped were the low-quality features on the target and most of the high-quality features were still retained, resulting in a comparable detection rate with threshold-based and resource-aware search algorithms. However, in Figure 5.30, additional objects and complex background resulted

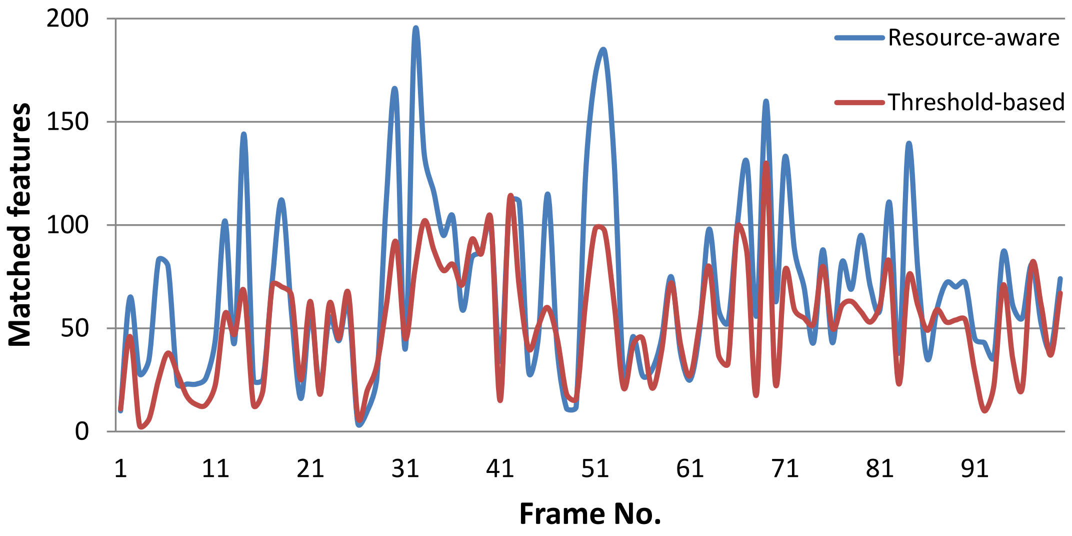


Figure 5.28.: Comparison between resource-aware and threshold-based NN-search [13]

in dropping high-quality features on the target leading to poor overall detection rate using threshold-based search. More details regarding these two test images can be found under Table 5.3. This result points to the ability of the resource-aware application to adapt itself to changing load conditions and generate better results in tightly constrained situations.

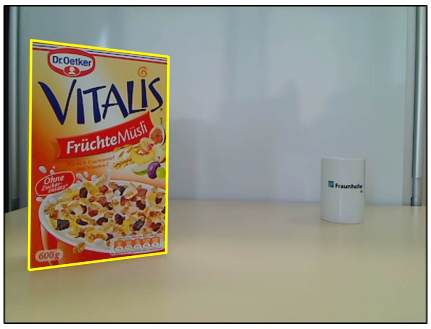


Figure 5.29.: Image(a): single object

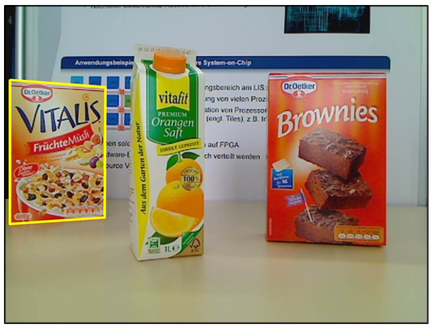


Figure 5.30.: Image(b): multi-objects

When compared to the threshold-based search, the iterative search algorithm offers improved results in some frames and equal or deteriorated results in other frames. A detailed analysis shows that under circumstances where a large number of features have to be processed using few PEs, the threshold-based model outperforms the iterative model. This is because the iterative model tries to perform a search without dropping any features in a scenario described above and the total number of leaf nodes visited during the search process may drop to very low values (see region(A) in Figure 5.24). Hence the overall quality can be too low as the NN-search may result in poorly matched candidates. The iterative model performs well compared to the threshold-based model in other situations as this model can avoid dropping features using the adaptive techniques described in Section 5.2.3.2.

A comparison between the resource-aware and the iterative NN-search is provided in Figure 5.31, where the resource-aware algorithm outperforms the iterative search in numerous

		Image(a)	Image(b)
PE count (alloc / required)	TR	12 / 23	14 / 31
	RA	12 / 23	14 / 31
FPs on object (PR / DR)	TR	177 / 320	**88 / 217**
	RA	320 / 320	217 / 217
FPs on others (PR / DR)	TR	21 / 42	143 / 284
	RA	42 / 42	284 / 284
Total FPs recognized	TR	108	**60**
	RA	111	**115**

Table 5.3.: Comparison between resource-aware and threshold-based NN-search (FP: Feature-point, PR:Processed, DR:Dropped, TR:Threshold-based search, RA:Resource-aware search) [13]

scenarios. The reason for this is the increased complexity within the iterative search resulting from the enhancements described in Section 5.2.3.2. The added logic to process the features in an iterative fashion results in higher complexity, leading to higher overall execution time. Moreover, the iterative model has to load every SIFT feature multiple times into the on-chip memory or data cache during the NN-search while the resource-aware model loads each feature once during the entire search process. This means, the iterative search algorithm creates a higher load on the external memory, TLM, AHB Bus, *i*-NoC, etc., which reduces its scalability and efficiency during execution. As a result, the iterative model cannot process as many features as the resource-aware model within a fixed interval, reducing the total matched features or the quality of the algorithm.

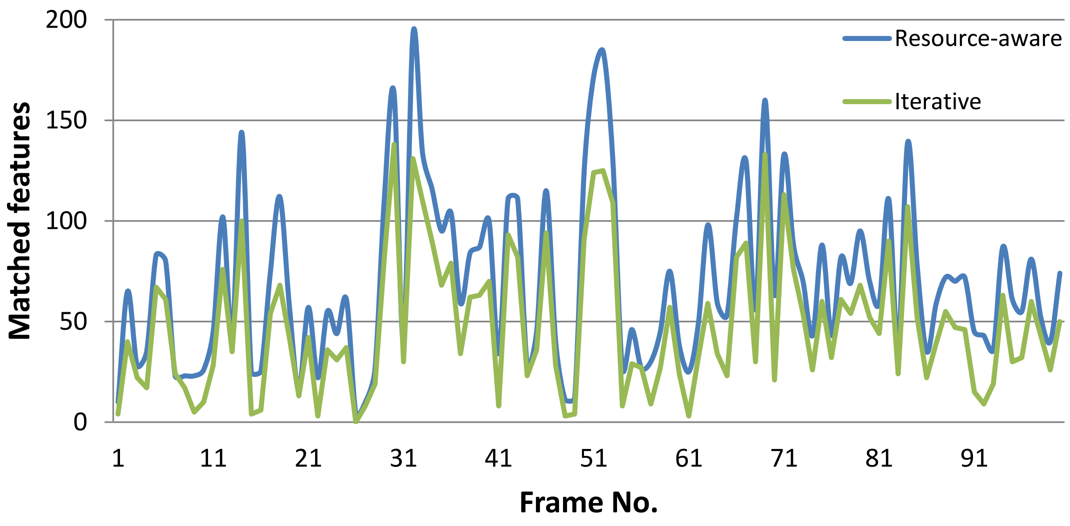


Figure 5.31.: Comparison between iterative and resource-aware NN-search [13]

5.2.6. Conclusion

With their immense computational power, MPSoCs provide the means to consolidate the whole computing system in mobile robotic applications onto a single chip instead of relying on separate physical computing units. With the case-study evaluating two widely used algorithms (Harris Corner detection and NN-search), this work demonstrated accuracy loss on a conventional MPSoC hardware due to resource sharing and thus, running once-separated applications on a unified hardware poses major problems.

As a way to overcome these challenges, this chapter presented novel resource-aware algorithms for Harris Corner detection and NN-search on kd-trees. This work also showed how to estimate the resources required for a specific task based on the scene (nature of the objects, number of objects present, type of background, the texture of the object to be recognized, etc.) Both the Harris and Shi-Tomasi corner detectors were enhanced to a resource-aware model by extending the already existing pruning techniques. Such enhancements enabled the corner-detection algorithms to recalculate its workload at runtime, based on the currently available resources on the MPSoC [10]. The ability of the applications to bargain for resources and adapt to available resources helped them to avoid frame drops and to complete the processing within the specified search interval. Our experiments show that incorporating resource-awareness into the conventional Harris Corner detection and NN-search algorithm can improve the quality of the object recognition process significantly. Evaluation showed that the adaptations made at runtime, made significant improvements in the performance of corner detection algorithms. The results showed up to 22 percent improvement in throughput and up to 20 percent improvement in accuracy. The results indicate that the use of resource-aware programming will allow migration of various applications mapped to multiple industrial CPU boards on conventional robot, to a single-chip MPSoC design, without suffering quality loss [13].

Detailed evaluations were conducted on an FPGA-based hardware prototype to ensure the validity of the results. Though the evaluations were conducted using the OctoPOS operating system and the InvasIC hardware, the benefits are expected to be visible on any resource-aware platform including ROS [59]. The newly proposed algorithms are simple and retains all the characteristics of the conventional algorithms. The resource allocation and release happens once per frame and the additional overhead in execution time is negligible when compared to the time taken by Harris Corner detection or NN-search, to process thousands of pixels or features in every frame. Profiling results based on an FPGA prototype indicate that the overhead from resource allocation and release can be kept very low by employing a specialized operating system and suitable hardware support.

6. Accelerating Image Processing using Specialized Hardware

Robot-specific tasks need to be executed on the hardware built into the robot. Timely execution of various tasks are guaranteed by providing the robots with sufficient underlying hardware (processing units) necessary to cope with worst case scenarios. Figure 6.1 show the architecture of InvasIC, an MPSoC if used, can handle the entire workload previous handled by several independent processing units on the ARMAR robot. The specialized PE types (*i*-Core, TCPA, etc.) can be utilized by some of the computer vision applications to achieve better performance when compared to the homogeneous counterparts.

This chapter describes how image processing algorithms can be accelerated using the heterogeneous processor InvasIC, the challenges involved in mapping application programs on to a heterogeneous processor, advantages and limitation of the proposed hardware architecture, etc. The case-study involves mapping of three different applications on the InvasIC. Section 6.2 describes Harris Corner detection implementation on TCPA and LEON3, Section 6.3 describes the implementation of SIFT feature matching on *i*-Coreand LEON3 followed by an optical flow implementation on the TCPA in Section 6.4. Each section describes the application mapping, data flow and performance benefits obtained using the heterogeneous design in comparison to the homogeneous model.

6.1. InvasIC - Programmer's View

As shown in Figure 6.1, the various PEs within the InvasIC are organized into tiles. PEs within the tile are interconnected using an AMBA (Advanced Microcontroller Bus Architecture) AHB (Advanced High-performance Bus). In addition to the PEs, each tile also contains a high speed on-chip memory (TLM), L2 cache and a network-adapter (NA). The applications may use the TLM to store frequently accessed data and it offers quick (tile-local) access to any data stored inside the TLM. The NA offers necessary features to copy data from the local TLM to the TLM of other tiles or to the external DDR-II memory (inside memory tile). The L1 cache maintains cache coherence (within the tile) by using the data snooping feature of the AHB Bus. Any data read/written from/to external memory or remote TLMs are cached by the L2 cache. The L2 cache is not coherent, hence the programmer has to take special care to ensure data consistency across tiles. In addition to the TLM, the InvasIC architecture also offers local scratchpad memories for each LEON3 core. The scratchpad memory can be accessed only by the core to which it is connected.

An application program running on the InvasIC has to make use of the distributed memory layout in order to achieve the best performance. This includes copying the frequently accessed data and data-structures from the external DDR-II memory to the TLM or scratchpad memories. If needed, the application programs can also make copies of the data within one TLM to other TLMs using the fast Direct-Memory-Access (DMA) feature offered by the *i*-NoC.

In addition to the homogeneous LEON3 cores, the InvasIC architecture also offers heterogeneous PEs as shown in Figure 6.1. On account of their immense computational power assembled in a compact design, the heterogeneous MPSoCs can bring in several benefits to the field of robotics. For example, the low level pixel processing algorithms can be accelerated using the TCPA [104] (within its massively parallel array of processing elements) or using special instructions offered by an *i*-Core. The entire design is prototyped on multiple FPGAs (Synopsys CHIPit System [103] consisting of six Xilinx Virtex-5 XC5VLX330 FPGAs) and runs at a frequency of 25MHz.

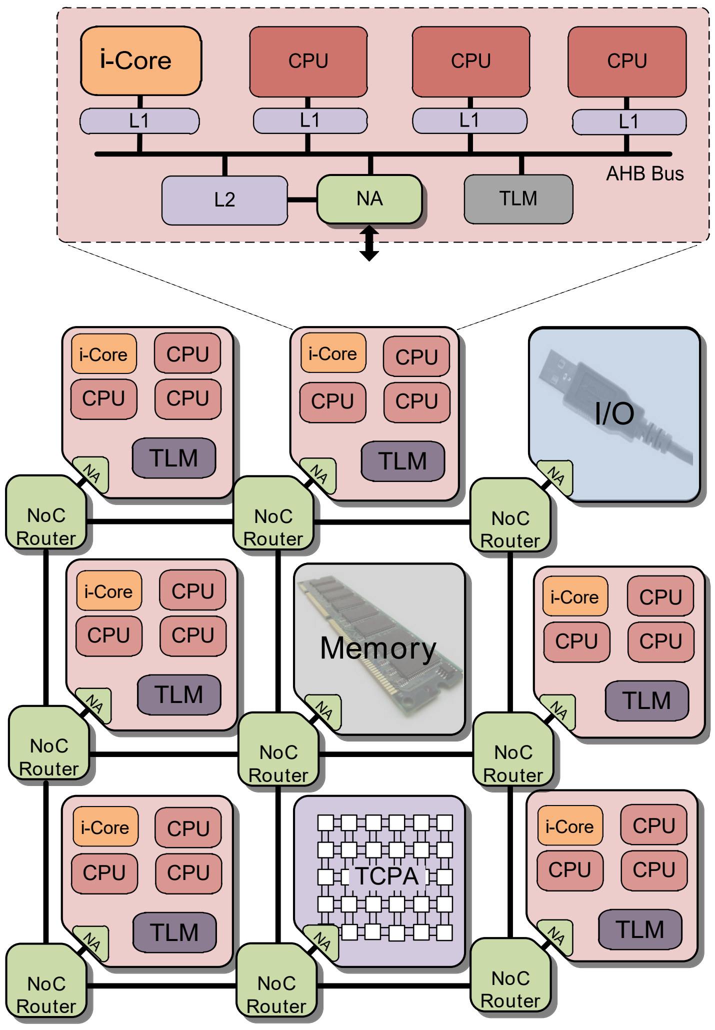


Figure 6.1.: Heterogeneous architecture of InvasIC with its tile internal structure

6.2. Harris Corner Detection on Tightly Coupled Processor Array

This Section describes the implementation of Harris Corner detection algorithm on a variant of the InvasIC, as shown in Figure 6.1. Each tile can contain I/O units, loosely coupled LEON3 CPUs or TCPA. TCPAs consists of numerous light weight PEs and they offer significant acceleration to image processing algorithms with regular loops for pixel processing. Loosely coupled processors on the other hand can be used extensively for post-processing of the initial features computed by TCPAs.

6.2.1. Conventional Approach

For corner detection, many studies have been proposed in literature with the purpose to reduce the computational complexity or to accelerate their execution. In [105], the authors reduced the complexity of corner detection by using a novel non-maximum suppression strategy. In [106], a fast corner detection algorithm is proposed to achieve real-time processing. In this work, the authors explore a highly parallel implementation on a Graphics Processing Unit (GPU). Reconfigurable architectures acting as coprocessor for real-time feature detection are described in [107] and [108]. Hosseini et al. [109] presents a highly parallel implementation of Harris Corner detection on SIMD (single instruction, multiple data) architectures. The architecture consists of 96 processing elements (PEs). In the initialization phase, the PEs are idle for a considerable time. This fact could be optimized utilizing different mechanisms to fetch the data from external memory. In addition, the approach utilizes a redundant external memory access that further decreases the performance of the system because of under-utilization of the available bus bandwidth.

Heterogeneous MPSoCs can offer significant performance boost when compared to their counterparts with off-chip GPUs. One main aspect that limits the performance of off-chip GPU is the bottleneck on the PCI Express (PCIe) used for data exchange between the CPU and GPU. With the emergence of heterogeneous computing architecture that fuse the functionality of CPU and GPU onto the same die, the problems with communication bandwidth is expected to be resolved. The AMD Fusion, an Accelerated Processing Unit (APU) or Intel's Knights ferry are examples of such heterogeneous architectures. To reduce the thread scheduling latencies, the AMD Fusion APUs have dedicated hardware scheduler for handling threads assigned to the GPU. Furthermore, automatic task scheduling based on OpenCL for APUs has been explored in [110]. This work presents a work-pool-based task queuing extension for OpenCL that allows programmers to easily harness the power of heterogeneous computing environments. This work also shows that there is a high potential for using the CPUs as compute devices, provided that there are sufficient CPU-based compute units available within the system.

However, differently from the aforementioned works, the InvasIC offers a highly reconfigurable architecture. Each PE within the TCPA has the flexibility to be programmed in real-time and efficiently utilize the resources (like TLM) inside the architecture, eliminating the need for redundant external memory access as well as the need to store multiple copy of the data between different partitions.

6.2.2. Implementation on InvasIC

The conventional Harris Corner detection has two main stages as described in Section 4.1.1. The algorithm used is based on the implementation from the Integrating Vision Toolkit (IVT) library [111]. The application program running on the CHIPit system is only responsible for the computation of corner points using the Harris Corner detection algorithm. The camera interface, pre-processing (color to gray-scale conversion), results visualization, etc., are handled by a host-PC connected to the CHIPit system. The setup used for evaluations is shown in Figure 6.2 and the various stages in the execution of the Harris Corner detection is shown in Figure 6.3.

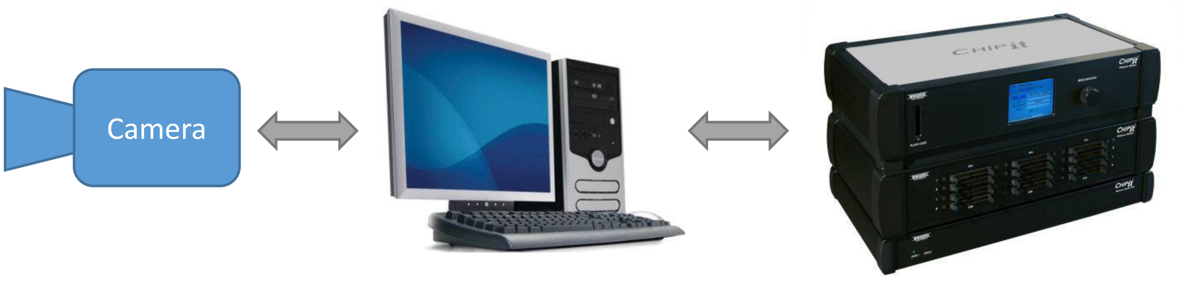


Figure 6.2.: Evaluation setup

The video frames from the camera are stored in the host-PC until the application program running on the CHIPit (on a LEON3 core) makes a function call to received the data over the Ethernet interface connecting the CHIPit and the host-PC. The Harris Corner detection application initiate this process by allocating sufficient memory in the TLM (size of the allocated memory should match the size of the input image). If the memory allocation is successful, the application program opens a channel between the CHIPit and the host-PC using the *eth_open* function as shown below. Once the Ethernet connection is established, the resource-aware application waits for the input image by calling the *eth_receive* function, where the *imgAddr* specifies the location for storing the image and *i*-let indicates the code/-function to execute once the data transfer has completed.

- eth_open(HCD_CNTRL_CHNL, ETH_MODE_READ);
- eth_receive(Channel_ID, imgAddr, sizeOfImg, *i*-let);

The OctoPOS schedules the application once again upon receiving the input image and the next step is to transfer the input image to the TCPA tile for computation of *harris-map*. Any pre-processing, if needed, can be performed by the LEON3 core at this stage. *Harris-map* is the first stage in the Harris Corner detection algorithm and it involves computation of intensity gradients based on a sliding window. As the algorithm for *harris-map* is highly regular and compute intensive, its mapped to the highly parallel TCPA PEs on the heterogeneous MPSoC.

The computation of *harris-map* on the TCPA tile is initiated using some special function calls, as described below. At the application level, the execution starts with an *invade* request to allocate a sufficient number of PEs to perform the *harris-map* computation. The *invade* request is processed by the runtime system (OctoPOS), which checks for the availability

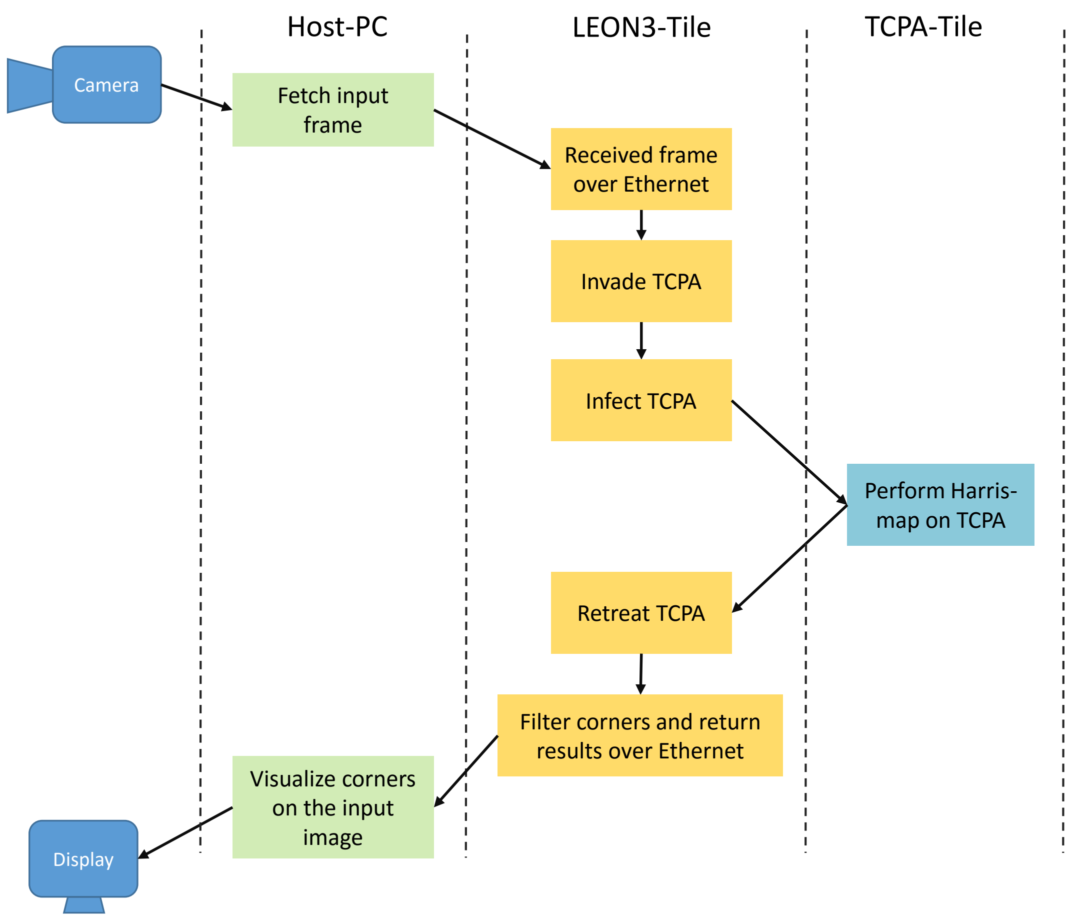

Host-PC
LEON3-Tile
TCPA-Tile
Camera
Fetch input frame
Received frame over Ethernet
Invade TCPA
Infect TCPA
Perform Harris-map on TCPA
Retreat TCPA
Filter corners and return results over Ethernet
Visualize corners on the input image
Display

Figure 6.3.: Flow diagram showing the interaction between various systems involved in Harris corner detection

of resources on the TCPA tile. If sufficient resources are available, the resulting claim is immediately returned for executing the Harris Corner detection algorithm.

Subsequently, the application program performs exactly one *infect* call per image frame to be worked on. In the process, an input frame is copied from the local TLM into the TLM of the invaded TCPA tile. The function call *InfectTCPA* is used to pass the input image address and its size to the the TCPA tile. This is followed by *CaptureResultTCPA* which will perform the *harris-map* computations on the TCPA.

- InfectTCPA(tcpaClaim, &infectFut, imgAddr, imgSize);
- nCorners = CaptureResultTCPA(tcpaClaim, &infectFut, listOfCorners, maxCornerCount);

Within the TCPA tile, the fetched image frame is split into smaller chunks which are processed in parallel by the claimed processor array. After finishing the computations of each chunk, a buffer controller generates an interrupt, requesting the next data transmission. Here, each data transmission includes filling the TCPA input buffer and writing back the output buffers to the TLM. Once the whole frame is processed, the TCPA tile sends back the position of the detected corners to the TLM of the tile that raised the *invade* request. Finally, when the entire frames has been processed, the Harris Corner detection application terminates by issuing a *retreat* command that releases the claimed PEs.

6.2.2.1. Application Mapping on TCPA

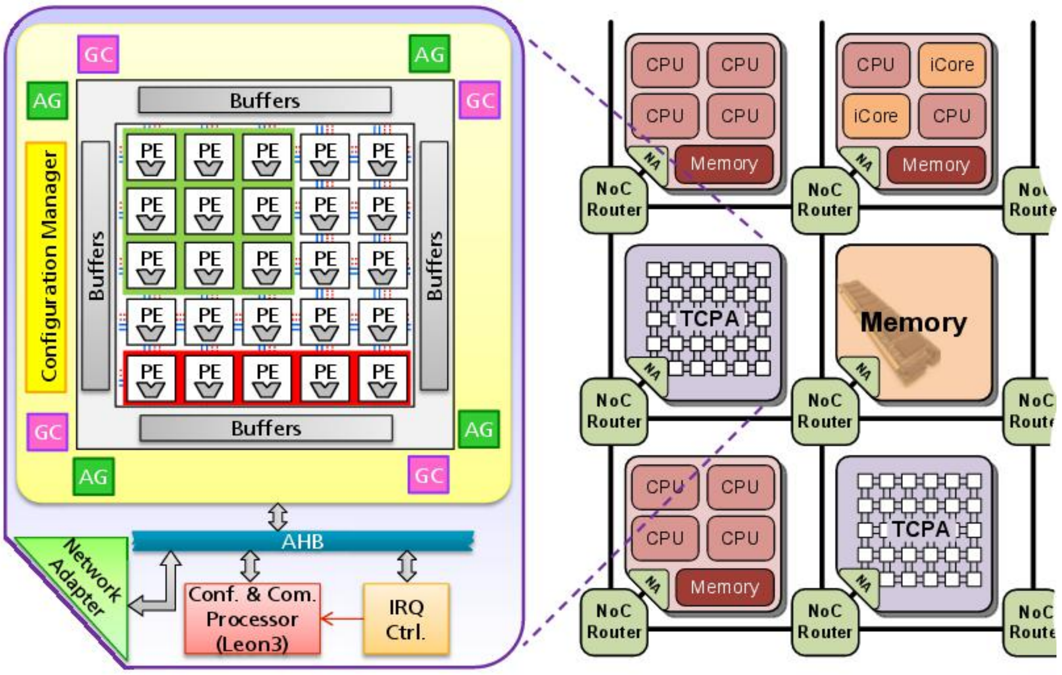


Figure 6.4.: Architecture of a TCPA tile. The abbreviations AG, GC, and IM stand for address generator, global controller and invasion manager [9]

The heart of the accelerator tile comprises of a massively parallel array of tightly coupled VLIW (very long instruction word) PEs as shown in Figure 6.4; complemented by peripheral components such as I/O buffers as well as several control, configuration, and communication companions [104]. A TCPA can exploit the parallel and direct PE-to-PE communication, where data is streaming from the surrounding buffers through the array. Through the VLIW nature of each PE and the parallel and synchronous execution of loop iterations assigned by the compiler to each PE, a TCPA nicely exploits both instruction- and loop-level parallelism.

On the processor array, the Harris Corner detection algorithm may be implemented on four PEs as shown in Figure 6.5. The first PE, which is normally one of the PEs on the border of the array, receives the pixel intensities from a TCPA buffer bank connected to it. This PE calculates the partial derivatives for Harris Corner detection. The results are sent to the other three PEs to identify the corner pixels. Thus, the full processing can be performed by using four processing elements. Finally, the corner position for each pixel are stored into the TCPA output buffer. The OctoPOS internally reserve the resources, so that the Harris Corner detection algorithm gains exclusive access to the PEs during the *harris-map* computations.

In this example, a complete (fully overlapping) implementation of the Harris Corner detection algorithm could be achieved by using four PEs. However, depending on the number

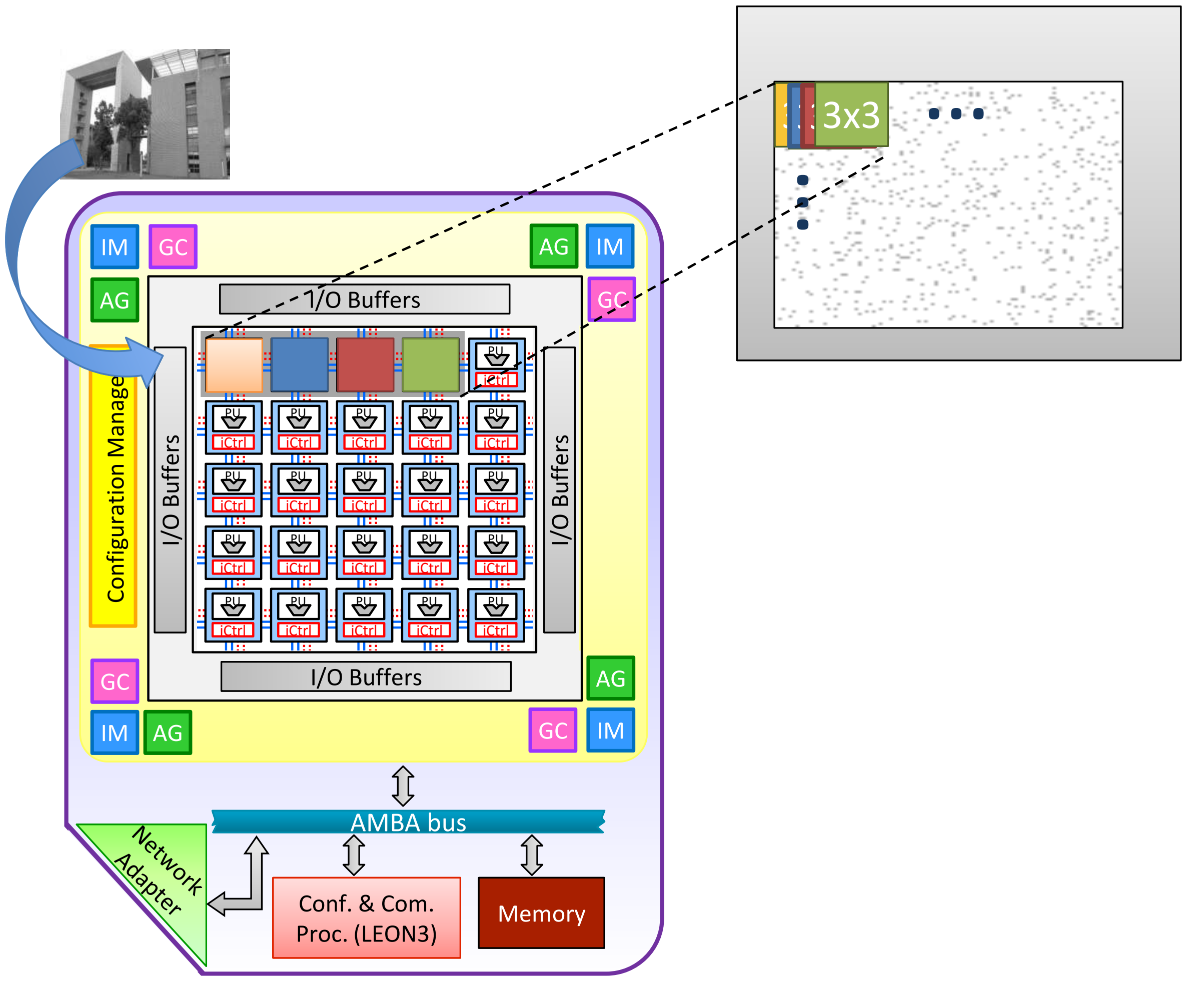


Figure 6.5.: Harris Corner detection on TCPA (application mapping). The abbreviations AG, GC, and IM stand for address generator, global controller and invasion manager.

of processor elements allocated for computation, two other mapping schemes are possible: non-overlapping or partial-overlapping, if two or three PEs are used, respectively. In both cases, one processor element calculates the partial derivatives and the corner pixels are computed on the remaining PEs. Thus, the accelerator offers different levels of quality chosen by the application according to its needs. To increase the algorithm throughput, a partitioning strategy over the input picture may be applied by adopting a locally serial globally parallel (LSGP) partitioning scheme. Latency-optimal schedules for the pixel computations on the PEs may be derived independent of the size of the image and claim, obtained using a new technique called *symbolic scheduling* [112].

In [113] the authors present a comparative case study on the accuracy and performance of implementing computationally intensive image processing algorithms, i.e., the Harris corner detector on TCPAs. The accuracy of the fixed-point computation on a TCPA architecture is compared against OpenCV (an image processing library for real-time computer vision), using 64-bit floating-point precision. Considering a set of input images, the presented case study shows a maximum error of 3.5% in terms of corner detection while achieving a much higher performance in comparison with a state-of-the-art digital signal processors.

The second stage of Harris Corner detection involves filtering the pixels with high intensity gradients based on a threshold. The pixels with intensity gradients above threshold are declared as corner points. Those corner points are then sorted in such a way that the points with very high intensity gradients appear on top of the list. Such a filtering and sorting process is more suitable for a CPU-based architecture and hence mapped to a LEON3 PE.

Finally the function *eth_send* is used to return the detected corner coordinates to the host-PC. The host-PC is responsible for visualizing the detected corners on the output image.

- eth_send(Channel_ID, resultAddr, sizeOfResults, *i*-let);

6.2.3. Results

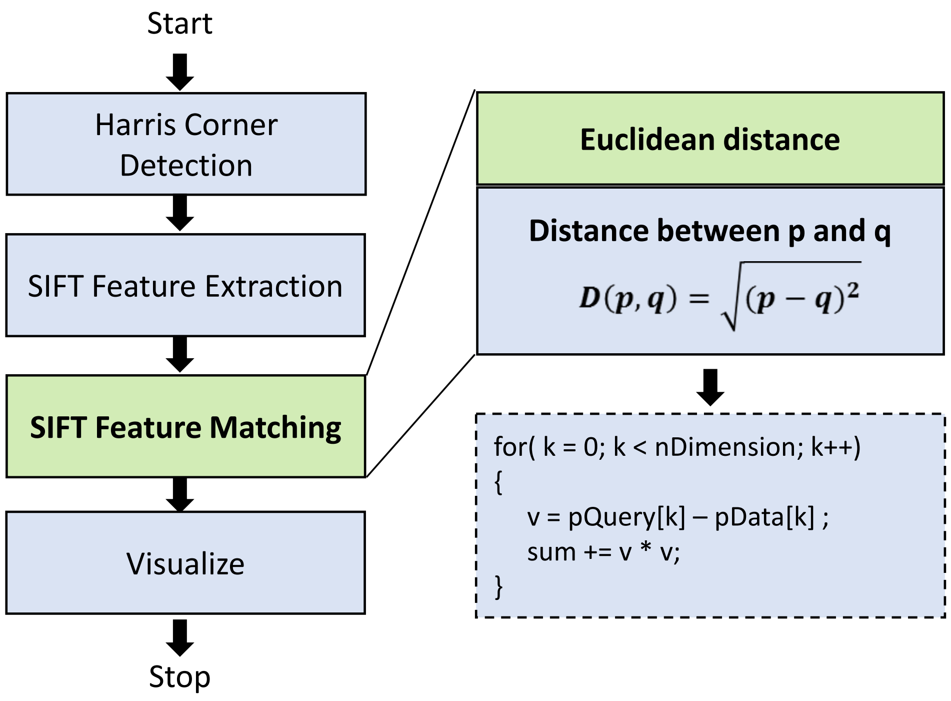
The overall goal of the evaluation model was to achieve a frame rate of 10 fps on the FPGA demonstrator. Profiling on the FPGA prototype (25MHz) indicates that 4 PEs on the TCPA are necessary to perform the corner detection at 10 frames-per-second (fps) or 100 milliseconds per frame. The algorithm, if mapped to the LEON3-based CPU units would require 10 PEs to handle the same workload at 10 fps. Therefore, the TCPA offers a higher performance (2.5×) compared to the general purpose LEON3 PE. The TCPA is also more efficient in computing *harris-map* as it can stream the pixel data through the PEs, reducing the data access latencies significantly. On the other hand, the LEON3 PEs spent significant amount of time in reading/writing data from/to the memory over the AHB Bus.

6.3. SIFT Feature Matching on *i*-Core

At each interest point detected by the Harris corner algorithm, a SIFT descriptor has to be computed. In order to recognize an object, the SIFT descriptors extract from the real-time input is then compared with the SIFT descriptors extracted from the training image. The SIFT descriptor matching is the final step in the object-recognition process, as shown in Figure 6.6. Figure 6.7 presents a time-line demonstrating the interaction between various systems involved in the SIFT descriptor matching. The algorithm for SIFT descriptor matching is explained in Section 6.3.1 followed by the description of the *i*-Core implementation in Section 6.3.2. Finally Section 6.3.3 provides some insight into the results and performance improvements achieved when compared to a pure software implementation on a LEON3 processor.

6.3.1. Algorithm

For matching, the SIFT features extracted at every feature point in the input image have to be compared against the set of SIFT features obtained from the reference (training) image. Each SIFT feature, computed as described in Section 4.1.3, comprises of 128 floating-point



```
for( k = 0; k < nDimension; k++)
{
    v = pQuery[k] – pData[k] ;
    sum += v * v;
}
```

Figure 6.6.: Flow diagram showing the various stages in the object recognition algorithm and pseudo code for SIFT feature matching

numbers. In order to compare two SIFT features $\vec{a}$ and $\vec{b}$, the squared euclidean distance d is used, as shown in Equation (6.1).

$$d(\vec{a},\vec{b}) = \sum_{i=1}^{128}(a_i - b_i)^2 \tag{6.1}$$

Figure 6.6 shows the equivalent implementation in C/C++. The matching process involves thousands of floating-point subtractions and multiplications. The complex datapaths required for floating-point arithmetic and the memory-bound characteristic make this kernel a good candidate for an implementation on the reconfigurable fabric of the *i*-Core.

6.3.2. Accelerated Implementation on *i*-Core

An *i*-Core (shown in Figure 6.8) is an extension of a general-purpose processor (LEON3) with a reconfigurable fabric, which allows loading application-specific accelerators at runtime. An application can use the fabric to speed up its computationally intensive kernels by loading the appropriate accelerators and then using special instructions (SIs), which are an extension of the instruction-set architecture of the processor (SPARC-V8 for the LEON3). Kernels often involve use of multiple accelerators, thus SIs are usually multi-cycle instructions.

Apart from dedicated fine-grained reconfigurable accelerators (embedded FPGAs), the fabric contains an interconnect for (i) data transfer from/into the fabric and (ii) transfer of

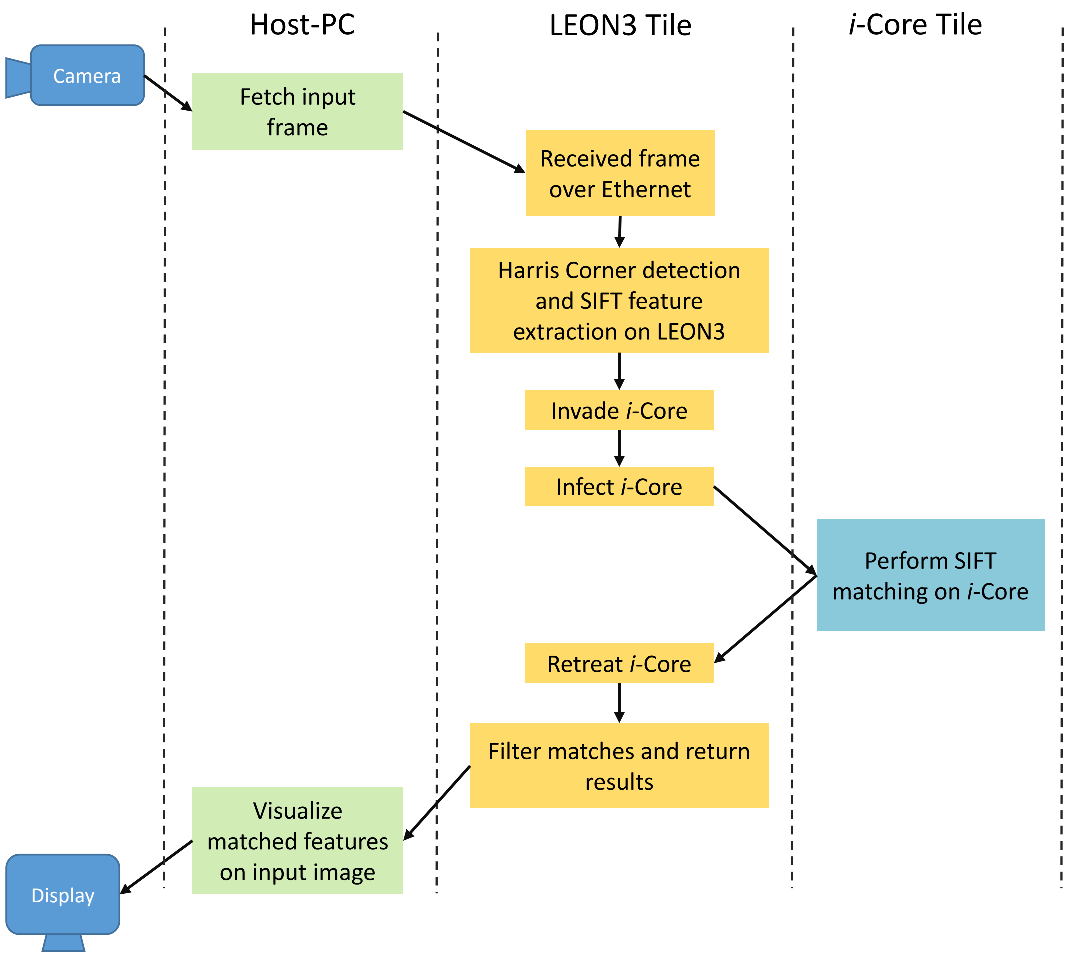

Host-PC
LEON3 Tile
i-Core Tile
Camera
Fetch input frame
Received frame over Ethernet
Harris Corner detection and SIFT feature extraction on LEON3
Invade *i*-Core
Infect *i*-Core
Perform SIFT matching on *i*-Core
Retreat *i*-Core
Filter matches and return results
Visualize matched features on input image
Display

Figure 6.7.: Flow diagram showing the interaction between various systems involved in the SIFT feature matching

intermediate results between accelerators. Two memory ports provide a high-bandwidth connection (2×128 bits) to the tile-local memory. All components of the fabric can be used in parallel, allowing interleaved processing of the current data in the accelerators and prefetching of the next data using the memory ports. When an SI is decoded in the *i*-Core pipeline, control is transferred to the *fabric controller*, which reads the micro-program [114] for the requested SI from the *SI micro-program memory* and uses it to configure the interconnect, memory ports and accelerator modes of the fabric in each cycle. Once the micro-program is finished, the fabric controller transfers control back to the *i*-Core pipeline.

In the invasive programming model, once an *i*-let is running on the *i*-Core, it issues an *invade* call to claim a number of reconfigurable accelerators. The number of invaded accelerators depends on the current utilization of the fabric (for example on the *i*-lets running on other cores in the same tile [115]). After that, the *infect* call is used to reconfigure the invaded accelerators as required by the SIs of the *i*-let.

The `siftmatch` SI was designed to accelerate SIFT Feature Matching (as described in Section 6.3.1) on the *i*-Core. `siftmatch` requires reconfigurable accelerators of the type *FMAV* (floating-point multiply-accumulate – vector) to be loaded on the fabric. FMAV accelera-

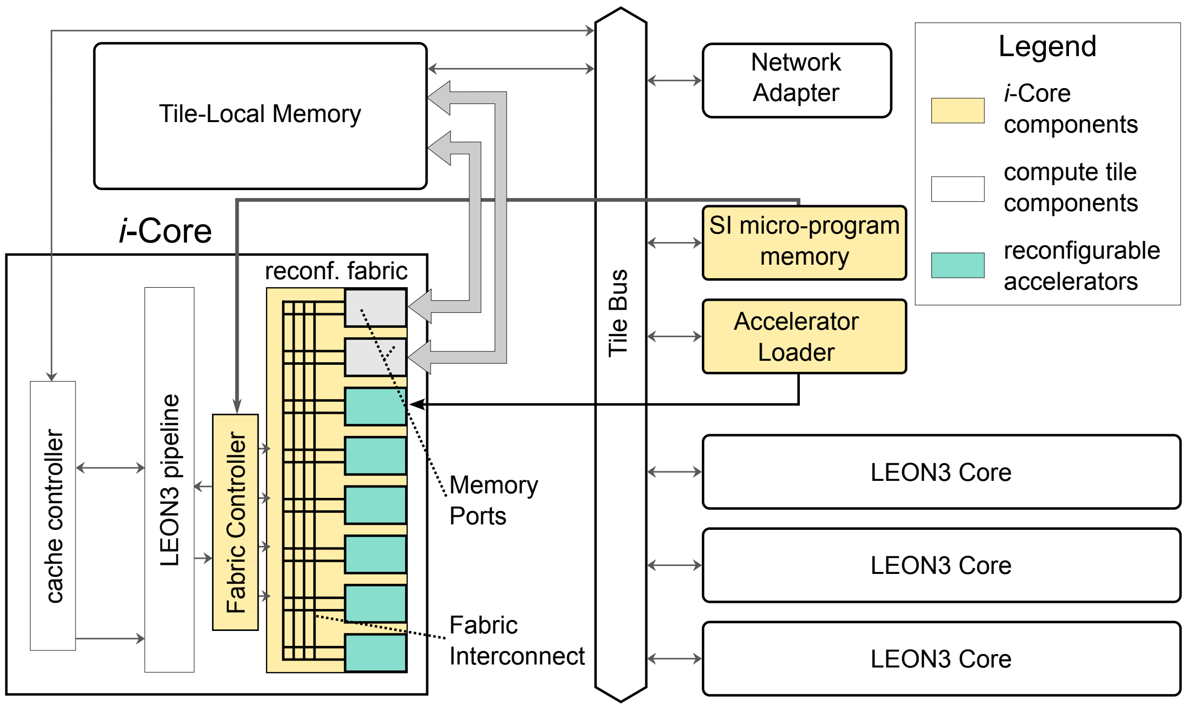


Figure 6.8.: Architecture of an *i*-Core tile [8]

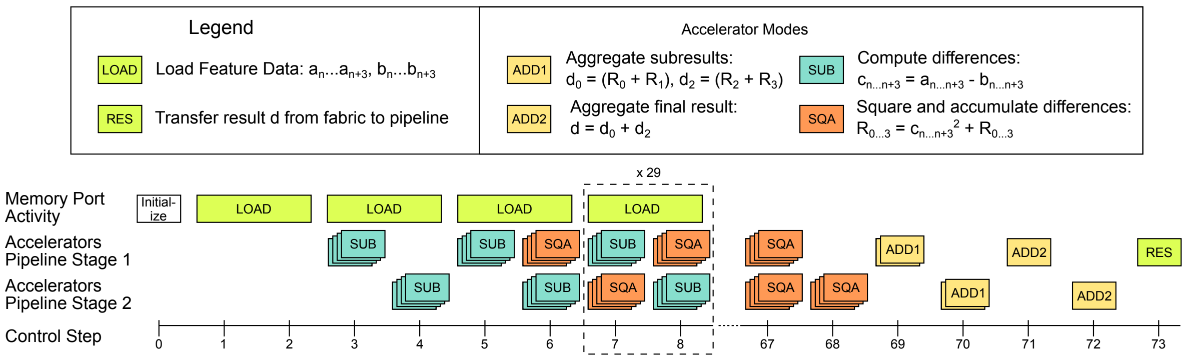

Legend
LOAD Load Feature Data: $a_n...a_{n+3}$, $b_n...b_{n+3}$
RES Transfer result d from fabric to pipeline
Accelerator Modes
ADD1 Aggregate subresults: $d_0 = (R_0 + R_1)$, $d_2 = (R_2 + R_3)$
ADD2 Aggregate final result: $d = d_0 + d_2$
SUB Compute differences: $c_{n...n+3} = a_{n...n+3} - b_{n...n+3}$
SQA Square and accumulate differences: $R_{0...3} = c_{n...n+3}^2 + R_{0...3}$
Memory Port Activity
Accelerators Pipeline Stage 1
Accelerators Pipeline Stage 2
Control Step
Initial-ize
LOAD
SUB
SQA
ADD1
ADD2
RES
x 29
0
1
2
3
4
5
6
7
8
67
68
69
70
71
72
73

Figure 6.9.: Schedule of the `siftmatch` SI [8]

tors are multi-purpose accelerators that work on single-precision floats and allow 12 modes (out of which three modes are used for `siftmatch`, see upper right part of the Legend in Figure 6.9), which are combinations of multiplication/addition/subtraction with optional storing of the result in an accelerator-internal register. Due to the long critical path resulting from some of the FMAV modes (for example, multiply $\rightarrow$ add $\rightarrow$ store in internal register), the accelerators use a two-stage pipeline in order to not reduce the fabric frequency.

To compute the difference of two feature vectors, the `siftmatch` SI utilizes the fabric memory ports and FMAV accelerators in a *schedule* consisting of multiple *control steps* that depend on the number of utilized accelerators. The schedule for four FMAV accelerators has 74 control steps and is shown in Figure 6.9. First, four elements from each vector are loaded ($a_n, ..., a_{n+3}$, $b_n, ..., b_{n+3}$ using the two 128-bit memory ports (designated LOAD operation in the figure). Next, the four differences $c_n, ..., c_{n+3}$ are computed in parallel on the four independent FMAV accelerators (SUB mode of the FMAV accelerators in the figure). Control step 3 shows the first pipeline stage of this operation and control step 4 shows the second pipeline stage of the very same operation. At the same time, the memory ports load the next four elements from both input vectors. Then the differences need to be squared and added (accumulated) to the internal registers R of the FMAV accelerator (SQA mode of the FMAV accelerators in the figure). Even though the input data for the first SQA operation would be available in control step 5, the operation is performed in control steps 6 and 7 (first and second pipeline stage), as the FMAV accelerators are already used in control step 5 to calculate the differences of the second LOAD operation. Control steps 7 and 8 show the steady state, which is repeated until all 128 elements are processed. Finally, the internal registers R of all four FMAVs are added (ADD1/ADD2 mode of the FMAV accelerators in the Figure) and the result d is transferred from the fabric to the pipeline (designated RES in the figure).

The special instruction supported by the *i*-Core, is as shown below and can be used within a C/C++ application as inline assembly. The special instruction for feature matching has two inputs and one output. Each SIFT feature to be compared consists of 128 floating point values. The address to the SIFT feature is passed along with the special instruction, as shown below, where *src_matrix0* and *src_matrix1* represents the starting address of each SIFT feature. The result will be stored in the location pointed by *res_matrix*

- asm volatile (" sigrp 0x0, %%g0, %%g0; " : :);
- asm volatile (" sift %[_src_matrix0], %[_src_matrix1], %[_res_matrix], %[_agu] "
  : [_agu] "r" (span_skip_stride),
  : [_src_matrix0] "r" (matrix0),
  : [_src_matrix1] "r" (matrix1),
  : [_res_matrix] "r" (dest_matrix)
  );

Figure 6.10 shows the internal structure of the *i*-Core tile with *i*-Core, LEON3, *i*-Core-TLM (8KB), normal LEON3-TLM (8MB), NA, etc. As described in Section 6.3.2, the SIFT feature matching using special instruction on *i*-Core achieves the best performance when the features to be compared are stored in the *i*-Core-TLM. This is because the *i*-Core can access the data stored in the *i*-Core-TLM using the dedicated high-speed 128 bit interface as shown in Figure 6.10.

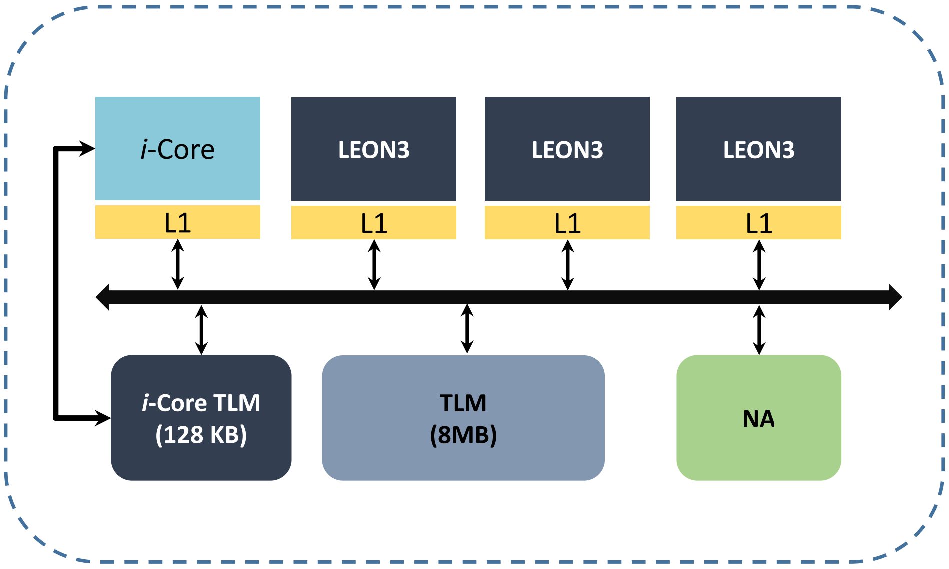


Figure 6.10.: Internal structure of the *i*-Core tile

The entire process for SIFT feature matching on *i*-Core can be summarized as follows:

1. Copy a set of SIFT features (generated by the previous stage) from the LEON3-TLM to the *i*-Core-TLM.
2. Initiate a feature comparison using the special instructions described above.
3. Copy the result back from the *i*-Core-TLM to the LEON3-TLM
4. Repeat steps 1 to 3 until all features are compared.

Section 6.3.3 provides insight into the acceleration offered by the *i*-Core special instruction and also describes some of the limitations of the current InvasIC hardware which prevent the application program from achieving the best possible results.

6.3.3. Results

Table 6.1 shows the execution time for SIFT feature matching when performed on an *i*-Core, compared to a standard LEON3 with Gaisler FPU-Lite. It can be seen that the *i*-Core SI offers a 10.5× speedup when compared to the LEON3 cores. For this evaluation process, the SIFT features for comparison were stored in the *i*-Core-TLM.

	LEON3 (with FPU-Lite)	*i*-Core SI
SIFT feature matching	0.095	0.009

Table 6.1.: Acceleration achieved for SIFT matching using *i*-Core (in milliseconds)

However, in a real-word situation, when the input data (SIFT features) is generated by LEON3 cores, they are stored in the LEON3-TLM (8MB) as it offers sufficient space to store hundreds of features at the same time. This means that the SIFT features have to copied from the LEON3-TLM to the *i*-Core-TLM before performing a matching operation using the *i*-Core's SI. One of the restrictions of the InvasIC hardware (at the time of writing this thesis) is that the NA does not support intra-tile DMA operation. This means that there is no DMA support to transfer data from the LEON3-TLM to the *i*-Core-TLM. Hence the LEON3/icore has to reply on laod/store instructions to copy the SIFT features. Under situations when large number of features have to be processed, the data copy may consume a significant fraction of the execution time affecting the overall performance. In order to clarify and isolate this problem, the evaluation process was performed using three different configurations, as described below:

1. LEON3 performing SIFT feature matching from LEON3-TLM
2. *i*-Core performing SIFT matching from LEON3-TLM
3. *i*-Core performing SIFT matching from *i*-Core-TLM (this involves data copy from/to *i*-Core-TLM)

The results from all three different evaluations models are provided below. For case-1 the input data (SIFT features) is stored in the LEON3-TLM and the results are written to the same TLM, as shown in Figure 6.11. This approach helps to avoid any data copy overhead and this resulted in an execution time of 0.095 milliseconds per SIFT feature.

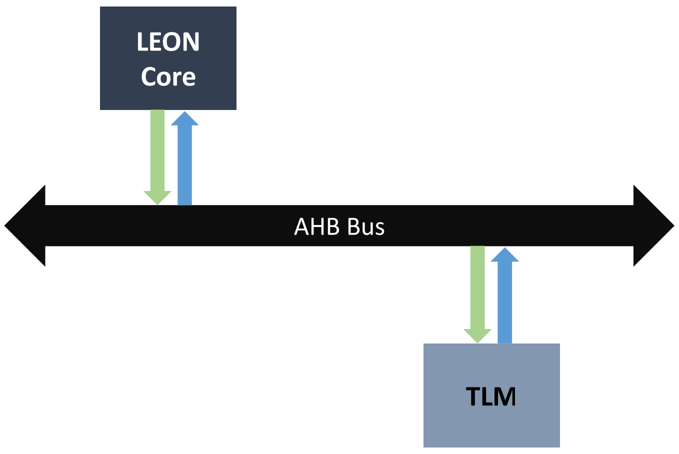


Figure 6.11.: Case-1: Evaluation model (SIFT feature matching on LEON3)

In case-2 the matching processes is accelerated using the special instruction available on the *i*-Core. The input and output data is stored in the LEON3-TLM (similar to case-1) and the data flow is as shown in Figure 6.12. Even though the special instruction can finish its operation within 0.009 milliseconds (see Table 6.1), the data access (load/store operations) from/to LEON3-TLM over the shared AHB bus consumes significant amount of time. The additional overhead reduced the overall benefits achieved by using the special instruction available on the *i*-Core. The average processing time per SIFT feature in this case was measured to be 0.051 milliseconds or 1.9× speedup compared to the LEON3 version (case-1).

In case-3 the matching processes is accelerated using the special instruction available on the *i*-Core, just like in case-2. However, the input data is copied from the LEON3-TLM to the *i*-Core-TLM before execution of the SIFT special instruction and the results are copied back

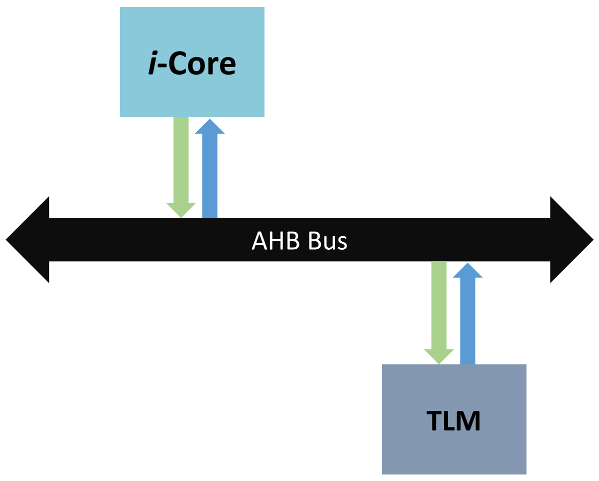


Figure 6.12.: Case-2: Evaluation model (SIFT matching on *i*-Core using LEON3-TLM)

after the execution of the special instruction as shown in Figure 6.13. This resulted in a significant acceleration as shown in Table 6.1. However, this execution time (0.009 milliseconds / SIFT feature) is only for the special instruction and it does not include the time needed to copy the SIFT features from the LEON3-TLM to the *i*-Core-TLM. The entire processing interval was measured to be 0.072 milliseconds per SIFT feature (speedup of 1.3×), including the data copy.

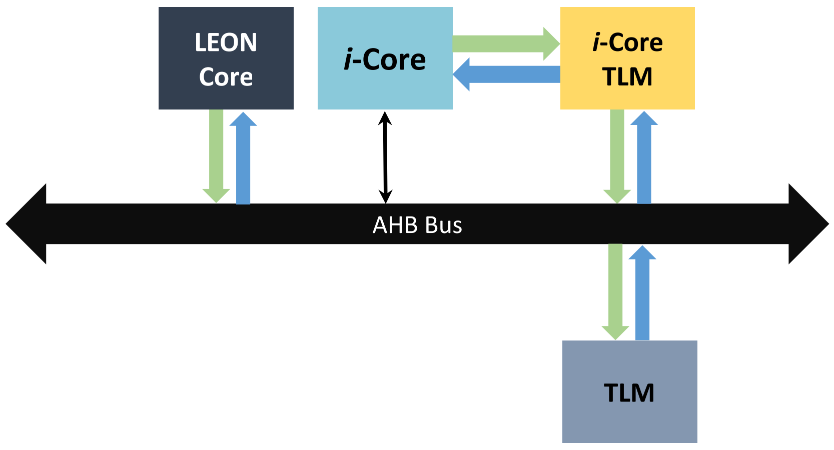


Figure 6.13.: Case-3: Evaluation model (SIFT matching on *i*-Core using *i*-Core-TLM)

Figure 6.14, shows the relation between the execution times as measured using the three cases mentioned above. As expected the case-1 offers the worst case execution time as the LEON3 processor handles the matching process using standard SPARC-V8 instructions. The execution time may be improved further using a faster FPU, instead of the FPU-Lite version used in this evaluation. Case-2 resulted in a significant improvement in execution time due to the use of special instructions available on the *i*-Core. Finally, case-3 resulted in a longer execution time compared to case-2, where the additional delay from the data copy using load/store instructions over the AHB bus significantly affected the expected benefits. Moreover, it should be noted that an *i*-Core is significantly bigger than a standard LEON3 core (approximately 2× bigger, depending on the size of the reconfigurable fabric).

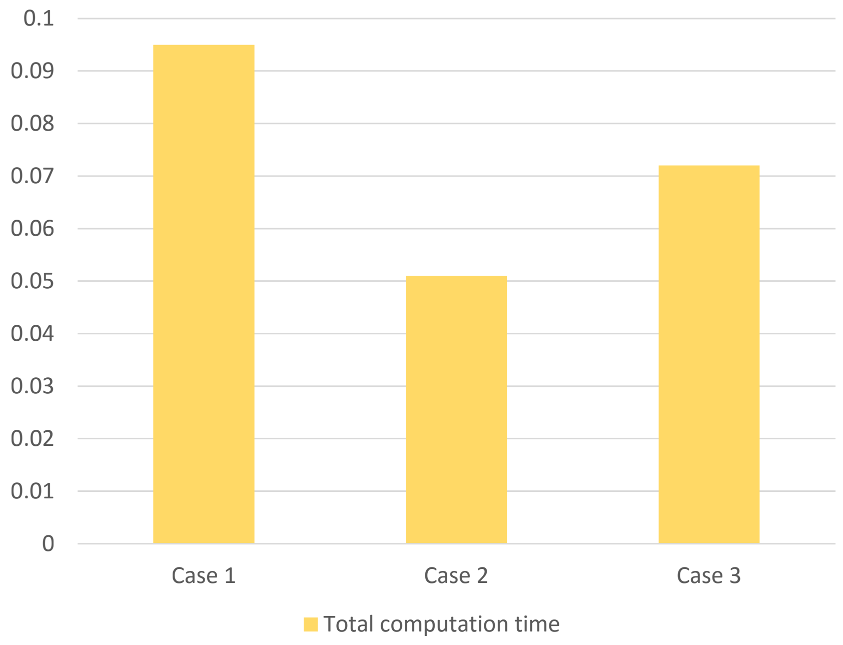
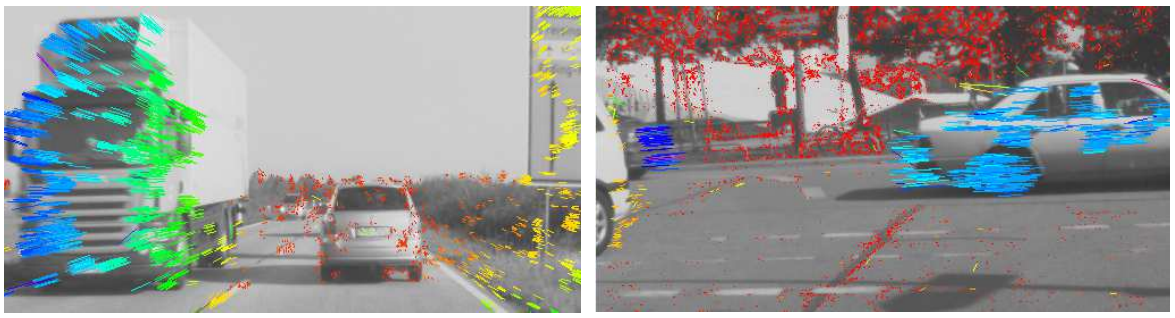


Figure 6.14.: SIFT matching, a comparison between three cases (time in milliseconds)

The results clearly point to the need for a DMA engine within the tile to handle intra-tile DMA transfers. With the help of such a DMA engine, the execution time can be brought down significantly (very close to the value of 0.009 milliseconds per SIFT feature), as shown in Table 6.1.

6.4. Optical Flow Computation on Tightly Coupled Processor Array

Optical flow algorithm has been used widely in autonomous systems where it can be used to extract important features from the environment to serve as navigational cues and to provide guidance during motion. Figure 6.4 represents optical flow, where the orientation of the flow fields (vectors) indicates the direction of motion and the length of the vector represents the relative speed at which the objects are moving.



(a) Forward motion

(b) Sideways motion

Figure 6.15.: Results from optical flow algorithm

This section describes how the optical flow algorithm can be accelerated using the massively parallel TCPA available on InvasIC. Various stages within the algorithm and their mapping on the TCPA are explained in Section 6.4.2. The results section (Section 6.4.5) presents the performance improvements achieved when compared to a conventional implementation on the LEON3 cores.

6.4.1. MPPAs for Optical Flow Computation

An efficient computation of optical flow using the census transform was published in [116] and an FPGA implementation of this algorithm was published in [117]. In addition to FPGAs, MPPAs can be used to accelerate image processing algorithms. The use of MPPAs for signal and image processing has received a lot of research interest over the past few years. Ambric (with 336, 32-bit RISC processors) [40], Xetal-II [6] (a SIMD processor with 320 processing elements), etc. are some of the MPPAs available today. Significant reduction in processing time for various applications could be achieved in the past using such MPPA architectures. For example an optical flow implementation on Ambric AM2045, running in real-time at 37 frames per second (fps) on an image stream with a resolution of 320x240 pixels, is described in [52]. The performance of this system running at 300 MHz is quite comparable to the state of the art FPGA implementation in [117]. This implementation assumes a fixed frame rate and resolution and with the algorithm statically mapped to the PEs, it consumed about 60% of the PEs on AM2045. Such implementations are not always flexible enough to handle the time-varying requirements from a realistic scenario as explained in

Section 1.2, under widely varying processor load. Moreover due to the limitations in the I/O interfaces of Ambric the PEs were stalled for more than 70% of the time, waiting for inputs.

In order to be able to flexibly run a wide range of signal and image processing algorithms as fast as possible, different kinds of reconfigurable multiprocessor-on-a-chip architectures have been developed. Examples of such architectures are DRP [118], PACT XPP [119] and ADRES [120]. The processor array of these architectures can be switched between multiple contexts by performing run-time configurations that are scheduled statically. The level of concurrency in TRIPS [121] may range from running a single thread on a logical processor composed of many distributed cores to running many threads on separate physical cores. In the CAPSULE project [122], the authors describe a component-based programming paradigm combined with hardware support for processors with simultaneous multithreading in order to handle the parallelism in irregular programs. Authors in [123] have introduced an approach based on integer linear programming for loop level task partitioning, task mapping and pipeline scheduling, while taking the communication time into account for embedded applications. Although all the above mentioned architectures have the potential capability of exploiting different level of parallelism, they all perform static resource allocation and hence the degree of parallelism for the applications are defined at compile-time.

Under the Invasive Computing programming methodology, we try to address the above mentioned issues by allowing the parallel programs to temporarily request and acquire computing resources in its neighborhood according to its computing requirements and release the resources after performing a parallel execution. This approach enables applications to self-explore the degree of parallelism available and to exploit dynamic resource requirements while avoiding fully centralized control of execution. Each PE within the TCPA architecture, presented in [71], is equipped with a hardware-based resource exploration controller that enables the processing element to explore the resource availability in its neighborhood [124]. The mentioned work established a platform to map such programs in a way that they can adapt themselves to the available degree of parallelism, with the help of resource exploration controller.

6.4.2. Various Stages in Optical Flow Algorithm

Details about the optical flow implementation on the TCPA is provided in this section with emphasis on image buffering and output generation. In addition to the computations performed inside the PE, the pattern in which the input pixels are buffered has a significant impact on performance. Inefficient buffering techniques may lower the performance due to the limited memory bandwidth on MPPAs.

This optical flow algorithm consists of three main stages (sub-algorithms) as listed below:

1. Noise reduction on the input image using a low pass filter.
2. Census signature generation for every image pixel based on its neighborhood.
3. Computing flow vectors through signature matching, within its neighborhood.

Each sub-algorithm by itself is computationally demanding and has to be performed continuously, once per video frame. Also the computing requirements of the algorithm might vary from time to time based on the application scenario. For example, when the robot is dealing with fast moving objects, the search region has to be widened significantly while it can be compressed when the robot is moving inside a building with slow moving objects around it. Such a change in behavior also makes optical flow an ideal candidate to demonstrate the concept of Invasive Computing.

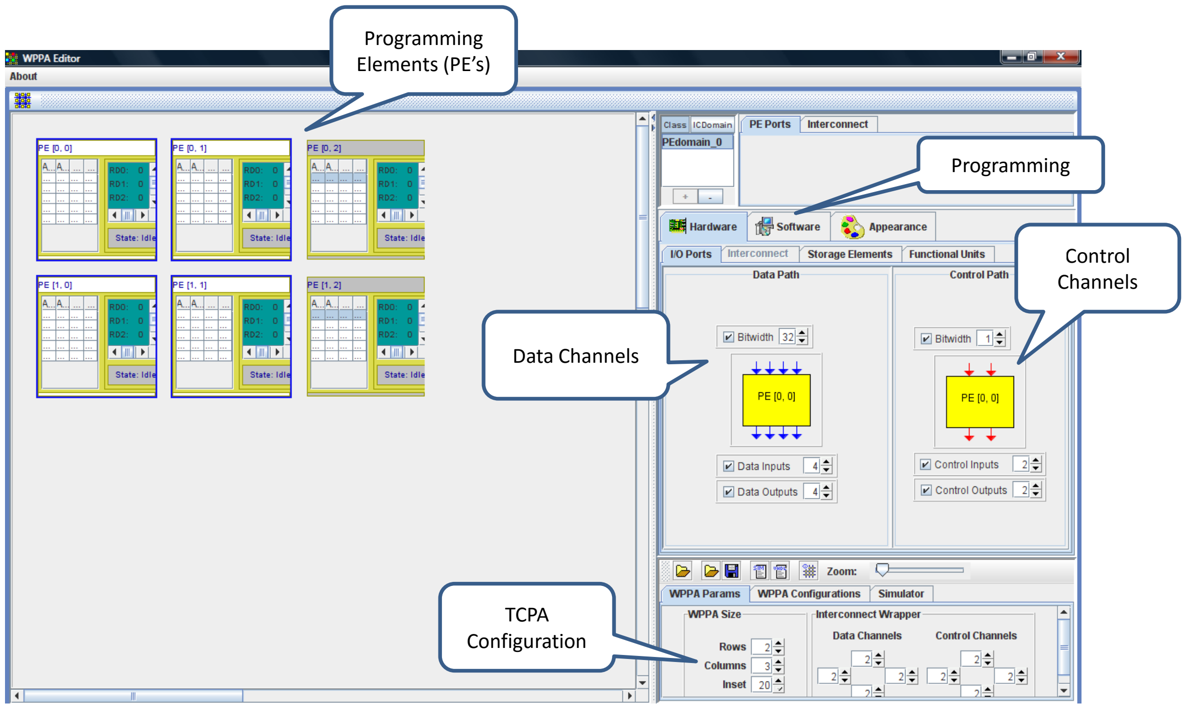


Figure 6.16.: Modeling optical flow on TCPA using the simulation framework

The optical flow algorithm was modeled on the TCPA using a cycle accurate simulator with an interface as shown in Figure 6.16. Using this graphical interface, the size of the array can be defined, the interconnects (data and control) between PEs can be configured and the instructions to be executed on each PE can be loaded.

6.4.2.1. Noise Reduction

As the first pre-processing step, image smoothing (using a low pass filter) is required to reduce noise in the input image. A noisy image will significantly reduce the number of possible flow vectors in the final output [117]. Noise reduction is a pixel-based operation, where each pixel is transformed based on the value of its neighboring pixels, using Equation (6.2), where $P(x,y)$ is the value of the pixel at location (x,y) and $W(x,y)$ is the corresponding weight.

$$P_s(x,y) = \frac{\sum_{x_1=x-w/2}^{x+w/2} \sum_{y_1=y-w/2}^{y+w/2} P(x,y) * W(x,y)}{\sum W(x,y)} \tag{6.2}$$

The window size is represented by w (set to a value of 3) and $P_s(x,y)$ is the resulting pixel. The pixels within the neighborhood as defined above, form a 2D window with $P(x,y)$ at the center. The window then slides over the complete image generating $P_s(x,y)$ for every possible value of x and y within the image boundary.

The input buffers (RAM) of the TCPA serve as line buffers, with each buffer storing one complete row of the input image. The buffer controller handles the data fetch operations as explained in [125]. The PEs along the periphery of TCPA have direct RAM access and they access the pixel data as shown in Figure 6.17. It should be noted that each PE processes only a single row within the 2D window and the individual results from PEs has to be combined to get the final resultant pixel.

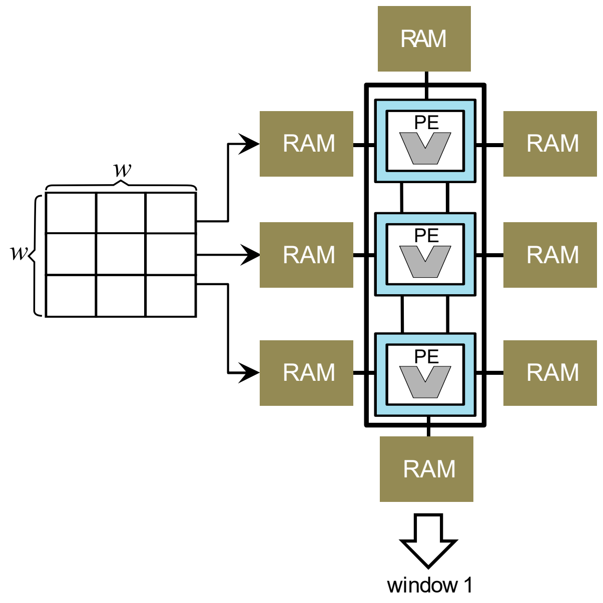


Figure 6.17.: Basic image filtering on a TCPA [14]

After processing one complete row of the window, every PE outputs its result to the next PE and the final result is computed by the bottom PE, according to the mapping scheme in Figure 6.17. When the output buffer is full an interrupt is raised and the results are moved to the external memory by the DMA controller which is part of the TCPA architecture as explained in [125]. A detailed explanation of the TCPA memory architecture and external bus interface is beyond the scope of this thesis.

According to this mapping scheme, the size of the window (w) decides the number of PEs required for program execution. It is interesting to note that for a sliding window approach, $[(w-1) \times (w)]$ pixels of any window is also part of the nearest neighboring window. Hence a second column of PEs if available, can process the second window by reusing the $[(w-1) \times (w)]$ pixels read for the first window along with an extra w pixels which is specific to the neighboring window. This leads to an acceleration of the overall algorithm as more windows are now processed simultaneously, as shown in Figure 6.18.

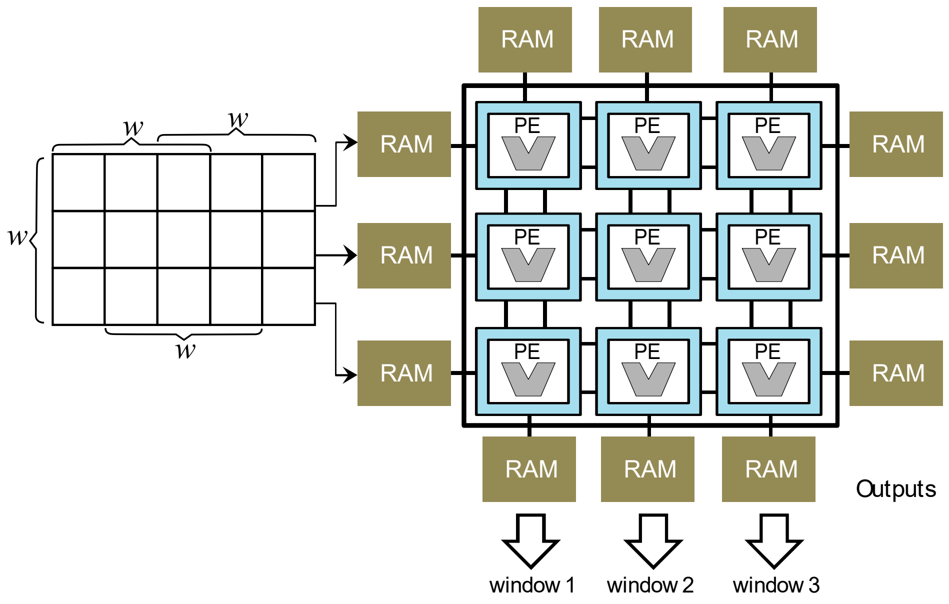


Figure 6.18.: TCPA processing multiple windows simultaneously [14]

6.4.2.2. Census Signature Generation

After image filtering, the next step is to generate census signatures from the filtered image. For this step, a technique described in [117] is used and it is briefly explained in Figure 6.19.

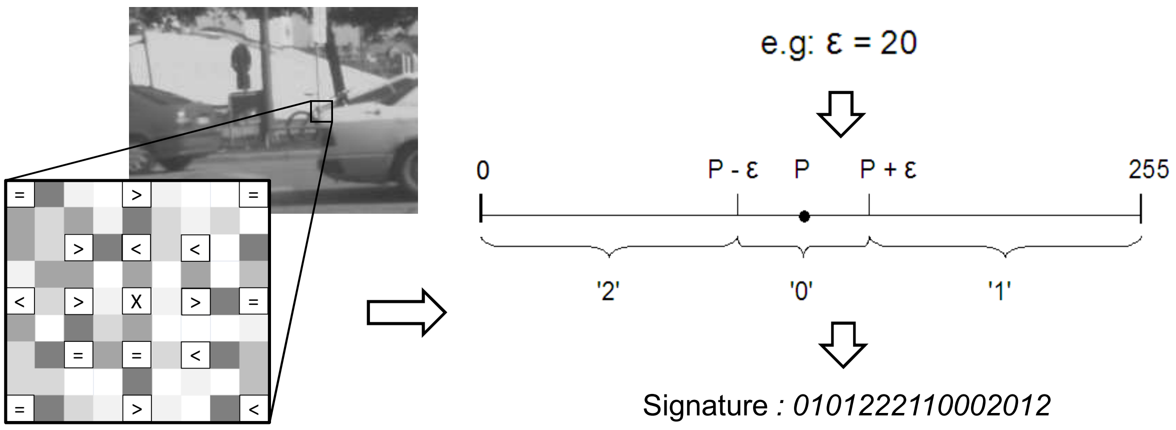


Figure 6.19.: Signature generation technique [14]

A unique signature is calculated for every pixel in the frame by comparing its value (marked by ’x’) with its neighboring pixels located within its window region. The result of each comparison consists of 2 bits and is calculated using an ϵ value as shown in Figure 6.19. Unlike image filtering, only a subset of the neighboring pixels are required for signature generation. Those pixels are marked by the symbols ($<, =, >$) and the symbol indicates the relation of the central pixel with the neighbor at that particular location.

From Figure 6.19 it is clear that the maximum number of pixels to be processed per column is five and this occurs at the center of the window. Hence within the TCPA, five PEs per column are required for computing the signature. The filtered image (input image) is buffered in a similar fashion as in case of image filtering. Moreover, the signature generation stage of the algorithms also involves a sliding window and hence the processing can be accelerated using multiple columns of PEs as shown in Figure 6.20.

According to the mapping scheme in Figure 6.20, all the pixels within a window are buffered in the RAMs on the west side while only the center pixel is buffered into the RAM on the north side of the array. More information related to this buffering scheme is provided under Section 6.4.4. Similar to the approach adopted for image filtering, the results are propagated through the PEs to the RAMs located at the bottom of the array. A unique signature is thus generated for each pixel in the input frame, forming a signature image as shown in Figure 6.21 and this is used for computing the flow vectors as explained in the next section.

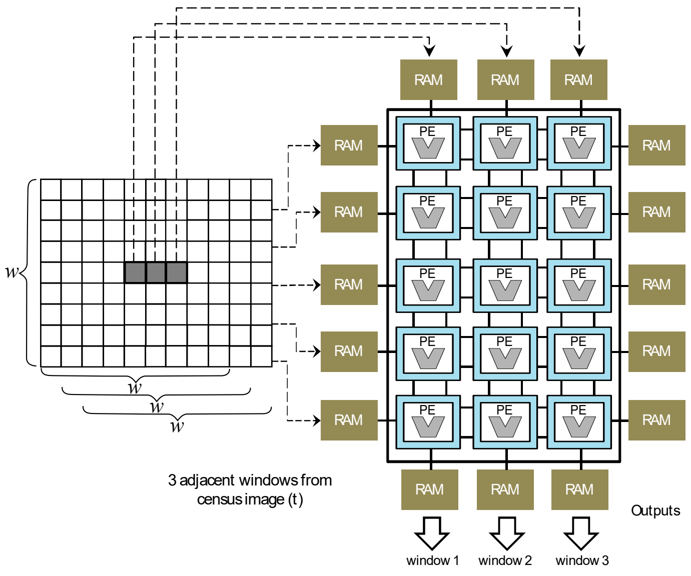


Figure 6.20.: Mapping scheme for signature generation [14]

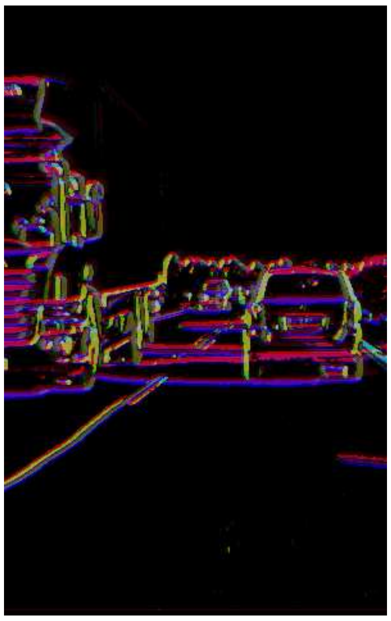


Figure 6.21.: Sample signature image

6.4.2.3. Flow Vector Generation

Flow Vector Generation is the final stage of the optical flow computation which generates flow vectors representing the relative motion between the camera and the objects in the frame. For vector generation two consecutive signature images are required (from frames (t-1) and (t)). As explained before, a signature represents the relation of the current pixel with its neighborhood. Thus, if the signature of a pixel match with the signature of same/another pixel in the next consecutive frame, then the pixel is said to have moved from its primitive location in frame (t-1) to the new match location in frame (t).

In order to understand this process, consider a window of size $[w \times w]$ centered at coordinates (x, y) in the signature image (t). A signature at location (x, y) in the frame (t-1) is compared with every signature within its window in the next frame (t). This is explained using the Figure 6.22. If a unique match is found (e.g., at location (x_i, y_i)) then a vector can be established connecting location (x, y) and location (x_i, y_i). This operation is repeated for every pixel in frame (t-1), using a sliding window, generating flow vectors wherever possible.

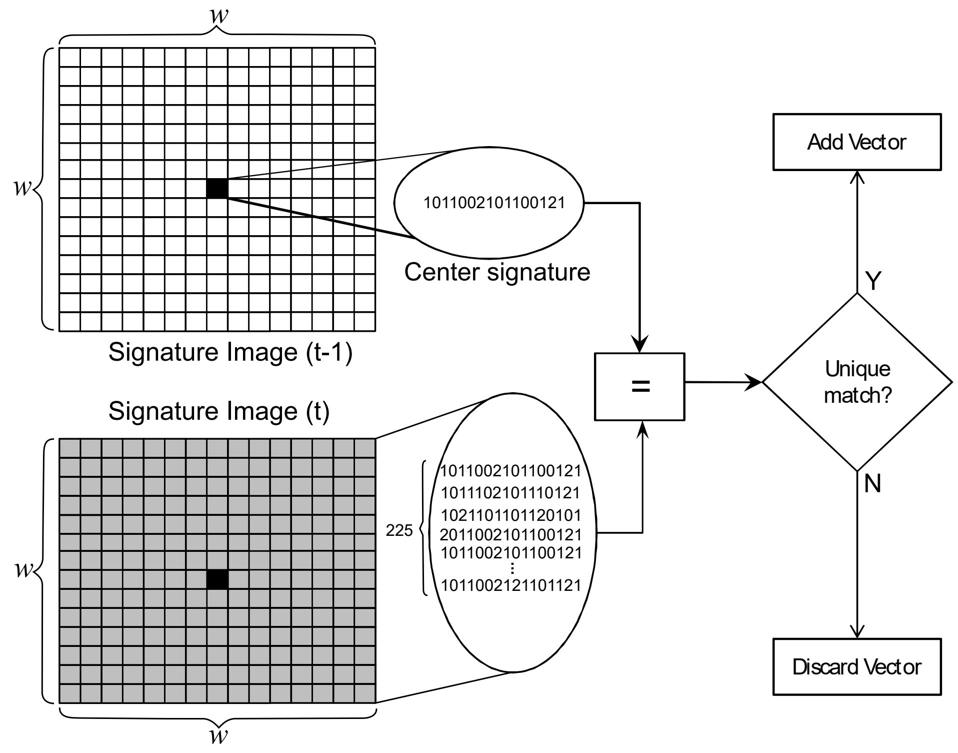


Figure 6.22.: Flow vector generation logic [14]

To start vector generation, the signature image computed in Section 6.4.2.2 is buffered into the TCPA in a very similar fashion as explained for signature generation, with a difference that no rows/columns are skipped. The size of the window (w) decides the search region. Larger the value of w, longer the vectors and hence fast camera motion or fast moving objects could be detected. But this also means that more pixels has to be processed per window, leading to higher computing requirements. The output buffering remains exactly same as in the previous stages of the algorithm. A sample output with ARMAR-III robot attempting for object grasping is shown in Figure 6.23.

6.4.3. Optical Flow in Invasive Notation

As the optical flow algorithm has three main stages, three phases of *invade* and *retreat* operations has to be carried out one after the other. This ensures that the application does not

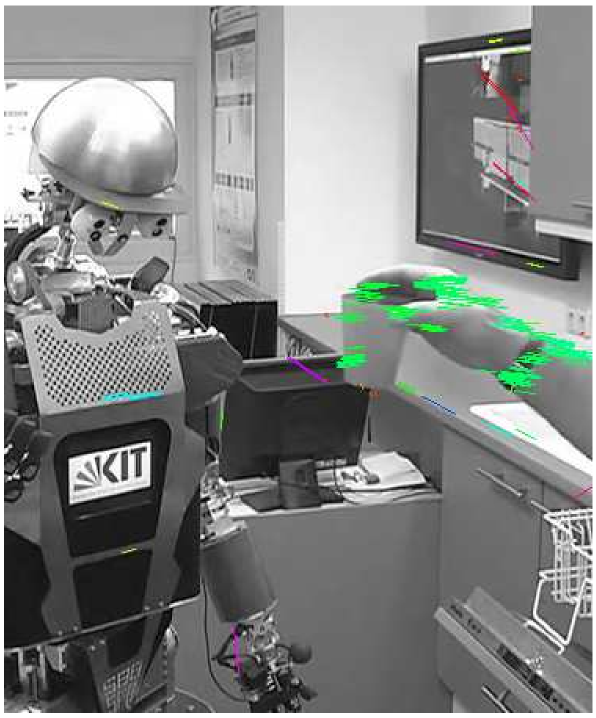


Figure 6.23.: Vision assisted grasping by ARMAR III humanoid robot

hold the resources unnecessarily, leading to higher resource utilization. Processing starts on a single PE (LEON3) where it calculates the amount of resources required to meet the applications real-time constraints under a particular frame rate and resolution. For example to meet the real-time constrains under a high frame rate, the application can raise an *invade* request for a $[3 \times 3]$ array of PEs for image filtering. If a set of $[3 \times 3]$ is not available, the operation can continue with a PE array of size $[3 \times id]$, where *id* is called *invasion depth*. The value of *id* can vary from one to window size (w). In general, based on the computing requirements, the value of *id* can be calculated and an invasion request can be raised by the seed PE, with the help of the invasive co-processor. It can be shown that the overall processing time decreases with increase in value of *id*. After the invasion, an infect stage begins by loading the predefined instructions and interconnect structure into the invaded PEs.

After completing image filtering the PEs are released through a *retreat* operation. This is followed by a second *invade* for a PE array of size $[5 \times id]$, where *id* varies from one to five. In a similar fashion, the vector generation stage use an array of size $[w \times id]$ where w is the window size or search region. It should be noted that multiple invasions can happen simultaneously as shown in Figure 6.24. Such invasions can handle image filtering and signature generation simultaneously, reducing the external memory bandwidth requirement and thereby the overall processing time, as the temporary results need not be written back to the external memory. Figure 6.24 shows an example scenario for resource-aware computing, where there is an application already running on the TCPA and two other applications try to invade from two different seed points. Due to their real-time requirements they decided to set their *id* values to the maximum possible value (3 and 5 respectively). The arrows represent the direction of invasions. After invasion, it could be seen that the image filter application could obtain an *id* of three, equal to its target, while the *id* for signature generation was limited to four, due to the presence of a 3rd application. Such a flexible implementation allows the signature generation to continue with its operation rather than waiting for remaining PEs to be released by the 3rd application.

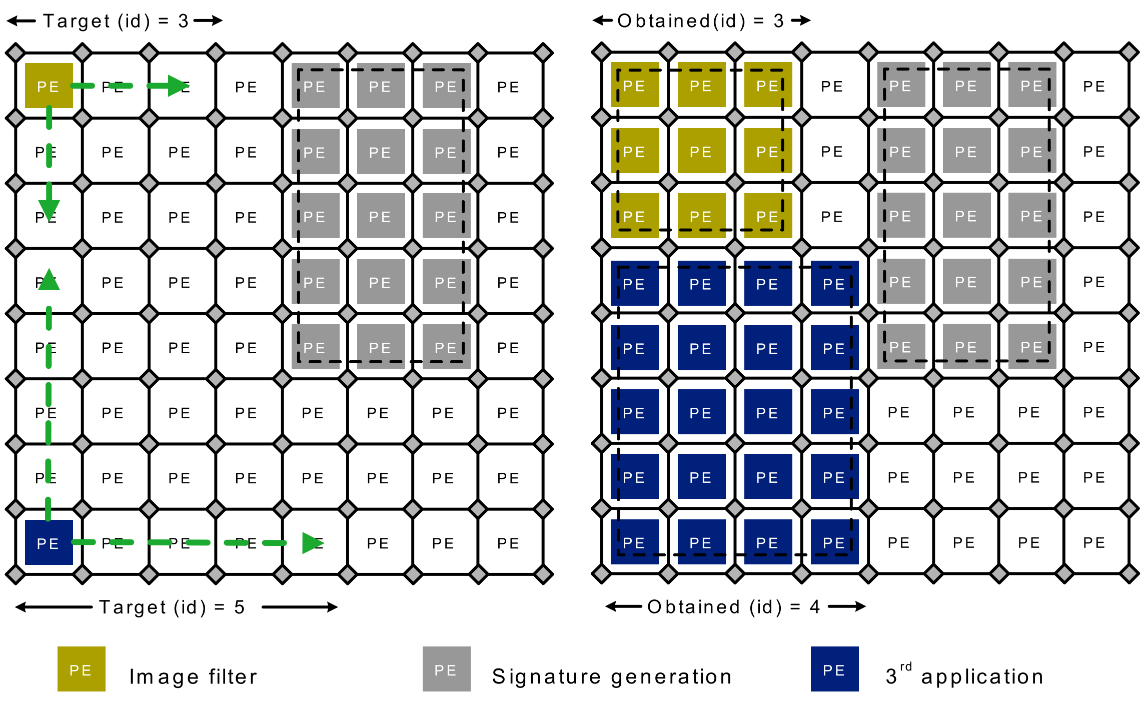


Figure 6.24.: Resource aware 2D invasions on TCPA

6.4.4. Implementation

This section describes the programming aspects and interconnect configuration for various stages of the optical flow algorithm on TCPA.

6.4.4.1. Noise Reduction

For noise reduction a low pass filter of size $[3 \times 3]$ with weights as described in [117] is used. The weights are power of two and hence all the operations can be carried out using logical shift operations. To begin image filtering, an invasion is carried out for a 2D region of PEs, defined as $[3 \times id]$, so that it can process *id* number of windows simultaneously. Based on the reply from the invasion controller, the application then fixes its final *id* value and starts the image filtering operation. A new configuration is now loaded into each PE, where each PE is configured to have two input and two output ports. The ports $inp0$ and $inp1$ of each PE along the periphery are connected to the RAMs while the ports $inp0$ and $inp1$ of remaining

PEs are connected to the output ports of the periphery PEs, as shown in Figure 6.25. This interconnect structure helps the PEs (without direct RAM access) to access the data in the RAM.

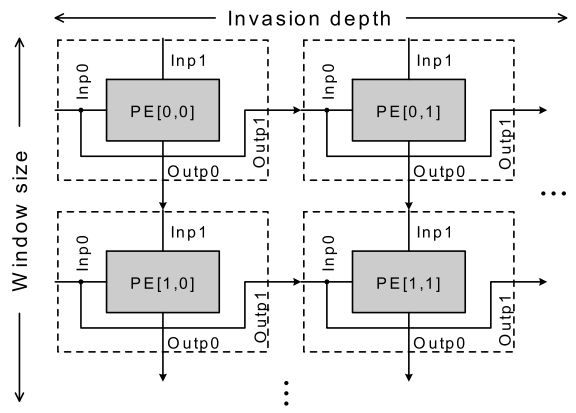


Figure 6.25.: TCPA Configuration for image smoothening

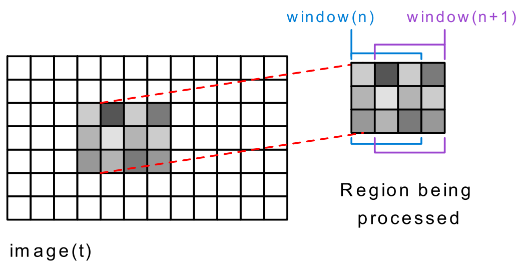


Figure 6.26.: Discard operation

After an invade operation, the PEs start pixel processing where each PE processes one complete row of the current window and the result is sent to the PE located below, through its output port $outp0$. The neighboring PE then adds its own result to the result coming from its neighbor above and transfers it to the next PE. Thus the final result is computed by the 3rd PE, for a window size of $[3 \times 3]$. Each PE now flushes its internal registers and continues processing the upcoming windows, in a similar fashion as explained before. Figure 6.26 explains the input data flow (with the invasion depth set to a value of 2), processing two windows simultaneously. Because of the TCPA architecture, only the PEs along the periphery can access the RAM. If other PEs want to access the RAM data, this has to be read by the periphery PE and any data read by the periphery PE is automatically available to other PEs in that row through the special interconnect configuration. Hence to process two windows simultaneously, an additional column of pixels has to be read by the periphery PE, beyond its own window region. Thus the actual window size is now increased to $[3 \times 4]$ or $[w \times (w + id - 1)]$ in general. This additional set of pixels belong to window 2, and will be discarded by PEs processing window 1 and vice versa.

Number of clocks required to produce one smoothed pixel is given by Equation (6.3), where CPP is clocks/pixel, id is the invasion depth, w is window size, C_p is the number of clock cycles for processing one pixel within the current window, C_d is the number of clock cycles for the discard operation, and C_{const} is the time for resetting the registers after processing one window and move on to the next. After processing all the pixels in the current frame, the PEs are released through a *retreat* operation.

$$CPP = \begin{cases} \frac{(C_p * w) + (id-1)C_d + C_{const}}{id}, & id \neq w \\ \frac{(C_p * w) + C_{const}}{id}, & id = w \end{cases} \tag{6.3}$$

Figure 6.27 provides some data indicating how the clocks-per-pixel value varies with windows size and invasion depth. These results are based on the cycle accurate TCPA simulator.

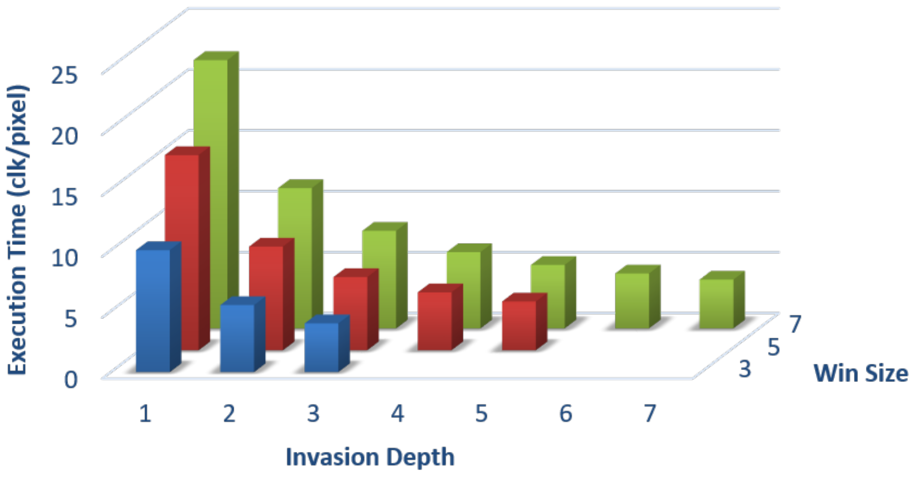


Figure 6.27.: Variation in execution time for image filtering application based on window size and invasion depth

6.4.4.2. Signature Generation

As described in Section 6.4.2.2, signature generation requires a sliding window of $[9 \times 9]$ pixels where the maximum number of pixels to be processed per column is five (see Figure 6.19). Hence the processing starts by invading a $[5 \times id]$ region inside the TCPA. Once the invasion is complete, the id value is fixed and the infect stage starts loading the new program and a new interconnect configuration as shown in Figure 6.28.

The center pixel is buffered through the port $inp2$ of PE[0,0] to PE[0,id] and this input port is also connected to the input port of the next PE (in that column) through output port $outp1$. This means the value reaches all PEs in the same column at the same clock. The remaining pixels within the window are buffered through the input port $inp0$ and hence available to all PEs in the same row. Each PE upon receiving the inputs, performs a comparison between two inputs as explained in Figure 6.19. It then performs the same operation on the next set of inputs and concatenates the results to form a signature. Once a complete row within the current window is processed, each PE outputs its partial signature on its output port $outp0$. Now the partial signatures need to be combined to generate a 32 bit complete signature and this is handled by the PE at the bottom of the invaded region (accumulator PE). Hence the individual results are propagated downward through the other PEs (through their $outp0$ port) and collected by the PE at the bottom. The final 32 bit signature is obtained at $outp0$ of the accumulator PE.

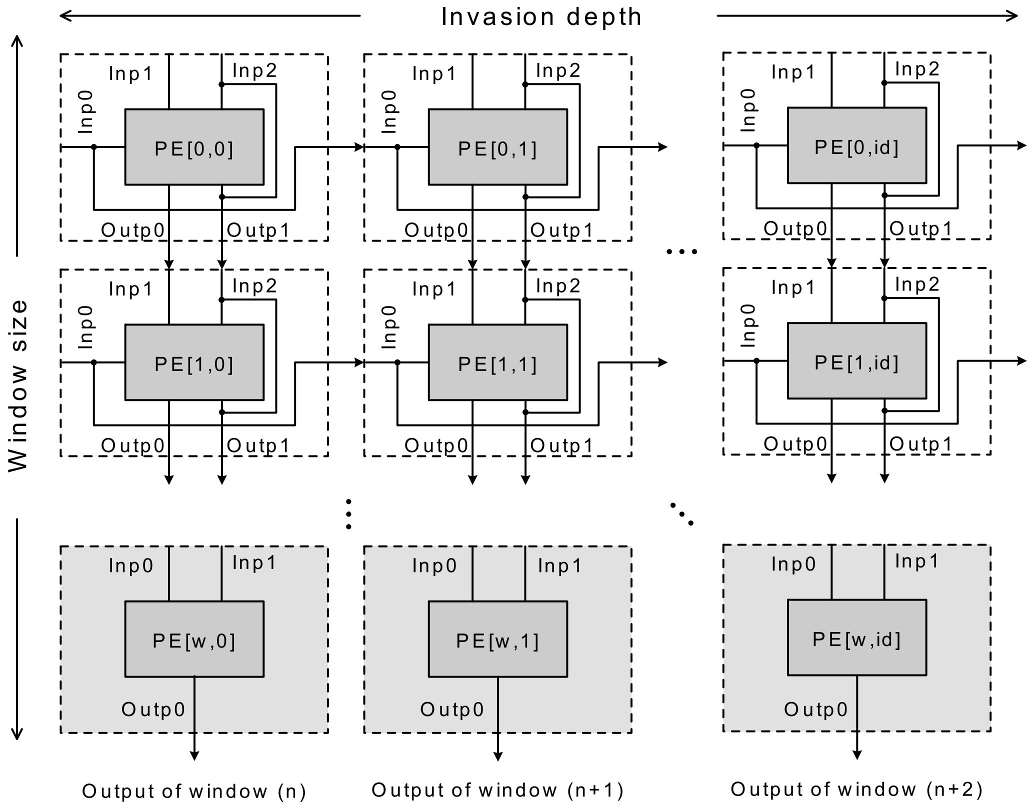


Figure 6.28.: TCPA configuration for signature and vector generation

As compared to image filtering in Section 6.4.4.1, the signature generation involves an additional propagate operation for sending the individual results to the accumulator PE. This can be avoided by connecting each of the PEs performing signature generation, individually to the accumulator PE. But this leads to complex routing schemes and hence this is not a preferred option for large windows. Therefore the results are propagated through the PEs and this leads to an additional factor in the performance figures as shown in Equation (6.4), where w is the window size (equal to 9), w_{act} is the maximum number of active pixels to be processed within any column in the window (equal to 5) and C_s is the clocks required to propagate one result from a PE's input port to its output port.

$$CPP = \begin{cases} \frac{(C_p*w)+(w_{act}-1)C_s+(id-1)C_d+C_{const}}{id}, & id \neq w \\ \frac{(C_p*w)+(w_{act}-1)C_s+C_{const}}{id}, & id = w \end{cases} \tag{6.4}$$

The PEs operating in this stage can be classified into two categories or domains (pixel comparison domain or domain-I and result accumulation domain or domain-II). All the shaded PEs form domain-II while the remaining PEs form domain-I, as shown in Figure 6.28. PEs in the domain-II are pipelined to the PEs in domain-I. Moreover, domain-II requires less number of clocks to complete its computation, as the accumulation operation performed by domain-II is a simple operation compared to signature generation by domain-I. Hence domain-II is not considered in Equation (6.4). For more information on PE domains, please refer to [71].

From Equation (6.4), it is clear that the performance of the system is dependent on id (when w is fixed) and that the CPP value does not scale linearly with id, due to the presence of id in the numerator. During the discard cycle, each PE waits (for a specific number of clock cycles) for an input pixel to arrive and then discards it. The number of discard operations is proportional to id, which leads to a saturation in performance figures over the range of values of id.

In order to resolve this issue a new mode of operation was developed such that the system can perform the propagate and discard operations simultaneously. Such an approach is realistic because the number of input discard operations is always less than or equal to the number of result propagation operations. With this modification Equation (6.4) reduced to Equation (6.5) where the performance scales linearly with id when the window size is fixed. The graphs in Section 6.4.5 shows the performance impact of the new model.

$$CPP = \frac{(C_p * w) + (w_{act} - 1)C_s + C_{const}}{id} \tag{6.5}$$

6.4.4.3. Vector Generation

The vector generation stage is described here with the details about the interconnect structure and operations inside the invaded PEs. For detecting fast movements, a wider search region is necessary. Hence, based on the scenario, the seed PE fixes the invasion depth and begins invasion, with a request to *invade* $[w \times id]$ region of the TCPA array. As described in case of image filtering, each PE processes its corresponding row within a 2D window. An incoming signature from frame (t-1) has to be matched with signatures from frame (t). As explained in Figure 6.22, in case of unique match, a vector is generated with its endpoints extending from center of the window to the location of the matching signature.

As one PE processes only a section of a complete window, the results are not final and have to be compared with results of other PEs for multiple match condition. The interconnect structure for vector generation is same as the one used for signature generation (see Figure 6.28). The signatures within the first window of frame (t) is buffered into the TCPA through $inp0$ of PEs on the west periphery. The central signature from frame (t-1) is buffered into the TCPA through $inp2$ of the PEs along the north periphery.

Each PE performs a signature matching between the central signature and the signatures from the corresponding row within the window and outputs the result in the form of (x,y) coordinates and match count to the neighboring PE below it. The various steps involved in signature matching together with the instructions used in each step, is shown in Figure 6.29. Similar to signature generation, the individual results are propagated to the accumulator PE through the other PEs. The accumulator PE is programmed to receive the (x,y) coordinates from all other PEs along with their match count. It then decides if there was a

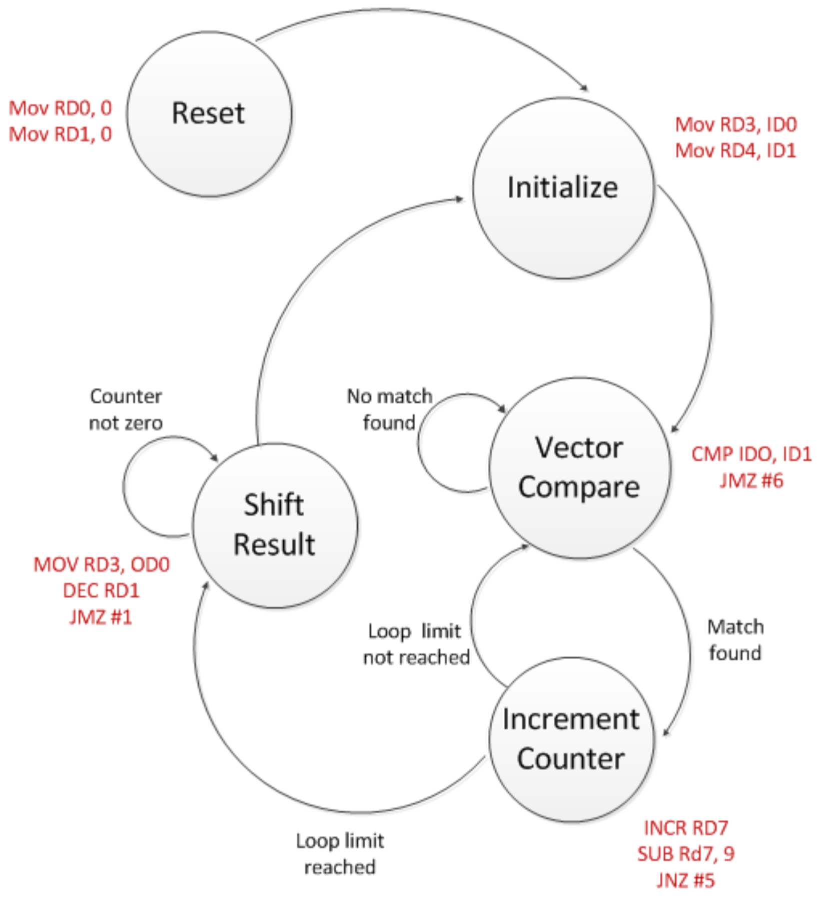


Figure 6.29.: Flow diagram showing various stages involved in vector matching

unique/multiple/no-match and then outputs the vector in the proper format through its $outp0$. The accumulator PEs have a different interconnect and program domain. The two domains work in a pipelined fashion to improve the overall throughput. The vector generation also involves discard and propagate operations similar to signature generation stage and hence the clocks/pixel values can be calculated using the Equations (6.4) and (6.5), with a difference that ($w = w_{act}$) as all the signatures within the window are active this time.

6.4.5. Results

The performance benefits of the optical flow implementation based on Invasive Computing on TCPA is evaluated in this section, using clocks/pixel (CPP) value as an index for comparison. A low CPP value indicates low execution time or low power consumption (by reducing the operating frequency). Figure 6.30 and Figure 6.31 show various graphs indicating the performance of the algorithm running on TCPA. These graphs were plotted using Equations (6.3), (6.4) and (6.5), for a fixed window size. The dotted lines indicate the results from sequential discard and propagate operation (represented by (a)) and the continuous lines indicate the scenario with simultaneous propagate and discard (represented by (b)). Note that the result propagation is not required in case of image filtering as each PE combines its own result to the result from the neighbors and output just one value. Hence there exists only one scenario for image filtering.

The values for C_p, C_s, C_d and C_{const} were obtained through simulation using a cycle accurate simulator published in [126] and are listed under Table 6.2. The time for TCPA reconfiguration (C_{conf}) is also listed under Table 6.2. However, the value of C_{conf} is not considered in the performance graphs, as its values become insignificant considering the fact that the configuration is done only once for processing a VGA frame (640 × 480 pixels).

Stage	$C_p(a)$	$C_s(a)$	$C_d(a)$	$C_{const}(a)$	$C_p(b)$	$C_s(b)$	$C_{const}(b)$	C_{conf}
Filter	2	-	1	1	-	-	-	42
Signature	2	1	2	8	2	2	9	105
Vector	2	2	2	4	2	2	4	101

Table 6.2.: Simulation results

Graph in Figure 6.30 indicates variation in CPP value against id, for all three stages of the algorithm. The graph clearly shows that the CPP value decreases with an increase in invasion depth. Also it can be seen that for large values of *id* for vector generation, approach (b) offer a better performance. This is because the number of discard operations is proportional to invasion depth and for large values of *id* the time for discard operation becomes a dominant factor compared to processing time and reduces the overall performance.

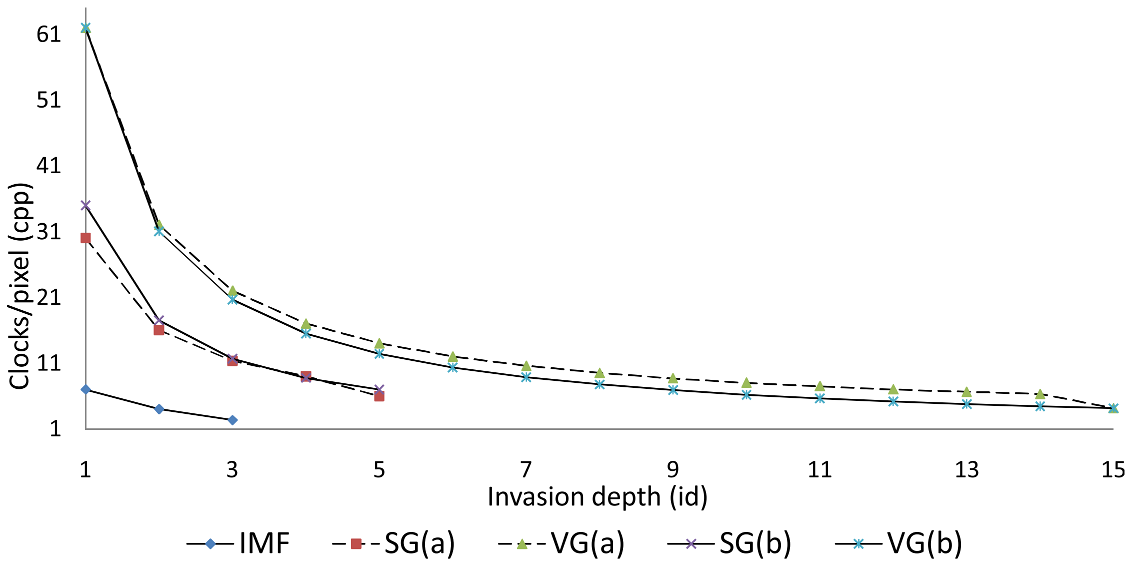


Figure 6.30.: Clocks/pixel Vs Invasion Depth (IMF: image filtering, SG: signature generation, VG: vector generation)

Figure 6.31 shows the acceleration achieved using the new implementation. These graphs were obtained by taking the ratio `[CPP(id=1) : CPP(id=n)]`. The obtained acceleration tends to saturate in case of approach (a) and this is clear from Equation (6.4), due to the increasing number of discard operations with invasion depth. This problem is fixed in approach (b) and the performance doubled while changing from an invasion depth of one to two. Moreover, the acceleration is consistent over the range of values for *id*, as discard operation is performed simultaneously with result propagation.

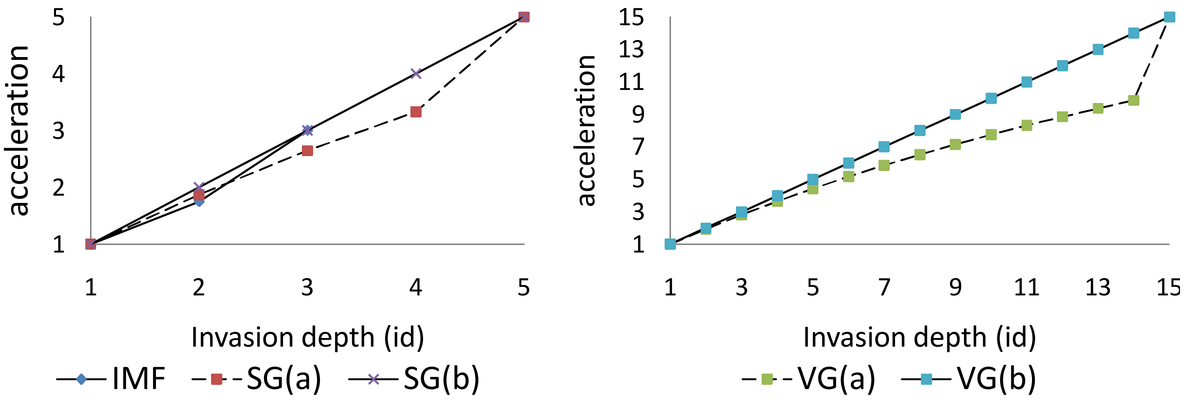


Figure 6.31.: Invasion Depth Vs Acceleration (IMF: image filtering, SG: signature generation, VG: vector generation)

Table 6.3 makes a comparison between the optical flow implementations on various architectures including an FPGA implementation at 100MHz [117], Ambric MPPA at 300 MHz processing QVGA frames at 37 fps [52] and a pure software implementation on Intel Core 2 Duo processor running at 1.86GHz [117]. The FPGA implementation is capable of processing one complete window per clock cycle, for various stages from the image filtering to the vector generation. Moreover the FPGA implementation has a pipelined structure for image filtering and signature generation and hence it does not consume any extra clock cycles for the image filtering.

Platform	execution time	image resolution	operating frequency
FPGA	9.87 ms	640×480	100 MHz
TCPA	13.72 ms	640×480	300 MHz
Ambric	27.03 ms	320×240	300 MHz
Intel Core 2 Duo	40.55 ms	640×480	1.86 GHz

Table 6.3.: Comparison between the optical flow implementation on different platforms

Figure 6.32 makes a comparison between the optical flow implementation on TCPA with a pure hardware implementation on FPGA [117] as well as a pure software implementation of an Intel Core 2 Duo processor (P8700, 32 bit) using *clocks*/*pixel* as an index for comparison. The *CPP* values are plotted on the y-axis on an exponential scale. The image filtering stage is pipelined with signature generation and hence does not take any extra clocks. For this reason the *CPP* for image filtering is not show in the graph.

Even though the FPGA implementation is designed for a *CPP* value equal to one, the system operates at an actual *CPP* value above one, due to the memory bandwidth bottleneck created by the external memory controller. Using a similar approach, the TCPA execution time will not be constrained by the external memory controller, because the best case *CPP* (when $id = w$) values from TCPA is slightly above the corresponding *CPP* values from FPGA

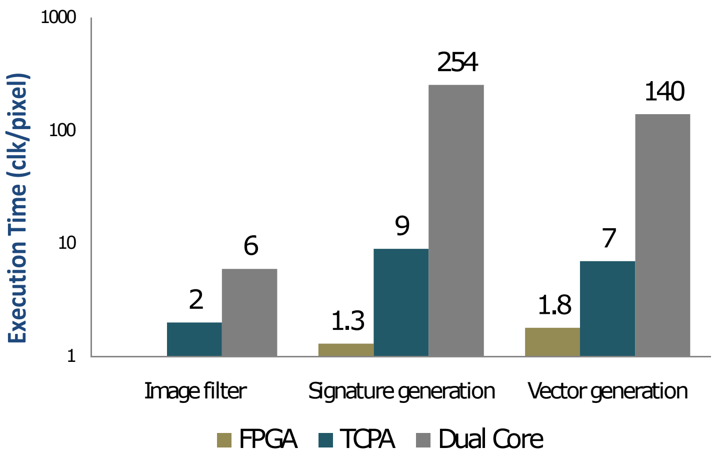


Figure 6.32.: A comparison between different architectures

and therefore the bottleneck will not be memory controller any more. The execution time of our implementation running at 300 MHz (with $w = id$) is close to the FPGA implementation and better than the Ambric implementation. The Ambric implementation used about 200 PEs for processing frames with 320×240 pixels while our implementation used about 225 PEs for processing 640×480 pixels, with a search window of size $[15 \times 15]$.

Equation (6.5) indicates the variation in CPP value for the vector generation stage of the optical flow algorithm. As explained in Section 6.4.4.3, a wider search region or window size is needed for detecting fast moving objects. But an increase in window size leads to an exponential increase in the number of pixels within the window. Such an increase in computing requirements can be met by invading more PEs, keeping processing time almost the same. This result proves that through Invasive Computing, time critical applications can attain more computing power and constrain their processing time effectively to meet their real-time requirements. Moreover additional resources were acquired by the application only when parallelism exists.

For a complete high speed implementation with a search region of $[15 \times 15]$ and window size equal to invasion depth, the complete execution involves mapping of 259 threads across the processor array. All these threads were mapped to neighboring PEs by the application running on the seed PE and hence eliminates the need for an external thread scheduler. Also such schedulers if used, cannot completely understand the requirements of the application, leading to inefficient resource utilization or high idle times for the PEs.

The performance figures also indicate that the overhead from dynamic resource allocation (C_{conf}) is negligible compared to overall execution time. This is because the TCPA architecture supports resource exploration and dynamic reconfiguration in hardware and any MPPA with similar features incorporated into its architecture can archive similar flexibility and resource awareness of invasive computing.

Next, we compare the execution time of our approach with the FPGA-based implementation as well as a pure software implementation on a soft RISC processor (LEON3), Table 6.5 shows the results of this performance comparison as well as hardware cost comparison.

	Image Filter	Signature Generation	Vector Generation
Configuration size (bits)	1 344	3 360	3 232
Configuration latency (ns)	840	2 100	2 020

Table 6.4.: Configuration bit stream sizes (bits) for the three parts of the optical flow as well as the configuration latency C_{conf}, where a 32-bit bus is used to load each configuration to the PEs (50 MHz clock frequency).

Architecture	Hardware cost			Performance
	slice regs	LUTs	BRAMs	(s)
FPGA	26 114	32 683	81	0.048
5×5 TCPA	35 707	105 799	85	0.427
LEON3	28 355	42 210	65	7.589

Table 6.5.: Hardware cost and performance comparison (time in seconds for fully processing one image frame) for the optical flow application implementation on a LEON3, 5×5 TCPA, and a custom FPGA implementation. All architectures operate at 50 MHz clock frequency and have been prototyped on a Xilinx Virtex-5 FPGA [14]

The hardware costs are reported for a Xilinx Virtex5 synthesis and the performance is reported in terms of the overall application computation time per frame. For the TCPA architecture, the configuration used consisted of a 5×5 array in order to show the achievable performance of TCPAs, having a hardware cost close to one LEON3. In order to exploit the full processing capacity of the TCPA, each part of the application is assumed to allocate the whole array. This is enabled by fast coarse-grained reconfiguration capabilities of TCPAs, where context switches of the array are performed at the nanosecond scale (see Table 6.4).

Although using more resources than the FPGA-based implementation and consequently achieving lower performance, TCPA architectures have the great advantage to be programmable and able to adapt itself regarding to different quality/performance requirements. The LEON3 is configured with 4 KB on-chip RAM, 32 KB data cache, and 16 KB instruction cache. The evaluation considered only the pure execution time of the application by assuming the input data already being loaded into the data cache of the LEON3 and input buffers of the TCPA, respectively. Therefore, the timing overhead of the communication between peripherals, bus, and the memory access are ignored in each case. In terms of execution time, the LEON3 needs 1 244 clock cycles to generate one signature and to compute a match. Consequently, it takes approximately 7.6 seconds to process one VGA frame, when running at 50 MHz. This implementation would therefore provide no more than 0.13 frames per second (fps).

At the same clock frequency, the 5 × 5 TCPA configuration achieves approximately 2.35 fps. However, the size of the processor array could be further increased to increase the performance. The total area needed to implement the 5×5 TCPA, i. e., 25 processing elements is practically equal to the area requirements of a single LEON3. But, the overall execution time achieved by our solution outperforms that of the LEON3 by a factor of 18, and even faster than the Ambric MPPA implementation [52], where in case of Ambric the system operates at 300 MHz frequency and processes 320×240 images and obtains 37 fps performance. With the same system setup, the TCPA would reach a frame rate of 55.2 fps.

6.5. Conclusion

This chapter focused on evaluating the benefits of specialized PEs like *i*-Core and TCPA, available on the InvasIC architecture. Various algorithms like Harris Corner detection, SIFT feature matching, optical flow, were mapped to different types of PEs. The performance benefits achieved from the use of heterogeneous PEs were studied and the results were presented in Sections 6.2.3, 6.3.3 and 6.4.5.

The application mapping schemes are highly flexible and the processing time can be manipulated by varying the amount of acquired resources. We believe that the concepts developed during this work is not just limited to optical flow, Harris Corner detection and SIFT feature matching; but in general can be applied to a wide range of image processing applications. The novel TCPA implementation could bring down the execution time for optical flow algorithm [14], close to pure hardware implementations and far below the execution time of similar MPPA implementation, with algorithm statically mapped to PEs. Also it should be noted that the TCPA is a processor array and can be programmed using a high level language. This means that TCPA offers great deal of flexibility in design and would help to reduce the design, implementation and verification time compared to FPGA implementations.

The resource-aware implementation acquired resources based on the computing requirements and this helped to reduce the chance for stalling of PEs which was the major bottleneck with the implementation in [52]. In our implementation, the memory bandwidth is kept to the lowest possible figures, by reading no pixels more than once from external memory. This is critical for any data intensive algorithm especially in image processing and this enable our design to operate with low speed memory controllers as in [117].

The SIFT feature matching using special instructions on *i*-Core could achieve significant speedup when compared to the conventional general purpose LEON3 PE. The absence of the DMA controller (for data transfers within tile boundary) has brought in some performance issues which can be fixed by adding an intra-tile DMA feature into the network adapter. Moreover the PEs were released immediately after a parallel execution, thereby enabling other applications to use them whenever required. This work also showed that time critical computer vision algorithms benefit from specialized PEs with special instructions and stream processors like TCPA. Moreover, the applications like optical flow also benefit from self-organizing and resource-aware programming which can be achieved using the new paradigm called Invasive Computing. The combined system (with LEON3, *i*-Core and TCPA) offers great prospects in future, especially in the areas of computer vision.

7. Object Recognition on Heterogeneous MPSoC

Conventional multi-core processors with identical cores are becoming inefficient due to their large area (from complex processor pipeline, caches, etc.), relatively high power consumption, etc. On the other hand, simple micro-architectures offers smaller area and better power efficiency. However, such designs offer poor single thread performance. Therefore, the heterogeneous systems achieve best of both worlds. The paradigm shift from multicore processors (with few identical cores) towards heterogeneous MPSoCs (with 100s of cores with different characteristics) bring in benefits and new challenges as described in Sections 1.4 and 1.5.

In the past, the CPU would have to perform a calculation, save it to memory, and then tell the GPU that the data was ready to be used. At thsi point, the GPU would have to copy the data from main system RAM into its own memory, perform some calculation, and output it to the display. With HSA 2.0 and HUMA, available on the APU, this process is massively simplified. There is only one pool of RAM, and the CPU can just pass a memory pointer directly to the GPU, allowing the GPU to very quickly pick up where the CPU left off. The on-chip interconnect allows to transfer data at a much faster rate between CPU and GPU blocks and therefore reducing the delay experienced on a conventional system where the CPU and GPU are interconnected using a PCIe. The thread scheduling across different types of PEs ((heterogeneous queuing on an AMD APU) is shown in Figure 7.1. Software can decide to use the GPU instead of the CPU, without worrying about additional complexity, latency, and so on.

Even thought the heterogeneous architecture like AMD APU are relatively new, the approach towards application mapping and thread scheduling follows a conventional strategy. Each application program is examined (profiled) to identify the parts of the application which can be accelerated using a GPU or a specialized PE. Once the parts for acceleration are identified, the application program is restructures in a way that the GPU friendly parts are mapped to the GPU. The high level control still remains under the CPU and they generate workload which gets scheduled on the GPU units. Even though this approach results in significant acceleration for the application program, such a static mapping of application parts to different PE types may lead to considerable performance loss and load imbalances on the heterogeneous MPSoC.

This chapter try to address such challenges, using the resource-aware programming model of Invasive Computing, for mapping the application program to various types of PEs available on the heterogeneous MPSoC called InvasIC, based on the instantaneous load on the

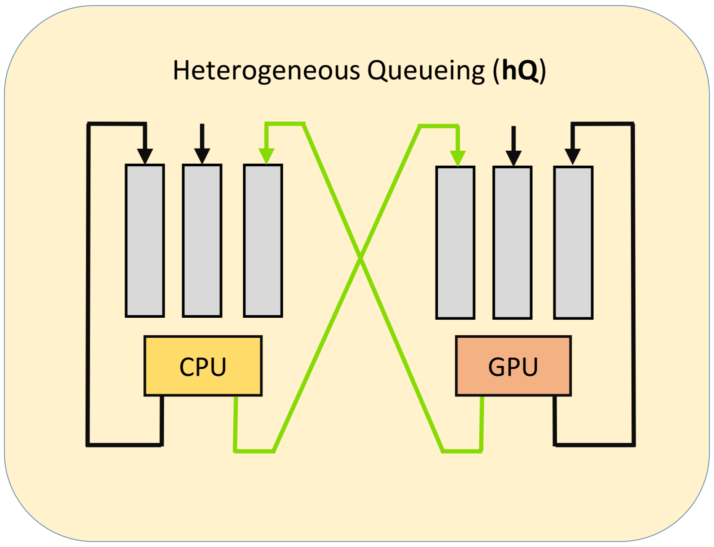


Figure 7.1.: Heterogeneous queuing (hQ) on AMD APU

PEs. All evaluations were performed of a FPGA prototype of the heterogeneous MPSoC to ensure the validity of the results.

7.1. Hardware Prototype

This section presents the prototype used for evaluation, as shown in Figure 7.2. Different configurations of the heterogeneous MPSoC can be modeled on the CHIPit system [103], a multi-FPGA platform from Synopsys. The prototype of the InvasIC design used for evaluations used a clock frequency of 25 MHz. The setup used consisted of a camera, monitor, host-PC, the CHIPit system, etc., were connected as shown in Figure 7.2.

Input frames captured by the camera (four frames per second, 640×480 pixels per frame) are stored in an input frame buffer within the host-PC that is configured to hold one frame at a time. The input images and final results are transferred in and out of the InvasIC prototype (on CHIPit) using an Ethernet interface. All three stages of the object-recognition algorithm run on the FPGA prototype of the heterogeneous MPSoC. The input frames are read (from the frame buffer) by the object-recognition, whenever it is ready to accept new input data. Upon completion of Harris corner detection, the results are passed to the next stage, and application waits for the next frame. All three stages within the object recognition were implemented in a pipelined manner and a new image frame will be available every 250 milliseconds. The results are returned to the host PC for comparison and visualization on the monitor.

The heterogeneous variant of the InvasIC architecture under evaluation consists of six tiles (five compute tiles): Tile-0 consists of three LEON3 PEs and one *i*-Core, tile-1 is a TCPA tile and tiles-2, -3 and -5 consist of four LEON3 PEs. Tile-4 is an I/O tile which is used to transfer the audio/video data in and out of the FPGA platform. The homogeneous variant

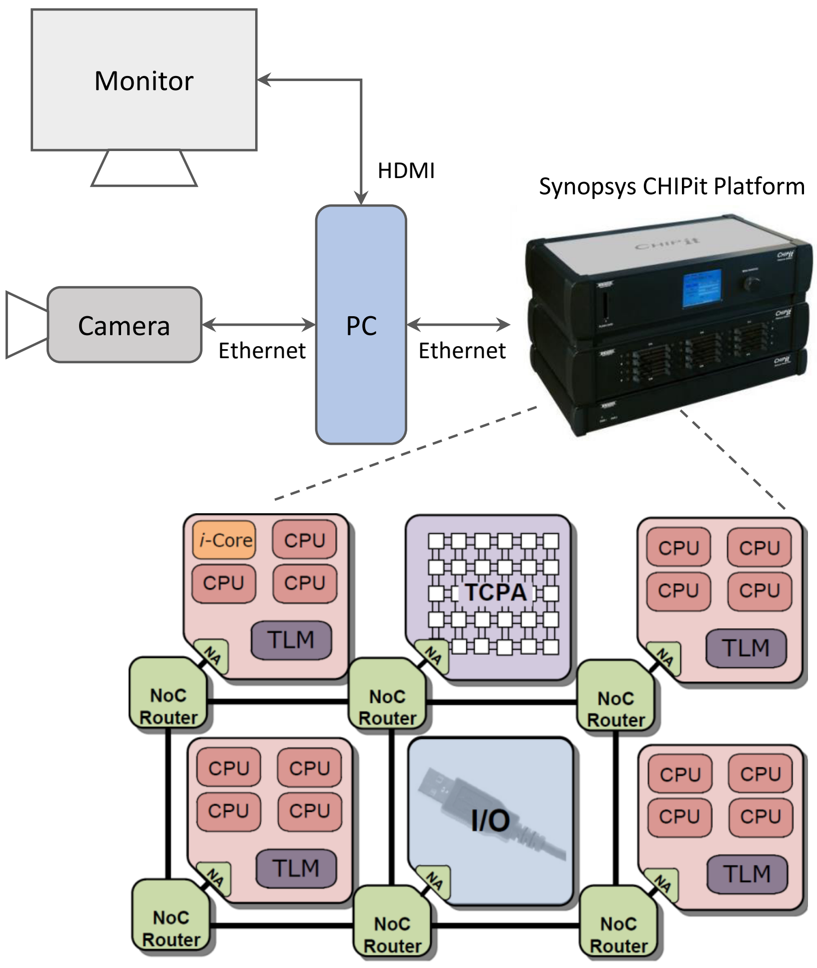


Figure 7.2.: Setup for evaluation [8]

used for comparison, has a similar configuration but with the TCPA and *i*-Core replaced by conventional LEON3 PEs.

7.2. Evaluation Models

The entire evaluation process is divided into three different steps. The first step demonstrates how the threads from various stages of the application program are scheduled on to different types of PEs available on the heterogeneous MPSoC. It also compares the heterogeneous version of the object recognition with its homogeneous implementation (with all three stages of the algorithm mapped to LEON3 PEs), in terms of flexibility, performance obtained, etc. The next step analyzes the performance degradations in a dynamic scenario where the PEs are shared by multiple applications running concurrently. Together with the object recognition, audio filtering and matrix multiplication were used as additional applications competing with each other for various resources. Finally, step three, demonstrates how the resource-aware programming paradigm of Invasive Computing can improve the performance of the application program by adapting itself to the available resources on a heterogeneous MPSoC, at runtime.

The results from different evaluations are compared in terms of throughput (frames processed per second) and worst observed latency (WOL), where latency is defined as the time elapsed from the task dispatch until the availability of results. The WOL values presented in the results section is based on the evaluation model described above and it may vary based on the load generated by other applications, application scheduling patterns, etc.

7.2.1. Homogeneous vs. Heterogeneous Implementation

As described in Section 4.1, the object recognition algorithm consists of three main stages.

1. Harris Corner detection
2. SIFT feature extraction
3. SIFT feature matching

Figure 7.3 shows the execution-time profile and scheduling scheme on the heterogeneous MPSoC for all three stages of the object recognition. The algorithm starts with the Harris Corner detection on TCPA. The implementation used here is similar to the one described in Section 6.2, with the difference that a 2-PE model was used instead of the 4-PE model.

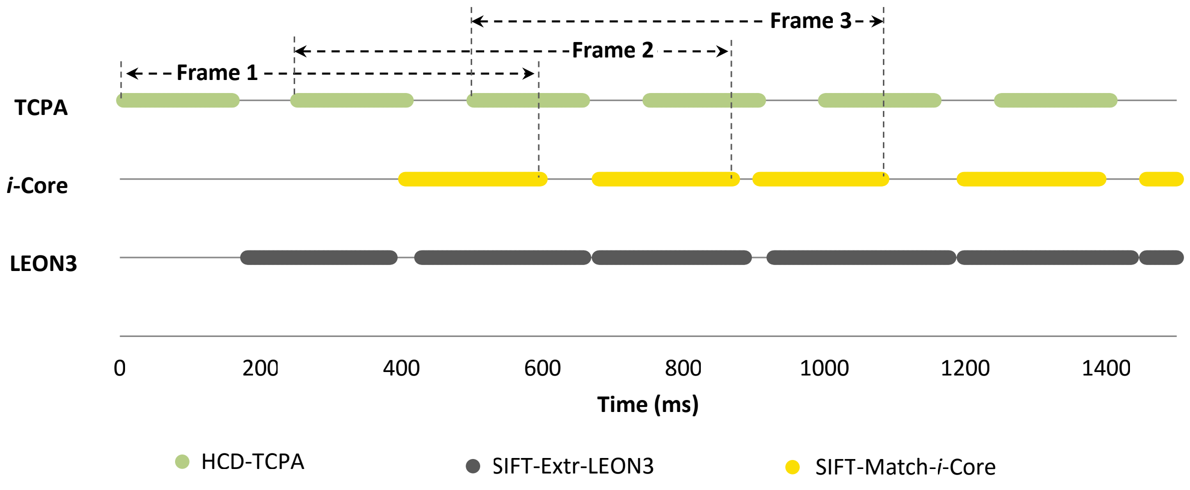


Figure 7.3.: Object recognition on heterogeneous MPSoC [8]

The Harris Corner detection is followed by the SIFT feature extraction. The algorithm for SIFT feature extraction has to operate at different feature points detected by Harris Corner detection algorithm. The distribution of the feature points typically depends on the nature of the input image and the location of various object within the frame. Hence, the algorithm incorporates rather thread-level than loop-level parallelism as exploitable on a TCPA. The SIFT feature extraction can be implemented on *i*-Core with special instructions. However, this would result in multiple special instructions as the algorithm consists of several different stages. Hence, the algorithm was mapped to the the loosely coupled LEON3 PEs, which can execute the conventional algorithm from the Integrated Vision Toolkit library [111], in a multi-threaded environment. Finally the SIFT feature matching is performed on the *i*-Core, using the special instructions described in Section 6.3.

The homogeneous implementation, on the other hand, has all three stages of the object recognition algorithm mapped to the general purpose LEON3 PEs. In order to keep the execution-time profile comparable to the heterogeneous model, the Harris Corner detection algorithm was mapped to four LEON3 PEs. The SIFT feature extraction uses four LEON3 PEs in both homogeneous and heterogeneous models. Finally the SIFT feature matching uses seven LEON3 PEs in order to achieve the same performance as a single *i*-Core. The execution-time profile for the homogeneous variant is shown in Figure 7.4.

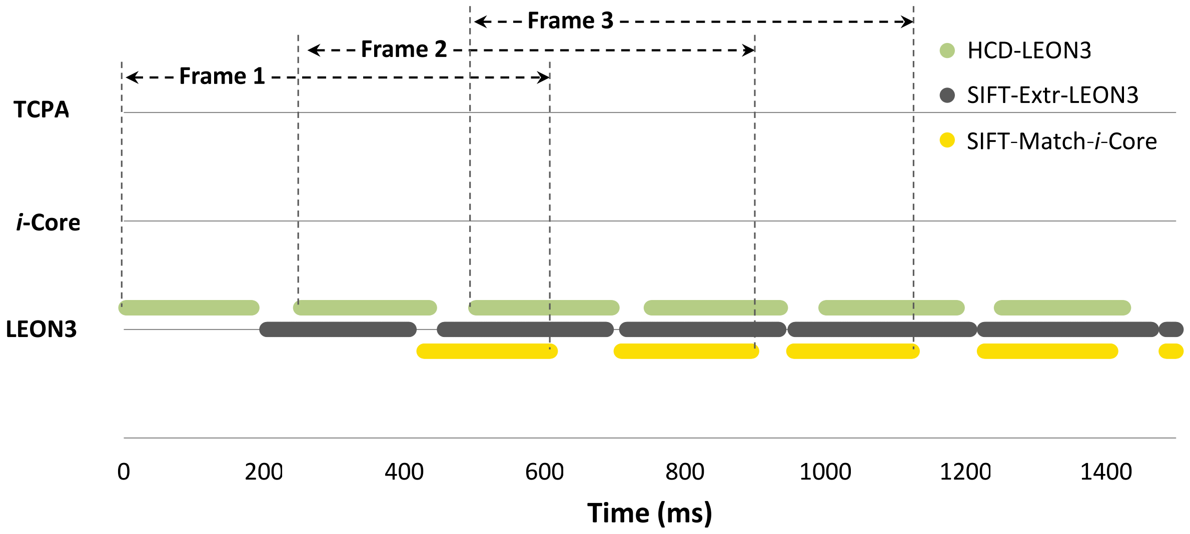


Figure 7.4.: Object recognition on homogeneous MPSoC (LEON3 PEs) [8]

The entire evaluation process consists of 100 frames and takes about 25 seconds on the FPGA prototype (with a 25MHz clock). As it is difficult to show the scheduling pattern for all 100 frames, the overall results are presented in table format (see Table 7.1). Remapping the Harris corner and SIFT feature matching to TCPA and *i*-Core helped to reduce the overall load on the LEON3 PEs.

	Load on LEON3	Load on TCPA	Load on *i*-Core	Throughput	WOL (msec)
Homogeneous	59.16%	0.0%	0.0%	97 frames	732
Heterogeneous	23.03%	62.17%	72.41 %	97 frames	683

Table 7.1.: Object recognition: homogeneous vs. heterogeneous [8]

7.2.2. Multiple Applications on Heterogeneous MPSoC

This section demonstrates the challenges and performance issues encountered by application programs on a conventional heterogeneous platform. In order to evaluate a situation where multiple applications compete for resources, additional applications such as audio filtering and matrix multiplication (an application that is widely used in computer graphics) were used. The audio-filtering application is capable of processing thousands of audio samples by filtering the samples to eliminate unwanted noise signals. As audio filtering belongs to the class of streaming applications, it can be performed efficiently on the TCPA. This application is scheduled on the TCPA at regular intervals of 1000 milliseconds and each run takes approximately 500 milliseconds to process all audio samples assigned for that run. Similarly, matrix multiplication is used to inject additional load on the *i*-Core. This application is capable of multiplying large 2D matrices, scheduled at regular intervals of 3000 milliseconds and each run takes approximately 440 milliseconds.

The above mentioned applications (audio processing and matrix multiplication) were allowed to run concurrently with the object-recognition application. The available resources were shared between these applications. All applications maintain equal priority and were scheduled on a first-come-fist-served basis on the TCPA, *i*-Core and LEON3 PEs. As a result, under some circumstances the object recognition application has to wait until the audio-processing application finishes its current task on the TCPA or the matrix multiplication has completed execution on the *i*-Core. This is depicted in Figure 7.5, where the label (a) indicates the expected time for Harris Corner detection (frame-2) could begin on TCPA. However, this was delayed until label (b) as the TCPA was loaded with audio-processing tasks. A similar situation occurred between labels (c) and (d), where the SIFT matching was delayed as the *i*-Core was loaded with matrix multiplication.

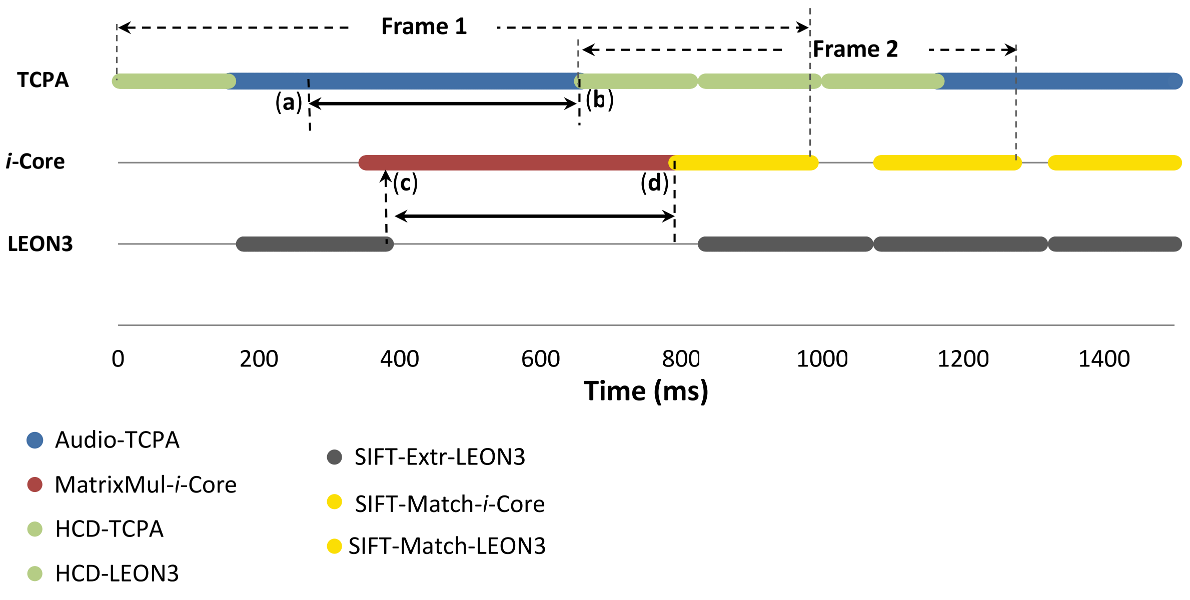


Figure 7.5.: Multiple applications executing concurrently on our heterogeneous MPSoC with conventional mapping scheme [8]

The results from this evaluation are provided in Table 7.2, which indicates a reduction in overall performance of the object-recognition application. The overall throughput (frame count) was reduced from 97 frames to 72 frames during the entire evaluation. The reduced throughput leads to a low load situation on the LEON3 PEs. Also, the prolonged wait time leads to a higher worst observed latency (WOL) as shown in Table 7.2.

	Load on LEON3	Load on TCPA	Load on *i*-Core	Throughput	WOL (msec)
Obj-Recog only	23.03%	62.17%	72.41 %	97 frames	683
All three apps	17.48%	95.77%	67.32%	72 frames	1400

Table 7.2.: Object recognition: single vs. shared, conventional mapping scheme [8]

7.2.3. Heterogeneous MPSoC with Resource-aware Mapping

This section demonstrates the benefits achieved by using the resource-aware programming model for the object-recognition application on the heterogeneous architecture. Among the three different stages forming the object recognition, the Harris Corner detection and the

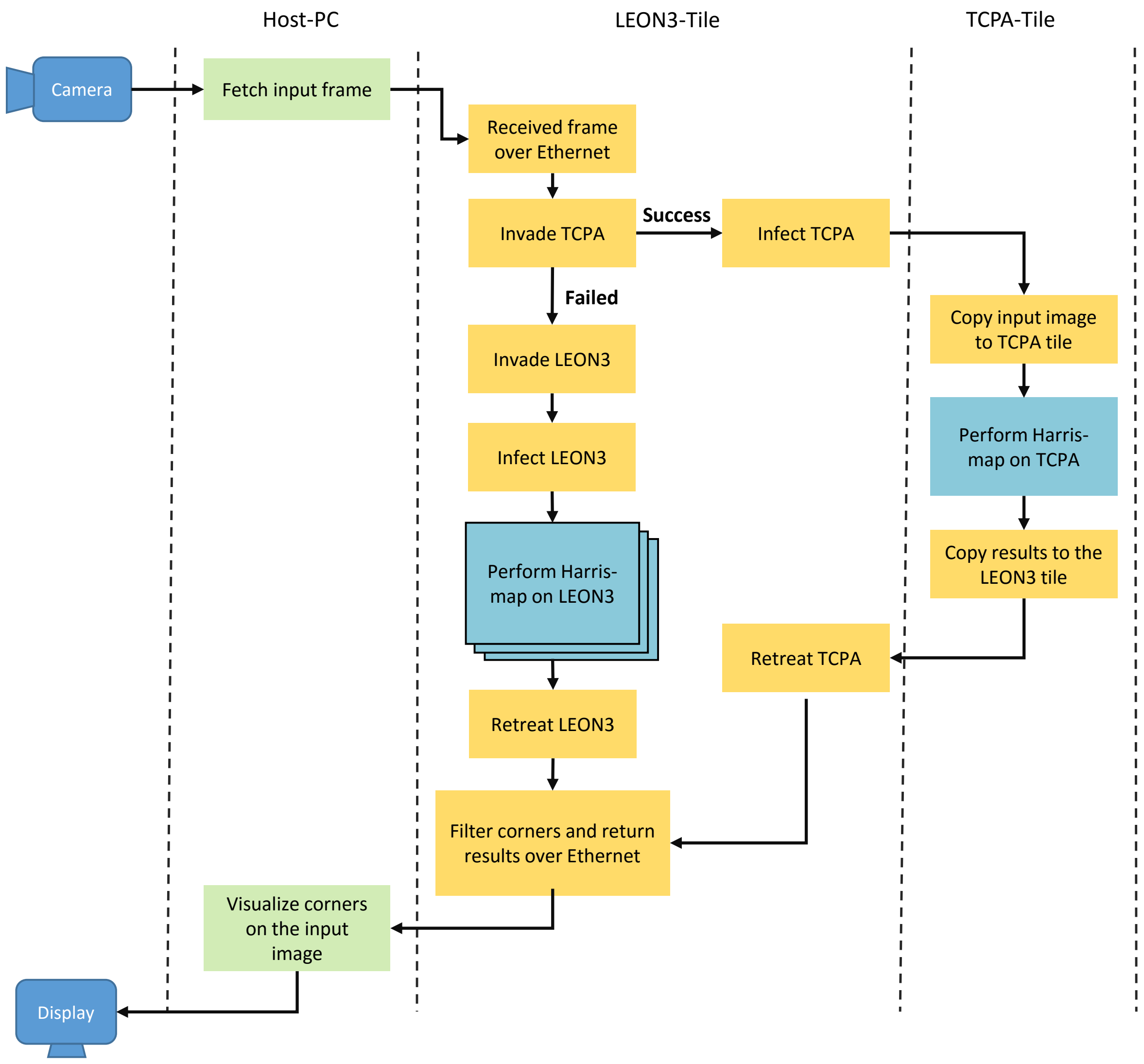

Host-PC
LEON3-Tile
TCPA-Tile
Camera
Fetch input frame
Received frame over Ethernet
Invade TCPA
Success
Infect TCPA
Failed
Copy input image to TCPA tile
Invade LEON3
Perform Harris-map on TCPA
Infect LEON3
Copy results to the LEON3 tile
Perform Harris-map on LEON3
Retreat TCPA
Retreat LEON3
Filter corners and return results over Ethernet
Visualize corners on the input image
Display

Figure 7.6.: Self adaptive Harris Corner detection algorithm running on InvasIC

SIFT feature matching were designed to run on different types of PEs, on InvasIC. The TCPA implementation (used for this evaluation) achieved an equivalent performance compared to an implementation of Harris Corner detection on four LEON3 PEs. However, under circumstances as demonstrated in Section 7.2.2, when the TCPA is occupied by the other applications like audio filtering, the resource-aware modeling enabled the object-recognition application to adapt itself and remap its workload to the freely available LEON3 PEs. The new flow diagram of the resource-aware Harris Corner detection algorithm is depicted in Figure 7.6.

At each interest point detected by the Harris Corner detection, a SIFT descriptor is computed. This algorithm was mapped to the the loosely coupled LEON3 PEs (as its not suitable for TCPA not *i*-Core), which can execute the conventional algorithm from the Integrated Vision Toolkit library [111], in a multi-threaded environment.

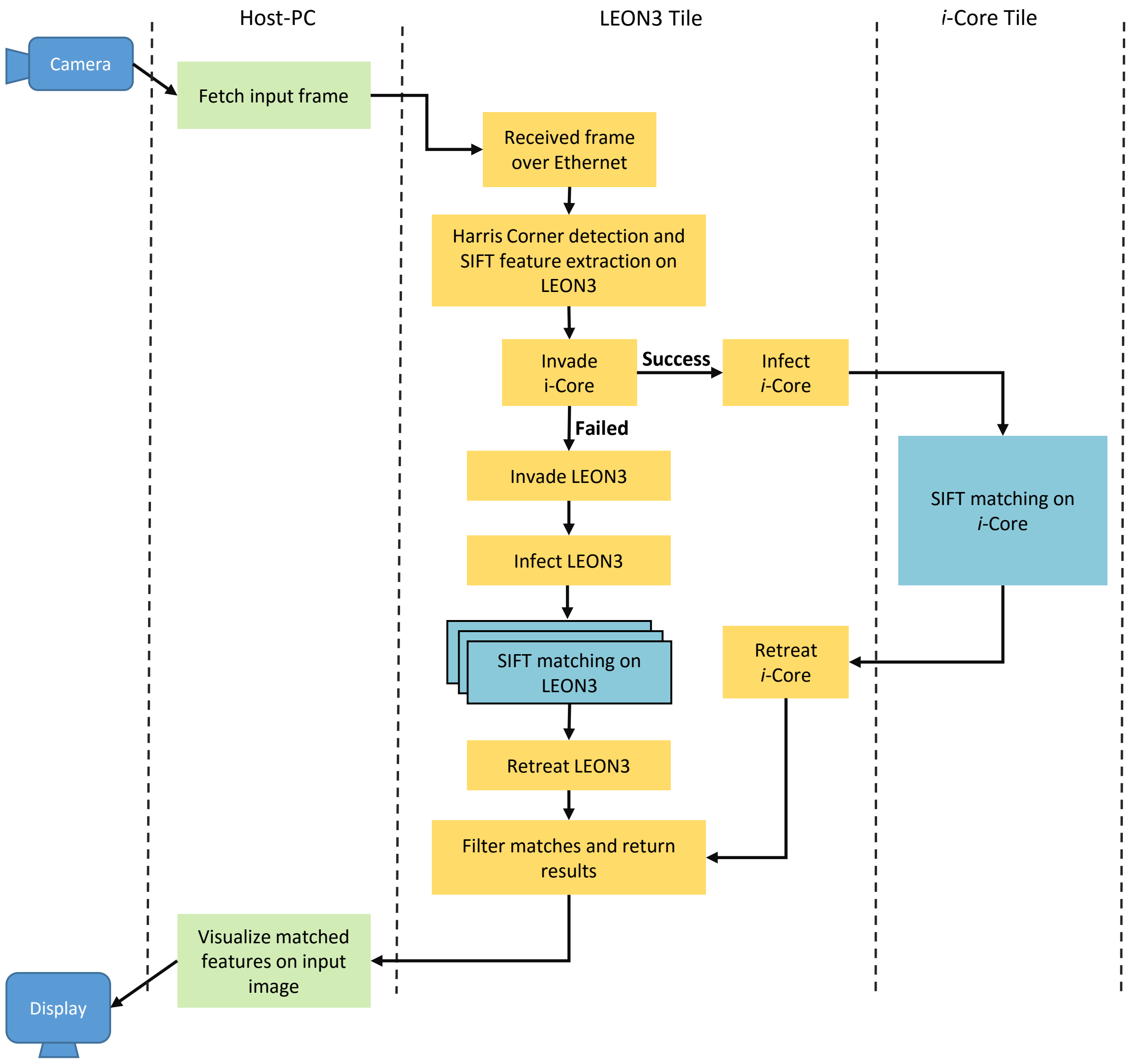

Host-PC
LEON3 Tile
i-Core Tile
Camera
Fetch input frame
Received frame over Ethernet
Harris Corner detection and SIFT feature extraction on LEON3
Invade i-Core
Success
Infect *i*-Core
Failed
Invade LEON3
Infect LEON3
SIFT matching on LEON3
SIFT matching on *i*-Core
Retreat *i*-Core
Retreat LEON3
Filter matches and return results
Visualize matched features on input image
Display

Figure 7.7.: Self adaptive SIFT feature matching algorithm running on InvasIC

Finally, the SIFT feature matching can be mapped on to an *i*-Core or to the conventional LEON3 PEs based on their instantaneous availability. The new flow diagram of the resource-aware SIFT feature matching is depicted in Figure 7.7.

The three individual algorithms as described above can be combined to form a resource-aware object recognition algorithm with the resulting flow diagram as shown in Figure 7.8. The corresponding execution time profile and scheduling pattern are shown in Figure 7.9.

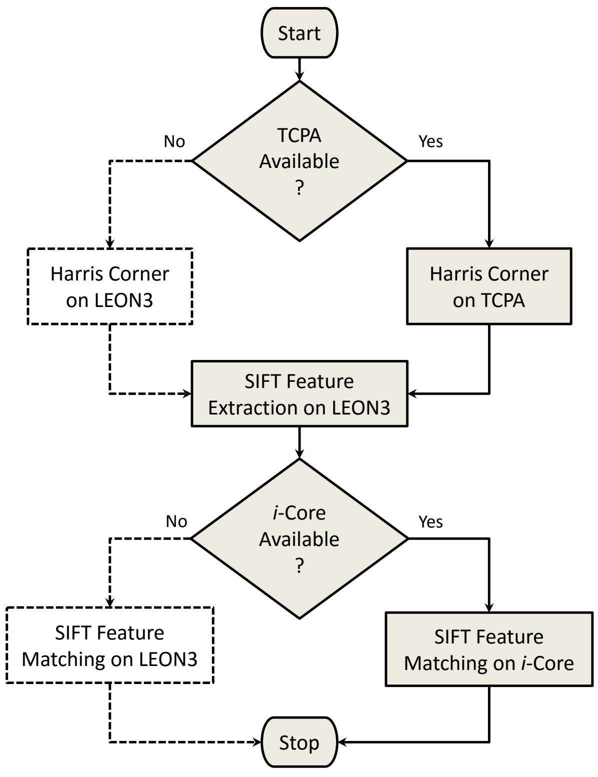


Figure 7.8.: Flow diagram for resource-aware object recognition [8]

Labels (e) and (f) in Figure 7.9 indicate two different instances where the Harris corner detection could not be performed on TCPA as the TCPA was occupied by the audio-processing application. The resource-aware modeling enabled the application program to remap the Harris-corner algorithm on to the available LEON3 PEs. Subsequently, the Harris Corner detection algorithm for frame-4 was scheduled on the TCPA, as indicated by label (g). A similar situation for the SIFT feature matching is indicated by labels (h) and (x), where the *i*-Core was occupied by the matrix multiplication. The SIFT feature-matching algorithm could recover its execution on *i*-Core from frame-3, indicated by label (y). In contrast to the scheme in Section 7.2.2, the application adapts itself at runtime to use the available PEs.

Table 7.3 compares the conventional model with the resource-aware model in terms of throughput, WOL and resource utilization. It can be seen that the throughput has increased to 98 frames, almost equivalent to the scenario in Section 7.2.1, where the entire hardware was dedicated for the object-recognition algorithm. The results prove that the WOL (reduced to 705 milliseconds) can be regulated. The higher throughput and dynamic adaptations lead to a higher load on the LEON3 PEs. There is a marginal decrease in the load on the

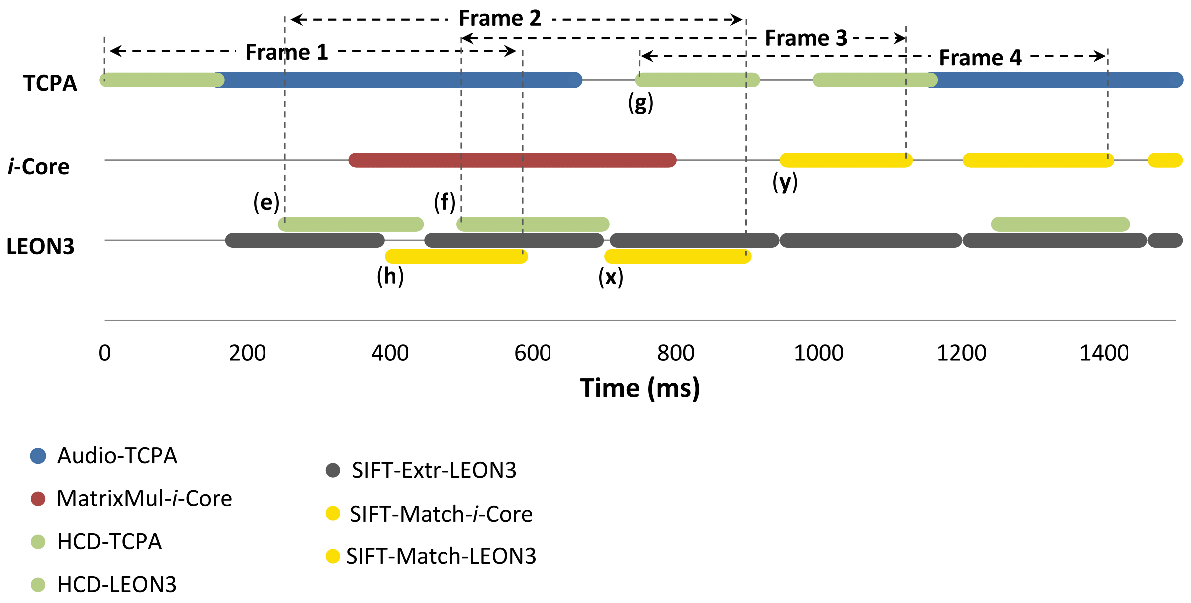


Figure 7.9.: Multiple applications on heterogeneous MPSoC using resource-aware mapping [8]

TCPA, as the Harris corner algorithm was often remapped to the LEON3 PEs. The load on the *i*-Core increased slightly, as the SIFT feature-matching algorithm often managed to run on the *i*-Core within the long intervals (lasting about 2560 milliseconds), where the matrix-multiplication application remains idle. The results in Table 7.3 show that the resource-aware programming model enabled the object-recognition application to improve its performance by adapting itself to the available resources at runtime.

	Load on LEON3	Load on TCPA	Load on *i*-Core	Throughput	WOL (msec)
Conventional	17.48%	95.77%	67.32%	72 frames	1400
Resource-aware	38.26%	79.93%	74.79%	98 frames	705

Table 7.3.: Object recognition: conventional vs. resource-aware [8]

7.3. Conclusion

In chapter focused on exploring the benefits of the *Invasive Computing* methodology, on a heterogeneous platform InvasIC. The flexibly configurable tiled architecture used for the case study comprised several SPARC-V8 LEON3 cores, massively parallel processor arrays (TCPAs) and processors with configurable instruction sets (*i*-Cores). The case study was based on the widely used object-recognition algorithm. Various stages within the algorithm were analyzed and mapped to different types of PEs based on their computational requirements.

However, the presence of multiple processing elements with different characteristics raises issues related to programming and application mapping [8].

The case study using the various statges within the object-recognition algorithm showed that the applications may suffer performance loss and a higher worst observed latency (WOL) when the PEs on the heterogeneous processor are shared between different applications. As a remedy,the resource-aware programming paradigm called *Invasive Computing* enabled the application program to adapt itself to the available resources at runtime. Such adaptations were shown to help the object-recognition application to improve its throughput by 36% and its WOL to be reduced from a factor of 2.05x to 1.03x (close to the ideal value of 1.0x). The overhead from resource allocation and release was kept very low by employing a specialized operating system called OctoPOS and through special hardware support [8].

8. Conclusion and Outlook

Multi-Processor System-on-chip (MPSoC) designs are gaining increasing interest in the field of embedded computing and robotics, due to their enhanced performance capabilities. Significant reduction in processing time, for computer vision applications, could be achieved using homogeneous MPSoC architectures like Tilera [48], Intel's Single-chip Cloud Computer (SCC) [50] or heterogeneous architectures like AMD's Fusion APU [5]. Experiments conducted on the AMD's APU shows that addition of massively parallel processor arrays (MPPA) or GPU units can lead to significant improvement in overall performance. Despite the emergence of new heterogeneous architecture, the approach towards application mapping and thread scheduling follows a conventional strategy, where each application program is examined (profiled) to identify the parts of the application which can be accelerated using specialized PEs. Such a static implementation is not flexible enough to handle dynamically changing requirements of robotic applications.

In this context, the novel programming methodology based on Invasive Computing was introduced which enabled the application program with the power to manage and coordinate various processing resources by themselves, in a decentralized manner. This approach enabled various applications to self-explore the degree of parallelism available and to exploit dynamic resource requirements while avoiding fully centralized control of execution.

This research work mainly investigated the principles of Invasive Computing for computer vision algorithms used on the humanoid robot ARMAR [1], which has to deal with various tasks like stereo vision, object recognition, object grasping, obstacle detection, autonomous navigation, etc. The scientific contribution of this thesis, as described in Chapters 5, 6 and 7, can be summarized as follows:

Chapter 5 presented the challenges faced by image processing algorithms like Harris Corner detection and Nearest-neighbor search on kd-trees, on a shared homogeneous MPSoC. Using real-world experiments, this thesis demonstrated an accuracy loss on a conventional MPSoC hardware, due to resource sharing. As a way to overcome these challenges, this chapter presented novel resource-aware algorithms for Harris Corner detection and NN-search on kd-trees which can estimate the resources required for a specific task based on the scene (nature of the objects, number of objects present, type of background, the texture of the object to be recognized, etc.). The experiments showed that incorporating resource-awareness into the conventional Harris Corner detection and NN-search algorithm can improve the quality of the object recognition process with up to 22 percent improvement in throughput and up to 20 percent improvement in accuracy.

Chapter 6 focused on evaluating the benefits of specialized PEs like *i*-Core and TCPA, available on the InvasIC architecture. Various algorithms including Harris Corner detection, SIFT feature matching and optical flow, were implemented on *i*-Core and TCPA. The performance benefits achieved from the use of heterogeneous PEs were studied and the results were presented. The SIFT feature matching using special instructions on *i*-Core achieved significant speedup when compared to the conventional general purpose LEON3 PE. Moreover, the PEs were released immediately after a parallel execution, thereby enabling other applications to use them whenever required. This work also showed that time critical computer vision algorithms benefit from specialized PEs with special instructions and stream processors like TCPA. The combined system (with LEON3, *i*-Core and TCPA) offers great prospects in future, especially in the areas of computer vision.

Chapter 7 demonstrated the challenges involved in programming and mapping vision application on a heterogeneous design with multiple processing elements with different characteristics. The case study showed that the applications may suffer performance loss and a higher worst observed latency (WOL) when the PEs on the heterogeneous processor are shared between different applications. Adapting to a different type of processing element based on the load conditions helped the object-recognition application to improve its throughput by 36% and its WOL to be reduced from a factor of 2.05x to 1.03x (close to the ideal value of 1.0x).

The work carried out in this thesis should be considered as the first significant step towards the migration of various applications mapped to multiple industrial CPU boards on conventional robot, on to a single-chip MPSoC design, without suffering quality loss. The unified processing system will make task migration and load balancing easier and also offers better communications bandwidth between PEs. Lat but not the least, the used of MPSoCs also help to reduce power consumption and occupy less space on the robot.

Suggestions on Future Work

As a future work, the runtime system could be extended so that the information provided by the application layer (resource requirements) can be used to improve the overall efficiency of the system. For instance, the runtime system may shut down PEs if they are not claimed by any application, and wake up PEs only when necessary. Additionally, the resource-aware model could be extended to more robotic applications, perform evaluations on the complete chain of algorithms, and also develop prediction models within the applications so as to foresee the resource requirements. Additional focus on techniques to efficiently allocate resources like communication bandwidth (over *i*-NoC), on-chip memory (TLM) and AHB Bus; among various applications running on the MPSoC may lead to better performance and resource utilization numbers. Moreover, the runtime system (OctoPOS) could be extended to allow the applications to allocate (temporal) fractions of a PE and to migrate running applications to different tiles if needed. Another aspect which can reduce the programming effort on MPSoCs like InvasIC is to offer an optional inter-tile cache-coherence mechanism where an application program, spread across multiple neighboring tiles, can request the underlying hardware to provide a temporary cache-coherence across the tiles.

Acknowledgement

This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre Invasive Computing (SFB/TR 89).

A. Hardware Monitoring Framework for InvasIC

In addition to the contribution on the vision algorithms, this thesis also contributed to the development of the hardware monitors (performance counters) used in InvasIC. The design, implementation and application of the hardware monitors is described in this section.

On a large MPSoC design like InvasIC, numerous factors influence the performance of an application program. This includes the size of data/instruction caches, floating point unit (FPU), bus interconnects, network-on-chip, on-chip, off-chip memories, etc. Various blocks deciding the performance (execution time) of a CPU instruction is shown in Figure A.1. Each

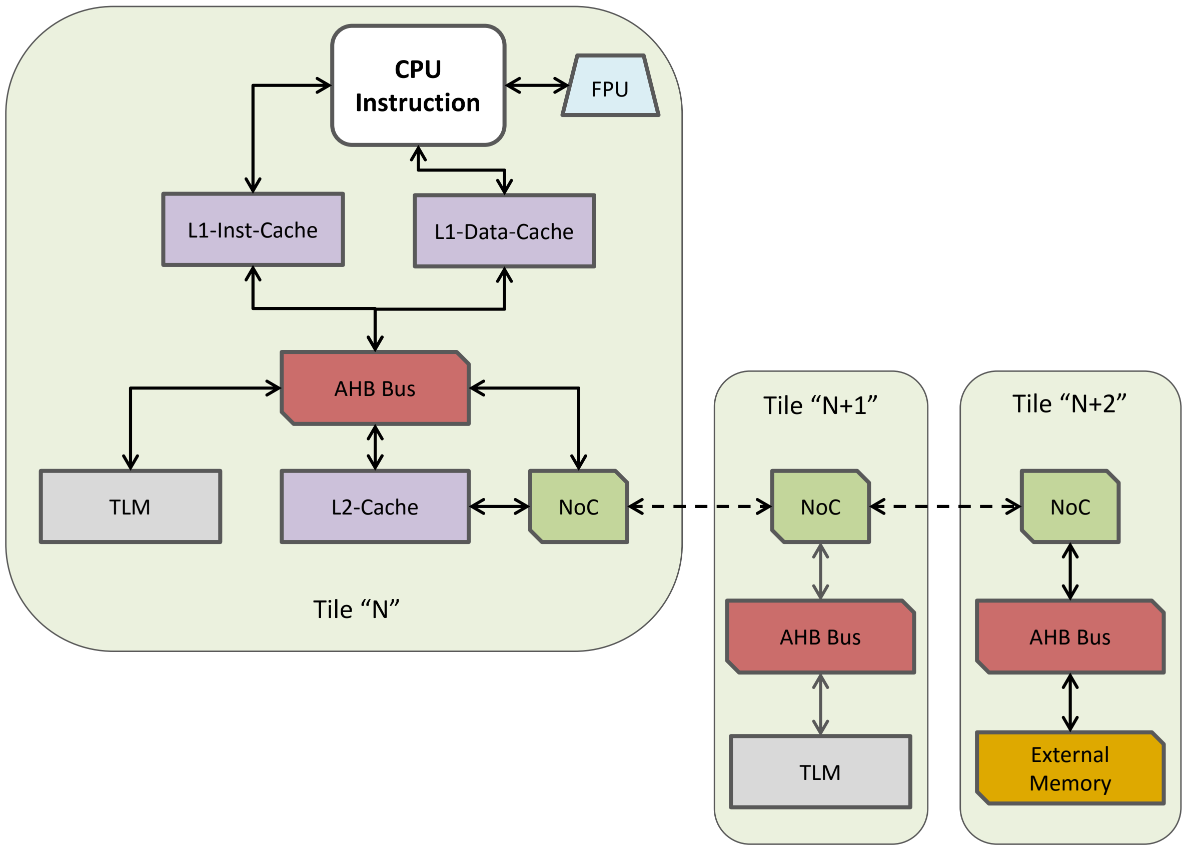


Figure A.1.: Various hardware blocks affecting the performance of a CPU-instruction in a multi-tile MPSoC

of these blocks are configurable from the design point of view, leading to several possible MPSoC configurations. In order to achieve a balance in the design and to avoid bottlenecks during the design phase of InvasIC, detailed evaluations using real-world applications were necessary. In order to facilitate the design process and to avoid bottlenecks, certain performance counter or hardware monitors were built into the design. The design and application of such hardware monitors are described in this appendix section.

A.1. Hardware Monitor Design

The hardware monitor design involves the design of three major blocks, as listed below:

- Performance counters to count various events within the MPSoC
- Interface to send the performance data to the host-PC in a non intrusive way
- Visualization modules to capture and display the monitoring data, gathered by performance counters

The techniques used in the design and implementation of each of the above blocks are described in the upcoming sections.

A.1.1. Performance Counters in Hardware

Performance counters are useful hardware mechanisms which can provide insight into the performance of various hardware blocks. Difference types of performance counter developed for InvasIC are listed below:

- **Processor monitors** to estimate the performance of the CPU during the execution of various different applications. The processor monitors include:
  - Instruction monitors to count the number of instructions executed by the CPU. This counter increments its value each time an instruction is executed by the CPU.
  - FPU monitors to measures the utilization rate of FPU unit, if present. The aim of the FPU monitor is to count the number of clock cycles, the FPU is busy and thus providing useful information to the programmer. This information could be used by the programmer to include an FPU as a constrain during the *invade* operation.
- **Cache/Memory monitors** to model the behavior of the cache memories and on-chip/off-chip memory blocks like TLM or DDR-II.
  - TLM utilization monitors to measure the usage of TLM. This is performed by counting the number of cycles bus is in non-idle state due to TLM accesses.
  - L1 and L2 utilization monitors to measure the cache-miss latency during program execution (number of cycles to retrieve the data from external memory) and thus provide profiling information to the programmer.

  AHB Bus monitor: Analyze the AMBA AHB bus load and estimate the bus bandwidth utilization. The job of the bus monitor is to calculate the number of non-idle cycles on the bus and generate bus utilization numbers. This is done by monitoring the type of transaction on the bus.

***i*-NoC monitors**: Monitors the *i*-NoC router parameters *i*-NoC monitor can track two parameters on each *i*-NoC router, the link utilization and the reserved channels.

  – Link utilization counts the number of cycles the *i*-NoC link is busy. The counter is incremented each clock cycle when the *i*-NoC link is busy (utilized).
  – Reserved channel monitor counts the average number of channels reserved during the measuring interval.

- In order to quickly assign numerous *i*-lets to various types of PEs on the MPSoC, the OctoPOS is assisted by a special hardware mechanism named **Core *i*-let Controller (C*i*C)**. The C*i*C can analyze the requirements specified my the application program and can assist and accelerate the *infect* process. The performance of the C*i*C can also be measured using the performance counters. Various counters used for this purpose includes:
  – The arrival rate of *i*-lets, in C*i*C.
  – The C*i*C's FIFO fill levels (C*i*C uses one FIFO per core to store the *i*-lets).
  – The dispatch rate of *i*-lets, by C*i*C.

Monitoring mechanism described above are completely implemented in hardware in a non-intrusive manner, so that the application programs running on InvasIC does not encounter a performance impact. Moreover, any application program can be profiled without inserting special instructions, in contrast to some of the conventional profiling methods. A monitor block with the performance counters described above, is built into each tile and they analyze the predefined set of signals from various PEs, as shown in Figure A.2.

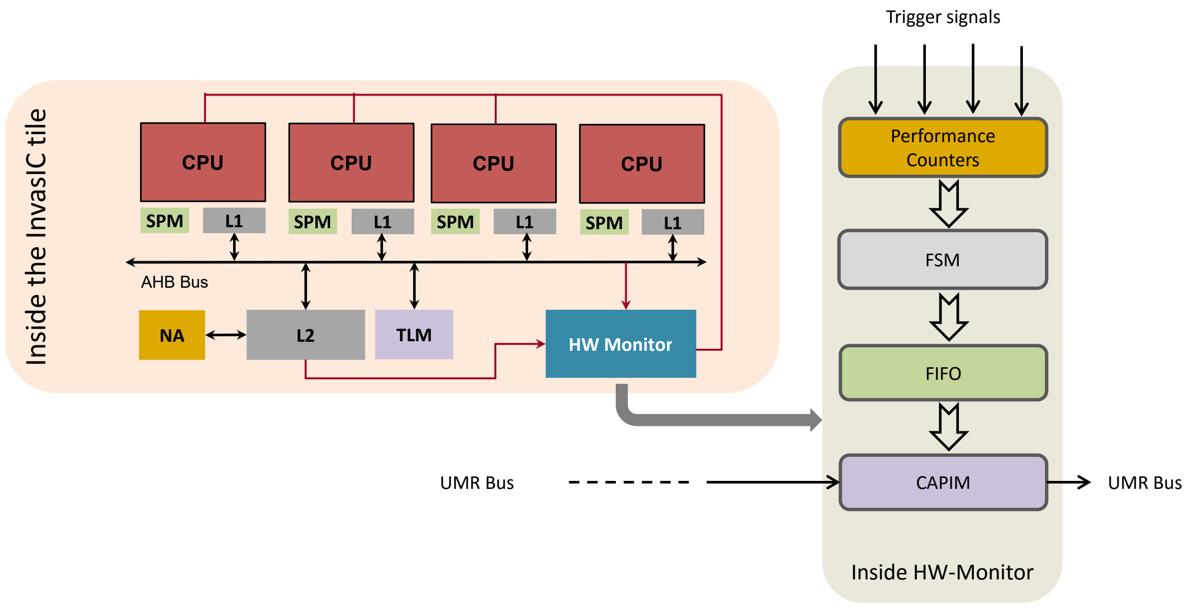


Figure A.2.: Implementation of the hardware monitor (inside each tile)

A.1.2. Interface and Data Exchange

The data captured by the monitors is organized into packets (marked by tile-ID, core-ID, etc.) and the information is then transmitted over the Synopsys Universal Multi Resource Bus (UMR Bus). The UMR Bus is a high-speed communication interface designed into the CHIPit system to enable workstation-based access to the FPGA-based prototype. The UMR Bus offers up to 400MB/s host connection sufficient for most applications. The interface between the monitor and the UMR Bus is based on the CAPIM module, as shown in Figure A.4.

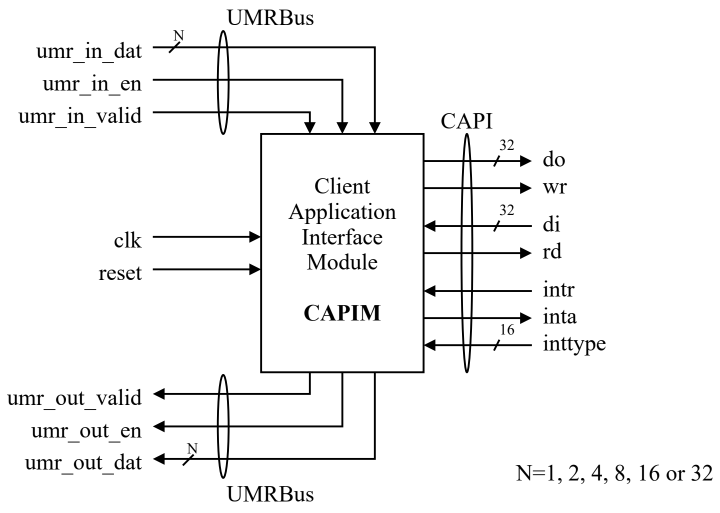


Figure A.3.: CAPIM module used to interface the HW monitors to the UMR bus

The CAPIM module offers a simple and easy to use interface to exchange data with the UMR Bus. The UMR Bus runs through every FPGA/tile in the system and collects the monitoring data. The collected data is then transferred to the host-PC using a dedicated cable provided as part of the CHIPit system.

A.1.3. Visualization

During the design of InvasIC, it was essential to analyze the performance of PEs, their load conditions, AHB Bus load, link utilization over the *i*-NoC, etc. In order to easily interpret the monitoring data, a real-time visualization of the monitoring data was necessary. Such a visualizer should be able to visualize the data from the sensors listed in Section A.1.1, in the form of line/bar graphs. As the amount of data capture from the MPSoC is of the order of Mbytes/sec, the visualization software has to be fast enough to handle it. A visualization system based on C++ QT framework, was developed for this purpose, with focus on:

- Techniques to visualize the large amount of data from numerous monitors, in a compact way.
- Necessary optimizations to enable real-time processing
- Features to log the monitor data for offline analysis.

The entire setup used is shown in Figure A.5.

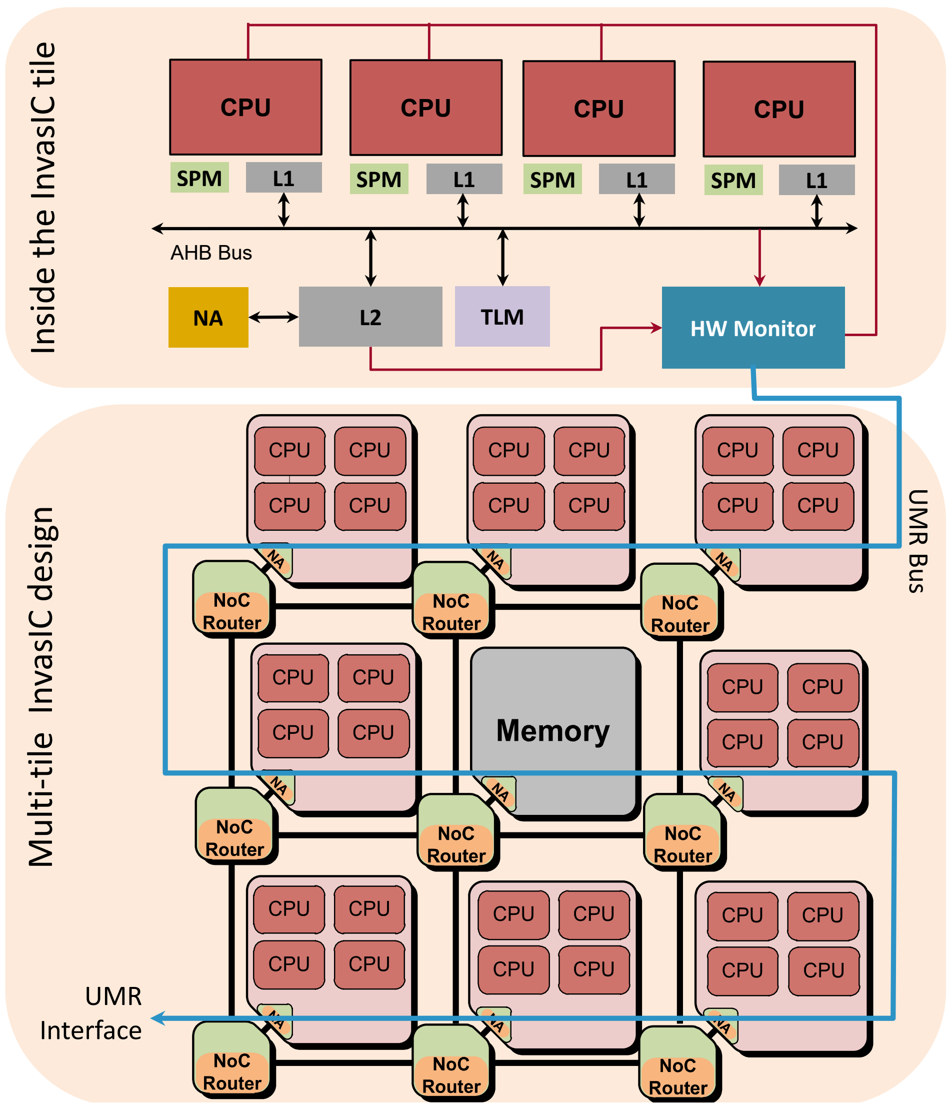


Figure A.4.: Hardware monitor design

Section A.3 describes an application of the hardware monitoring system described above, a novel methodology for mapping stack and heap data structures based on the self adaptive principles of Invasive Computing.

A.2. Stack and Heap Mapping using Hints from Hardware Monitors

MPSoC designs are widely used today as they offer significant performance boost at lower power consumption, by running each PE at a lower frequency compared to the conventional single core designs operating at very high frequencies. However, the number of PEs on such

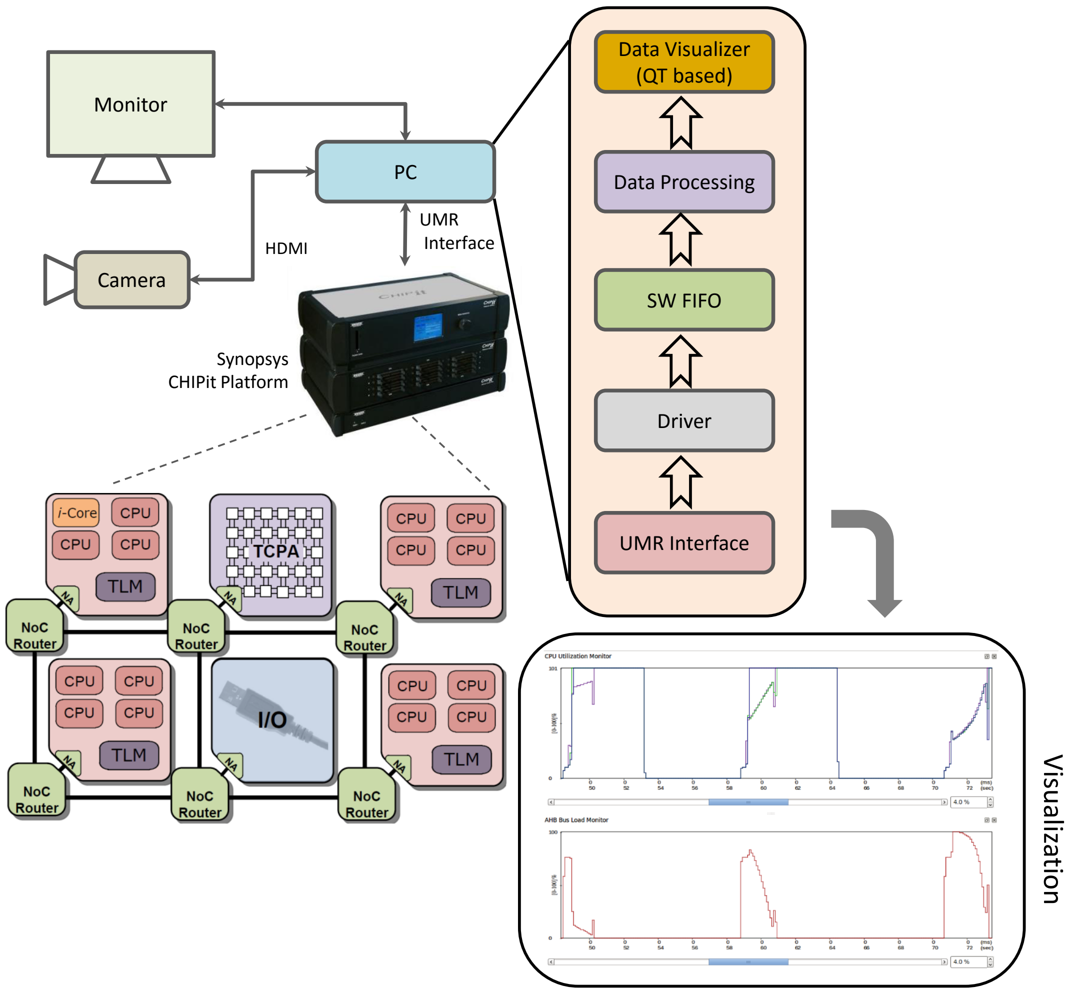


Figure A.5.: Various blocks involved in the visualization of monitoring data

designs are often limited by the bandwidth of the shared bus, ring or a NoC. A bottleneck on the bus/NoC may prevent the data intensive video application from scaling beyond few PEs, since the shared interconnect is heavily loaded as more threads starts to run in parallel.

For example, Intel's SCC does not offer cache coherency over the whole chip, as maintaining such a coherence, will lead to heavy data exchange (cache coherence data) over the NoC. This will reduce the overall efficiency of the NoC and also lead to higher power consumption. In contrast to SCC, Tilera [3] supports cache coherency over the whole chip. This comes with additional costs of an enhanced multi-layer NoC which has dedicated NoC layers for cache coherent data traffic resulting in more complex HW in terms of interconnect and transistor count as well as development costs. On the other hand it keeps the system fully cache coherent making programming easier as the programmer does not have to write additional code to maintain data consistency across threads running on different cores.

Performance results (see Figure A.6) from Harris Corner detection demonstrates the above mentioned scalability problem on a single tile, within the InvasIC. In a bus based system, cache coherency is maintained using various snooping techniques, with bus snooping being a widely used one nowadays, due to its implementation simplicity. The system consists of a write through L1 cache where each L1 update triggers a bus access to update the data in the L2 cache. The L2 cache is write back, the data is written back to DDR memory only when there is a read or write miss. The write-through nature on the L1 cache creates a significant load over the bus and limits the scalability of the system.

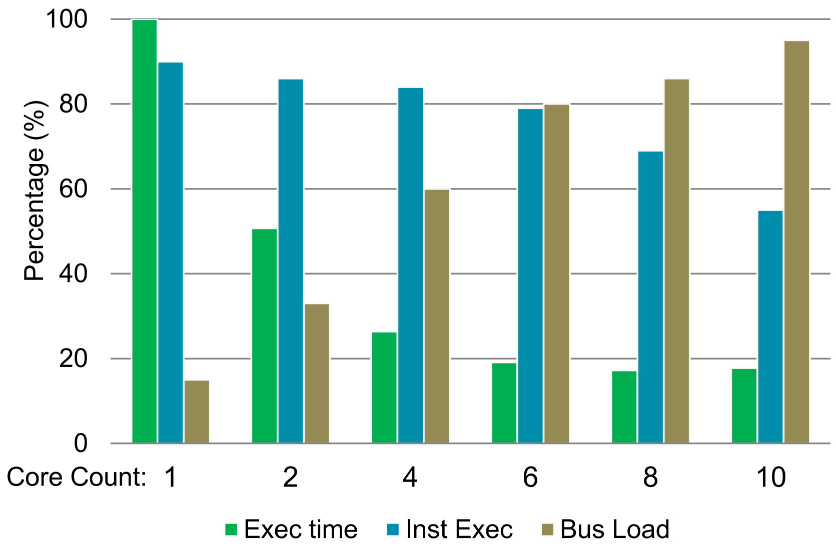


Figure A.6.: Execution profile for Harris Corner Detection

To ensure scalability in the HCD application (algorithmically), the input image is split into many small blocks of pixels which can be processing independently by each *i*-let/PE. The application is structured to scale almost linearly with the number of PEs. However, Figure A.6 shows that the application does not scale linearly with increasing parallelism due to the heavy bus load. Figure A.7 shows a more detailed view of the data traffic on the bus, and also indicates what percentage of the traffic results from input and output data, shared data-structures and *i*-let local data. Some data structures in the heap are *i*-let local (heap local), while others can be accessed and updated by other *i*-lets (heap shared). Input & output refer to the input and output data of the application program (it is assumed that input data is always read from and output data written to external memory). It can be seen clearly that approximately 40% of the bus load is created by read and write operations on the stack or local variables within the *i*-lets. It should be noted that this traffic is completely unnecessary as the stack of an *i*-let is local to the *i*-let and hence cache coherence is not required. Nevertheless, cache coherence is maintained over the complete data-set by the write through L1 cache since it does not distinguish between *i*-let local and shared data.

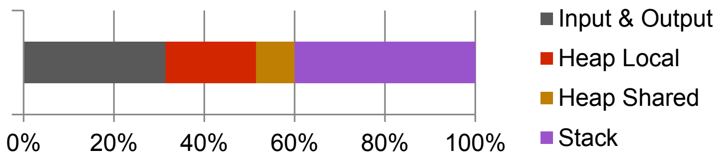


Figure A.7.: Busload distribution

An alternative solution which has been applied widely in the past is to include scratchpad memories (SPM) in the design. SPM are high-speed on-chip RAM blocks and are tightly coupled to a PE, in contrast to the TLM, which is shared between PEs on a tile. For example, the IBM Cell Processors [127] offer such on-chip buffers for each of its slave processing elements. Prefetched input data and intermediate results can be placed in these buffers and final results can be written back to external memory, thus reducing traffic on the bus, NoC and external memory.

A.3. Application Analysis using Hardware Monitors

The analysis starts with profiling the existing multi-threaded application using the performance counters within the monitoring system. The memory access profile is visualized on a display and analyzed by the programmer to understand the bottlenecks in the system. For example, if the CPI value of the core is high and there is significantly high access to stack data, the programmer can remap the stack of the thread to local-SPM. If stack access is not significantly high but access to the heap needs to be optimized, the programmer can use *malloc* to map the data structure to the SPM. It should be noted that data structures being accessed by more than one thread concurrently, cannot be mapped to SPM, but instead to shared TLM. Once the application code is adapted based on its memory access pattern, it is re-evaluated. The process completes if significant improvement is observed. In some cases the heap mapping might require multiple iterations as various *malloc* calls might exists in the program and the programmer may want to evaluate them one at a time or in combination based on nature of the algorithm implemented.

Various mapping schemes (for both stacks and heaps) were analyzed for a variety of applications. The upcoming sections demonstrate how the new scheme can lead to better performance and scalability on an MPSoC like InvasIC. Each core has a dedicated SPM of 8 KB size. A TLM (attached to the AHB Bus) of 64 KB size, was used for mapping shared datastructures. L1 & L2 caches are both 4-way set-associative with a size of 32 KB and 256 KB respectively.

A.3.1. Evaluation Techniques

This section demonstrates the effectiveness of the new technique using four different application, as described below. As the evaluation were performed in a early phase of the project, a multi-tile InvasIC design was not available for evaluation. Hence the evaluation performed in this section were based on a single tile implementation with 10 LEON3 PEs. In the first step the conventional multi-threaded application is used on the InvasIC design (single tile) without using SPMs or TLMs. In the second step the multi-threaded model is adapted by remapping stacks and heaps as mentioned before. This enables the applications to use SPMs and TLMs and to evaluate this model against the initial model for performance improvements, scalability, energy consumption, etc.

A.3.1.1. Harris Corner Detection

The Harris Corner Detection algorithm used for evaluation is based on the Integrating Vision Toolkit (IVT) library [111]. The sequential application code from the IVT was first converted into a multi-threaded model. The algorithm consists of three main stages. First stage is called *harris-map* which creates a signature for every pixel in the image based on its neighbors, using a 2D sliding window over the complete image. This is followed by a *quick-sort* to sort and filter the best corner points followed by a *spatial filtering* technique to remove corners too close to each other. Both *harris-map* and *spatial filtering* stages were converted to multi-threaded models and the *quick-sort* stage remains sequential as the time taken for sorting is negligibly low and does not offer a significant benefit when converted to a parallel model. The execution profile of the multi-threaded model was already explained in Figure A.6. It can be seen that the application is highly scalable as the execution time is reduced to half when two threads execute in parallel compared to the single threaded model. From Figure A.6, it is clear that the application continue to scale well up to 6 cores Beyond this point the performance does not improve significantly due to heavy bus load. Furthermore, the execution time increases slightly when executed on 10 cores as compared to 8, as the bus is almost saturated during this transition.

Table A.1.: Using SPM for Harris Corner detection

Core count		1	2	4	6	8	10
Time(ms)	B	2406	1220	634	459	415	**427**
	A	2270	1149	590	415	335	**294**
Busload (%)	B	15	33	60	80	86	95
	A	6	9	23	40	50	62
Mem access	B	980	1810	2650	3320	4200	5460
	A	910	1550	2010	2450	3020	3810

The hardware monitoring system was used to captured the bus load information for Harris Corner detection (see Figure A.7). This profile clearly shows that stack access (read & write) accounts for 40% of the overall busload while input output data constitute only for 33% and heap (local + shared) constitute up to 27%. This clearly indicates that mapping the stack to a core local SPM would lead to significant reduction in busload and better performance as data is available locally. Based on this information, the stack for Harris Corner detection is mapped to a core local SPM. Resulting execution time profile is shown in Table A.1. Comparing the two scenarios, without SPM (before scenario, represented by *B*) and with SPM (after scenario, represented by *A*), it is clear that the execution time improved by 31% while running on a multi-core hardware with 10 PEs.

The external memory access was reduced by 30% (for 10 cores, see Figure A.8) and considering the fact that external memory access is heavily power consuming (almost an order of magnitude more expensive), the power consumed by the additional memory blocks (SPMs) can be overlooked. The load on the bus is reduced by more than 50% in all combinations which leads to better scalability for the overall system.

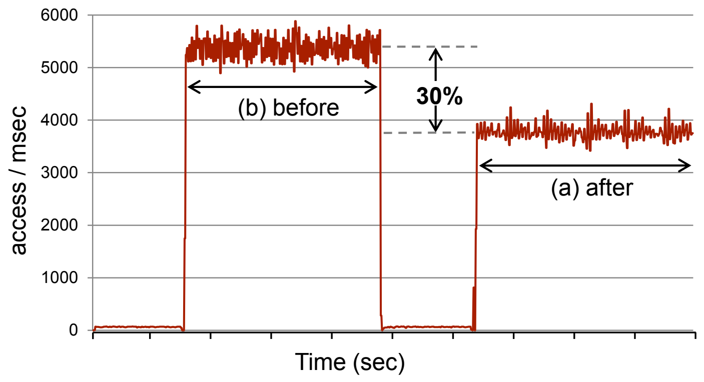


Figure A.8.: External memory access (before and after using SPM)

A.3.1.2. SIFT Feature Extraction

SIFT(Scale Invariant Feature Transform) is a widely used algorithm in the areas of robotics for object recognition. SIFT features are computed for specific points in the image and later compared to reference features extracted from the object to be recognized. The SIFT algorithm is computationally intensive due to the large number of floating-point operations to be performed during the SIFT feature extraction process. As a result the algorithm does not saturate the AHB bus (even while running in parallel on 10 cores) and is therefore an example to evaluate how compute bound applications behave under the new memory management scheme presented in this work. Figure A.9 shows the bus load distribution for stacks and heaps during the execution of the SIFT application.

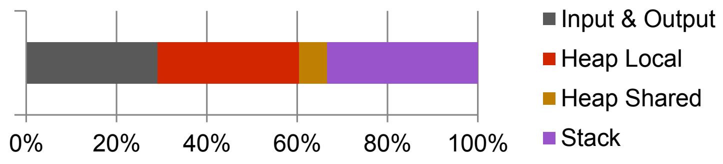


Figure A.9.: Bus load distribution for SIFT feature extraction

Similar to the Harris Corner detection, the SIFT application is evaluated in two steps. At first, a multi-threaded model of the application was developed where each core processes a specific set of features. This model is highly scalable as any feature in the image can be processed independent of others. The second model is an enhanced version of the first, with techniques to remap stack and heap data. From Figure A.9, it can be seen that stack data leads to a significant load over the bus and hence suitable for mapping on to the the local SPM. In case of heap, the heap local data points to data structures allocated by the thread for storing temporary results. As the stack space required by the SIFT application is quite low (less than 2 KB) some of the heap local data structures could also be mapped to the SPM. The shared heap contains data structures which mainly constitute input data to the thread (which feature point to operate on, where to store results etc.) Such data structures were mapped to TLM as they need to be accessed by all concurrently running threads.

Looking at Table A.2, it can be seen that the new approach significantly reduced bus load and external memory access. External memory access was reduced by up to 74% when

Table A.2.: Using SPM for SIFT feature extraction

Core count		1	2	4	6	8	10
Time(ms)	B	1359	764	429	325	266	237
	A	1352	757	422	318	258	225
Busload (%)	B	15	20	28	35	43	48
	A	8	10	12	14	17	20
Mem access	B	870	1600	3050	3400	3800	4200
	A	780	800	855	890	1030	1100

all cores were active. Correspondingly, bus load was reduced by 58% which offers space for adding more cores into the design. The execution time did not improve significantly as in case of Harris Corner detection. This is because of the compute bound nature of the application which did not saturate the bus, even when all cores where used.

A.3.1.3. Audio Processing

To show the benefits of using the new technique for audio processing, a multi-channel audio equalizer was used. The equalizer can process up to six audio channels with an input sampling rate of 96 kHz/channel. Each channel is processed independently, limiting the scalability to a maximum of 6 threads. At first, each channel is filtered into 10 different bands using a bandpass IIR filter. A unique gain value is then applied to each band based on the current equalizer setting and then combined to form a single output channel. Figure A.10 shows the bus load distribution. The stack space required for the audio application is quite low (less than 1 KB) which gives ample space for local heap data structures to be mapped to the SPM. Shared data structures (heap-shared) are mapped to the TLM (including the address where the raw audio samples are stored, equalizer inputs etc.).

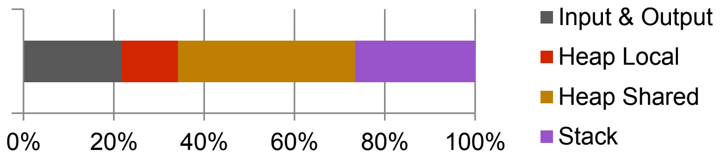


Figure A.10.: Bus load distribution for audio processing

Table A.3 lists the benefits of using the new technique on the audio equalizer application. The results show a significant reduction in external memory access and bus load.

A.3.1.4. Contrast Enhancement (AIVHE)

Contrast enhancement is a well known technique widely used in image processing. In this evaluation an advanced algorithm for contrast enhancement called AIVHE (Adaptive Histogram Equalization) [128] is used. This technique results in high quality results with low noise as compared to the standard algorithm. The process starts by building a histogram of

Table A.3.: Using SPM for audio processing

Core count		1	2	4	6
Time(ms)	B	1140	585	297	200
	A	1030	520	262	175
Busload (%)	B	21	48	53	65
	A	5	10	23	38
Mem access	B	2500	4950	7340	8700
	A	2080	2450	3130	3960

the input image (gray scale 8 bit image was used) resulting in a histogram with 256 bins. This histogram is then converted to an equalized histogram using the technique described in [128]. In the last step, each pixel in the input image is read again and updated with the new value based on the new equalized histogram. Therefore the input image gets loaded twice from external memory resulting in an application where most of the time is spend on reading and writing data, as shown in Figure A.11.

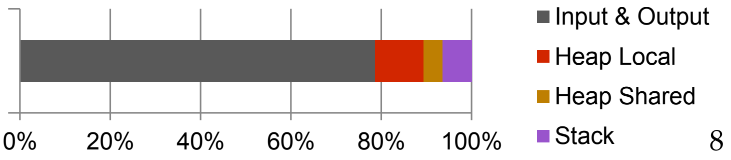


Figure A.11.: Bus-load distribution for contrast enhancement

As input & output data always flow over the bus (from external memory). Hence, this algorithm does not show a significant improvement through stack and heap relocation to SPMs and TLMs. Access to the stack and heap data structures are quite low and hence this application shows least improvement in terms of execution time as shown by Table A.4. Therefore, applications which mostly spend time on reading and writing data from external memory may not benefit much form the technique described in this chapter.

Table A.4.: Execution time for contrast enhancement with & without SPM

Core Count	1	2	4	6	8	10
No SPM(ms)	100.2	50.5	26.3	21.8	20.6	20.0
With SPM(ms)	99.1	49.0	25.0	19.7	18.8	18.0

A.3.2. Results

Table A.5 contains a summary of the overall benefits with the numbers indicating an overall reduction in external memory access, execution time and bus load, when executed on a multi-core platform with 10 Leon3 PEs.

It can be seen that a few modifications to the existing multi-threaded application can result in significant improvement in performance and power consumption.

Table A.5.: Summary of overall benefits

Application	Memory-access	Execution-time	Bus-load
Harris Corner	30%	31%	33%
SIFT Feature Ext.	74%	5%	58%
Audio Equalizer	55%	13%	27%

A.4. Conclusion

On a large MPSoC design like InvasIC, several factors including caches, FPU, bus interconnects, network-on-chip, on-chip and off-chip memories etc., influence the performance of the application program. In order to facilitate the MPSoC design process and to avoid bottlenecks, performance counter or hardware monitors were built into the design. The design (hardware, interface for data exchange and visualization software) were described in Section A.1.

Sections A.2 and A.3 presented some applications of the hardware monitors in real-world situations, by remapping stacks and heaps to various on-chip memories on a cache coherent multi-core design. The hardware monitoring system described in Section A.1 was used to extract highly accurate information related to memory access made by each application program. Case studies were conducted on an FPGA based hardware prototypes to ensure the validity of results. Different video and audio applications were used to validate effectiveness of this approach. Reducing the cache coherent traffic reduced the bus load leading to higher scalability or better performance. The present version of the operating system is based on a run to completion model. Hence, the results do not include the overhead to backup and restore data from local-SPM. This assumption remains valid for numerous video and audio applications as processing starts after the data is available and the massive parallelism allows threads to complete processing without blocking each other.

B. Own Publications

1. Resource Awareness on Heterogeneous MPSoCs for Image Processing [8]
2. Self-adaptive corner detection on MPSoC through resource-aware programming [10]
3. Self Adaptive Harris Corner Detection on Heterogeneous Many-core Processor [129]
4. Resource-Aware Programming for Robotic Vision [130]
5. Improving Efficiency of Embedded Multi-core Platforms with Scratchpad Memories [131]
6. Resource-Aware Harris Corner Detection based on Adaptive Pruning [12]
7. A Resource-Aware Nearest Neighbor Search Algorithm for K-Dimensional Trees [13]
8. Potentials and Challenges for Multi-Core Processors in Robotic Applications [132]
9. RTL Simulation of High Performance Dynamic Reconfiguration: A Video Processing Case Study [133]
10. Acceleration of Optical Flow Computations on Tightly-Coupled Processor Arrays [14]
11. Invasive Computing for Robotic Vision [134]
12. FPGA-based Real-time Moving Object Detection for Walking Robots [135]
13. Real-Time Motion Detection Based On SW/HW- Codesign for Walking Rescue Robots [136]

Bibliography

[1] T. Asfour, K. Regenstein, P. Azad, et al. ARMAR-III: An integrated humanoid platform for sensory-motor control. In *6th IEEE-RAS International Conference on Humanoid Robots*. IEEE, 2006.

[2] Tamim Asfour, Julian Schill, Heiner Peters, Cornelius Klas, Jens Bucker, Christian Sander, Stefan Schulz, Artem Kargov, Tino Werner, and Volker Bartenbach. Armar-4: A 63 dof torque controlled humanoid robot. In *Humanoid Robots (Humanoids), 2013 13th IEEE-RAS International Conference on*, pages 390–396. IEEE, 2013.

[3] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.C. Miao, J.F. Brown, and A. Agarwal. On-chip interconnection architecture of the tile processor. *Micro, IEEE*, 27(5):15–31, 2007.

[4] Philipp Gschwandtner, Thomas Fahringer, and Radu Prodan. Performance analysis and benchmarking of the intel scc. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 139–149. IEEE, 2011.

[5] Denis Foley, Maurice Steinman, Alex Branover, Greg Smaus, Antonio Asaro, Swamy Punyamurtula, and Ljubisa Bajic. Amd's llano fusion apu. In *IEEE/ACM Symposium on High Performance Chips (HOTCHIPS)*, 2011.

[6] A.A. Abbo, R.P. Kleihorst, V. Choudhary, et al. Xetal-II: A 107 GOPS, 600 mW massively parallel processor for video scene analysis. *IEEE Journal of Solid-State Circuits*, 43(1):192–201, 2008.

[7] Jiri Gaisler. GRLIB IP Library User's Manual (Version 1.1. 0). *Gaisler Research*, 2010.

[8] Johny Paul, Benjamin Oechslein, Christoph Erhardt, Jens Schedel, Manfred Kröhnert, Daniel Lohmann, Walter Stechele, Tamim Asfour, Wolfgang Schröder-Preikschat, ÉR Sousa, Vahid Lari, Frank Hannig, and Jürgen Teich. Resource awareness on heterogeneous mpsocs for image processing. *Journal of Systems Architecture*, 2015.

[9] Frank Hannig, Moritz Schmid, Vahid Lari, Srinivas Boppu, and Jürgen Teich. System integration of tightly-coupled processor arrays using reconfigurable buffer structures. In *Proceedings of the ACM International Conference on Computing Frontiers*, page 2. ACM, 2013.

[10] Johny Paul, Benjamin Oechslein, Christoph Erhardt, Jens Schedel, Manfred Kröhnert, Daniel Lohmann, Walter Stechele, Tamim Asfour, and Wolfgang Schröder-Preikschat. Self-adaptive corner detection on mpsoc through resource-aware programming. *Journal of Systems Architecture*, 2015.

[11] P. Azad, T. Asfour, and R. Dillmann. Combining Harris interest points and the SIFT descriptor for fast scale-invariant object recognition. In *Intelligent Robots and Systems, 2009. IROS 2009*. IEEE, 2009.

[12] Johny Paul, Walter Stechele, Manfred Kröhnert, Tamim Asfour, Benjamin Oechslein, Christoph Erhardt, Jens Schedel, Daniel Lohmann, and Wolfgang Schröder-Preikschat. Resource-aware harris corner detection based on adaptive pruning. In *Architecture of Computing Systems–ARCS 2014*, pages 1–12. Springer, 2014.

[13] Johny Paul, Walter Stechele, Manfred Kroehnert, Tamim Asfour, Benjamin Oechslein, Christoph Erhardt, Jens Schedel, Daniel Lohmann, and Wolfgang Schroder-Preikschat. A resource-aware nearest-neighbor search algorithm for k-dimensional trees. In *Design and Architectures for Signal and Image Processing (DASIP), 2013 Conference on*, pages 80–87. IEEE, 2013.

[14] ÉR Sousa, Alexandru Tanase, Vahid Lari, Frank Hannig, Jürgen Teich, Johny Paul, Walter Stechele, Manfred Kröhnert, and Tamin Asfour. Acceleration of optical flow computations on tightly-coupled processor arrays. In *Workshop on Parallel Systems and Algorithms (PARS)*, 2013.

[15] Paul S Segerstrom. Intel economics*. *International Economic Review*, 48(1):247–280, 2007.

[16] Nathan Goulding-Hotta, Jack Sampson, Steven Swanson, Michael Bedford Taylor, Ganesh Venkatesh, Saturnino Garcia, Joe Auricchio, Po-Chao Huang, Manish Arora, Siddhartha Nath, et al. The greendroid mobile application processor: An architecture for silicon's dark future. *IEEE Micro*, pages 86–95, 2011.

[17] Jonathan Dorn, Robbie Hott, and Michelle McDaniel. Exploring performance and power scaling in multi-core processors, 2011.

[18] Shekhar Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference*, pages 746–749. ACM, 2007.

[19] CARE: The Coordiantion Action for Robtics in Europe. Robotics visions – to 2020 and beyond: The strategic research agenda for robotics in europe, 2009.

[20] S. Sakagami, T. Watanabe, C. Aoyama, S. Matsunage, N. Higaki, and K. Fujimura. The Intelligent ASIMO: System Overview and Integration. In *iros*, pages 2478–2483, 2002.

[21] K. Akachi, K. Kaneko, N. Kanehira, S. Ota, G. Miyamori, M. Hirata, S. Kajita, and F. Kanehiro. Development of Humanoid Robot HRP-3. In *humanoids*, pages 2471–2478, 2005.

[22] Fujitsu, humanoid robot hoap-2, www.automation.fujitsu.com, 2003.

[23] Hiroyasu Miwa, Tetsuya Okuchi, Hideaki Takanobu, and Atsuo Takanishi. Development of a new human-like head robot WE-4. In *iros*, volume 3, pages 2443–2448, 2002.

[24] L. Aryananda and J. Weber. MERTZ: a quest for a robust and scalable active vision humanoid head robot. In *humanoids*, pages 513–532, 2004.

[25] G. Cheng, S.-H. Hyon, J. Morimoto, A. Ude, J. G. Hale, G. Colvin, W. Scroggin, and S. C. Jacobsen. CB: a humanoid research platform for exploring neuroscience. *Advanced Robotics*, 21(10):1097–1114, 2007.

[26] G. Sandini, G. Metta, and D. Vernon. RobotCub: An open framework for research in embodied cognition. In *iros*, pages 13–32, 2004.

[27] T. Asfour, K. Regenstein, P. Azad, J. Schröder, N. Vahrenkamp, and R. Dillmann. ARMAR-III: An Integrated Humanoid Platform for Sensory-Motor Control. In *humanoids*, pages 169–175, Genova, Italy, December 2006.

[28] T. Asfour, P. Azad, N. Vahrenkamp, K. Regenstein, A. Bierbaum, K. Welke, J. Schröder, and R. Dillmann. Toward Humanoid Manipulation in Human-Centred Environments. *Robotics and Autonomous Systems*, 56:54–65, January 2008.

[29] N. Vahrenkamp, S. Wieland, P. Azad, D. Gonzalez, T. Asfour, and R. Dillmann. Visual Servoing for Humanoid Grasping and Manipulation Tasks. In *humanoids*, pages 406–412, Daejeon, Korea, December 2008.

[30] N. Vahrenkamp, C. Böge, K. Welke, T. Asfour, J. Walter, and R. Dillmann. Visual servoing for dual arm motions on a humanoid robot. In *humanoids*, pages 208–214, Paris, France, December 2009.

[31] A. Benhimane and E. Malis. Homography-based 2d visual tracking and servoing. In *Special Joint Issue IJCV/IJRR on Robot and Vision. Published in The International Journal of Robotics Research*, volume 26, pages 661–676, July 2007.

[32] C. Connaire, E. Cooke, N. O'Connor, N. Murphy, and A.F. Smeaton. Background modelling in infrared and visible spectrum video for people tracking. In *International Conference on Computer Vision and Pattern Recognition*, pages 20–25, 2005.

[33] D. Scharstein and R. Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. In *Journal of Computer Vision*, volume 47, pages 7–42, 2002.

[34] B. Cope, P.Y.K Cheung, W. Luk, and S. Witt. Have GPUs made FPGAs redundant in the field of video processing. In *International Conference on Field Programmable Technology (FPL)*, pages 111–118, 2005.

[35] Sh. Che, J. Li, J.W. Sheaffer, K. Skadron, and J. Lach. Accelerating compute-intensive applications with GPUs and FPGAs. In *Symposium on Application Specific Processors (SASP)*, pages 101–107, 2008.

[36] K. Keutzer. Mapping applications onto manycore. In *Design Automation Conference (DAC)*, 2009.

[37] Zbigniew Bielewicz, Leszek Debowski, and Eugieniusz Lowiec. A dsp and fpga based integrated controller development solutions for high performance electric drives. In *Industrial Electronics, 1996. ISIE'96., Proceedings of the IEEE International Symposium on*, volume 2, pages 679–684. IEEE, 1996.

[38] Ping He, MH Jin, Lei Yang, R Wei, YW Liu, HG Cai, Hong Liu, Nikolaus Seitz, Jörg Butterfass, and Gerd Hirzinger. High performance dsp/fpga controller for implementation of hit/dlr dexterous robot hand. In *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, volume 4, pages 3397–3402. IEEE, 2004.

[39] Shekhar Borkar. Design challenges of technology scaling. *Micro, IEEE*, 19(4):23–29, 1999.

[40] B. Nelson, S. West, R. Curtis, et al. Comparing fine-grained performance on the Ambric MPPA against an FPGA. In *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2009.

[41] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, et al. An 80-tile 1.28 TFLOPS network-on-chip in 65nm CMOS. In *Solid-State Circuits Conference, 2007*, pages 98–589. IEEE, 2007.

[42] Mayank Daga, Ashwin M Aji, and Wu-chun Feng. On the efficacy of a fused CPU+GPU processor (or APU) for parallel computing. In *Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on*, pages 141–149. IEEE, 2011.

[43] T. Asfour, K. Regenstein, P. Azad, J. Schroder, et al. ARMAR-III: An Integrated Humanoid Platform for Sensory-Motor Control. In *Humanoid Robots, 2006 6th IEEE-RAS International Conference on*, pages 169 –175, Dec 2006.

[44] Y. Sakagami, R. Watanabe, C. Aoyama, S. Matsunaga, et al. The intelligent ASIMO: system overview and integration. In *Intelligent Robots and Systems, 2002. IEEE/RSJ International Conference on*, volume 3, pages 2478 – 2483, 2002.

[45] K. Kaneko, F. Kanehiro, M. Morisawa, K. Miura, S. Nakaoka, and S. Kajita. Cybernetic human HRP-4C. In *Humanoid Robots, 2009. Humanoids 2009. 9th IEEE-RAS International Conference on*, pages 7 –14, Dec 2009.

[46] Stefan Jorg, Mathias Nickl, Alexander Nothhelfer, et al. The computing and communication architecture of the dlr hand arm system. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pages 1055–1062. IEEE, 2011.

[47] Jürgen Teich, Jörg Henkel, Andreas Herkersdorf, Doris Schmitt-Landsiedel, Wolfgang Schröder-Preikschat, and Gregor Snelting. Invasive computing: An overview. In *Multiprocessor System-on-Chip*, pages 241–268. Springer, 2011.

[48] Anant Agarwal. The tile processor: A 64-core multicore for embedded processing. In *Proceedings of HPEC Workshop*, 2007.

[49] Sriram R Vangal, Jason Howard, Gregory Ruhl, Saurabh Dighe, Howard Wilson, James Tschanz, David Finan, Arvind Singh, Tiju Jacob, Shailendra Jain, et al. An 80-tile sub-100-w teraflops processor in 65-nm cmos. *Solid-State Circuits, IEEE Journal of*, 43(1):29–41, 2008.

[50] T.G. Mattson, M. Riepen, et al. The 48-core scc processor: the programmer's view. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.

[51] TR Maeurer and D Shippy. Introduction to the cell multiprocessor. *IBM journal of Research and Development*, 49(4):589–604, 2005.

[52] B. Hutchings, B. Nelson, S. West, et al. Optical flow on the ambric massively parallel processor array (MPPA). In *17th IEEE Symposium on Field Programmable Custom Computing Machines*. IEEE, 2009.

[53] Erik Saule, Kamer Kaya, and Ümit V Çatalyürek. Performance evaluation of sparse matrix multiplication kernels on intel xeon phi. In *Parallel Processing and Applied Mathematics*, pages 559–570. Springer, 2014.

[54] Benoît Dupont de Dinechin, Renaud Ayrignac, Pierre-Edouard Beaucamps, Patrice Couvert, Benoit Ganne, Pierre Guironnet de Massas, Frederique Jacquet, Simon Jones, Nicolas Morey Chaisemartin, Frédéric Riss, et al. A clustered manycore processor architecture for embedded and accelerated applications. In *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, pages 1–6. IEEE, 2013.

[55] Nvidia tegra, 2015.

[56] Qualcomm snapdragon, 2015.

[57] Deepu Talla and Jeremiah Golston. Using davinci technology for digital video devices. *Computer*, pages 53–61, 2007.

[58] David Barrie Thomas, Lee Howes, and Wayne Luk. A comparison of cpus, gpus, fpgas, and massively parallel processor arrays for random number generation. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 63–72. ACM, 2009.

[59] Kevin Klues, Barret Rhoden, Y Zhu, Andrew Waterman, and Eric Brewer. Processes and resource management in a scalable many-core OS. *HotPar10, Berkeley, CA*, 2010.

[60] Juan A Colmenares, Gage Eads, Steven A Hofmeyr, Sarah Bird, Miquel Moretó, David Chou, Brian Gluzman, Eric Roman, Davide B Bartolini, Nitesh Mor, et al. Tessellation: refactoring the os around explicit resource containers with continuous adaptation. In *DAC*, page 76, 2013.

[61] Henry Hoffmann, Jonathan Eastep, Marco D Santambrogio, Jason E Miller, and Anant Agarwal. Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments. In *Proceedings of the 7th international conference on Autonomic computing*, pages 79–88. ACM, 2010.

[62] Davide B Bartolini, Riccardo Cattaneo, Gianluca C Durelli, Martina Maggio, Marco D Santambrogio, and Filippo Sironi. The autonomic operating system research project: achievements and future directions. In *Proceedings of the 50th Annual Design Automation Conference*, page 77. ACM, 2013.

[63] Henry Hoffmann, Martina Maggio, Marco D. Santambrogio, Alberto Leva, and Anant Agarwal. Seec: A framework for self-aware management of multicore resources. Technical Report MIT-CSAIL-TR-2011-016, Computer Science and Artificial Intelligence Laboratory, MIT, March 2011.

[64] Stephen Bo Furber. *ARM system-on-chip architecture*. pearson Education, 2000.

[65] Jason Howard, Saurabh Dighe, Sriram R Vangal, Gregory Ruhl, Nitin Borkar, Shailendra Jain, Vasantha Erraguntla, Michael Konow, Michael Riepen, Matthias Gries, et al. A 48-core ia-32 processor in 45 nm cmos using on-die message-passing and dvfs for performance and power scaling. *Solid-State Circuits, IEEE Journal of*, 46(1):173–183, 2011.

[66] Jörg Henkel, Lars Bauer, Michael Hübner, and Artjom Grudnitsky. i-core: A run-time adaptive processor for embedded multi-core systems. In *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA 2011)*, 2011.

[67] Artjom Grudnitsky, Lars Bauer, and Jörg Henkel. Corefab: concurrent reconfigurable fabric utilization in heterogeneous multi-core systems. In *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, page 5. ACM, 2014.

[68] A. Downton and D. Crookes. Parallel architectures for image processing. *Electronics & Communication Engineering Journal*, 10(3), 1998.

[69] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, et al. The landscape of parallel computing research: A view from berkeley. Technical report,

Citeseer, 2006.

[70] Shravan Muddasani, Srinivas Boppu, Frank Hannig, Boris Kuzmin, Vahid Lari, and Jürgen Teich. A prototype of an invasive tightly-coupled processor array. In *Design and Architectures for Signal and Image Processing (DASIP), 2012 Conference on*, pages 1–2. IEEE, 2012.

[71] Dmitrij Kissler, Frank Hannig, Alexey Kupriyanov, and Jurgen Teich. A highly parameterizable parallel processor array architecture. In *Field Programmable Technology, 2006. FPT 2006. IEEE International Conference on*, pages 105–112. IEEE, 2006.

[72] Vahid Lari, Andriy Narovlyanskyy, Frank Hannig, and Jiirgen Teich. Decentralized dynamic resource management support for massively parallel processor arrays. In *Application-Specific Systems, Architectures and Processors (ASAP), 2011 IEEE International Conference on*, pages 87–94. IEEE, 2011.

[73] Vahid Lari, Shravan Muddasani, Srinivas Boppu, Frank Hannig, Moritz Schmid, and Jürgen Teich. Hierarchical power management for adaptive tightly-coupled processor arrays. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 18(1):2, 2013.

[74] Benjamin Oechslein, Jens Schedel, Jürgen Kleinöder, Lars Bauer, Jörg Henkel, Daniel Lohmann, and Wolfgang Schröder-Preikschat. OctoPOS: A parallel operating system for invasive computing. In *Proceedings of the International Workshop on Systems for Future Multi-Core Architectures (SFMA). EuroSys*, 2011.

[75] Andrew Baumann, Paul Birham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating System Principles (SOSP 2009), October 11–14, 2009, Big Sky, MT, USA*, pages 29–44, New York, NY, USA, 2009. ACM Press.

[76] Hans P Moravec. Rover visual obstacle avoidance. In *IJCAI*, pages 785–790, 1981.

[77] Chris Harris and Mike Stephens. A combined corner and edge detector. In *Alvey vision conference*, volume 15, page 50. Manchester, UK, 1988.

[78] Cordelia Schmid and Roger Mohr. Local grayvalue invariants for image retrieval. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(5):530–534, 1997.

[79] David G Lowe. Object recognition from local scale-invariant features. In *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*, volume 2, pages 1150–1157. Ieee, 1999.

[80] David G Lowe. Local feature view clustering for 3d object recognition. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, volume 1, pages I–682. IEEE, 2001.

[81] David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[82] Krystian Mikolajczyk and Cordelia Schmid. A performance evaluation of local descriptors. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 27(10):1615–1630, 2005.

[83] T. Asfour, K. Welke, P. Azad, A. Ude, and R. Dillmann. The Karlsruhe Humanoid Head. In *humanoids*, pages 447–453, Daejeon, Korea, December 2008.

[84] Jianbo Shi and Carlo Tomasi. Good features to track. In *Computer Vision and Pattern Recognition, 1994. Proceedings CVPR'94., 1994 IEEE Computer Society Conference on*, pages 593–600. IEEE, 1994.

[85] Daniel J Mirota, Masaru Ishii, and Gregory D Hager. Vision-based navigation in image-guided interventions. *Annual review of biomedical engineering*, 13:297–319, 2011.

[86] Shoaib Ehsan and Klaus D McDonald-Maier. On-board vision processing for small uavs: Time to rethink strategy. In *Adaptive Hardware and Systems, 2009. AHS 2009. NASA/ESA Conference on*, pages 75–81. IEEE, 2009.

[87] Adam Schmidt, Marek Kraft, and Andrzej Kasiński. An evaluation of image feature detectors and descriptors for robot navigation. In *Computer Vision and Graphics*, pages 251–259. Springer, 2010.

[88] Steffen Gauglitz, Tobias Höllerer, and Matthew Turk. Evaluation of interest point detectors and feature descriptors for visual tracking. *International journal of computer vision*, 94(3):335–360, 2011.

[89] Minjie Li, Liqiang Wang, and Ying Hao. Image matching based on sift features and kd-tree. In *2010 2nd International Conference on Computer Engineering and Technology*, volume 4, 2010.

[90] Kahlouche Souhila and Achour Karim. Optical flow based robot obstacle avoidance. *International Journal of Advanced Robotic Systems*, 4(1):13–16, 2007.

[91] Fridtjof Stein. Efficient computation of optical flow using the census transform. In *Pattern Recognition*, pages 79–86. Springer, 2004.

[92] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys (CSUR)*, 31(4):406–471, 1999.

[93] David Wentzlaff and Anant Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *ACM SIGOPS Operating Systems Review*, 43(2):76–85, 2009.

[94] S Alkaabi and F Deravi. Candidate pruning for fast corner detection. *Electronics Letters*, 40(1):18–19, 2004.

[95] Meiqing Wu, Nirmala Ramakrishnan, Siew-Kei Lam, and Thambipillai Srikanthan. Low-complexity pruning for accelerating corner detection. In *Circuits and Systems (ISCAS), 2012 IEEE International Symposium on*, pages 1684–1687. IEEE, 2012.

[96] Jonathan Klippenstein and Hong Zhang. Quantitative evaluation of feature extractors for visual slam. In *Computer and Robot Vision, 2007. CRV'07. Fourth Canadian Conference on*, pages 157–164. IEEE, 2007.

[97] Jan Elseberg, Stéphane Magnenat, Roland Siegwart, et al. Comparison of nearest-neighbor-search strategies and implementations for efficient shape registration. *Journal of Software Engineering for Robotics*, 3(1):2–12, 2012.

[98] J.S. Beis and D.G. Lowe. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In *IEEE Computer Society Conference on Computer Vision*

and Pattern Recognition, 1997.

[99] Vincent Garcia, Eric Debreuve, and Michel Barlaud. Fast k nearest neighbor search using gpu. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on*. IEEE, 2008.

[100] Lawrence Cayton. Accelerating nearest neighbor search on manycore systems. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 402–413. IEEE, 2012.

[101] Lawrence Cayton. A nearest neighbor data structure for graphics hardware. *Proceedings of ADMS*, 2010.

[102] S. Bell, B. Edwards, J. Amann, et al. Tile64-processor: A 64-core soc with mesh interconnect. In *Solid-State Circuits Conference, 2008. Digest of Technical Papers*, pages 88–598. IEEE, 2008.

[103] Synopsys. http://www.synopsys.com/Systems/FPGABasedPrototyping/CHIPit/CapsuleModule/CHIPit_platinum_Edition.pdf, 2009.

[104] Frank Hannig, Vahid Lari, Srinivas Boppu, Alexandru Tanase, and Oliver Reiche. Invasive tightly-coupled processor arrays: A domain-specific architecture/compiler co-design approach. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):133:1–133:29, 2014.

[105] Pradip Mainali, Qiong Yang, Gauthier Lafruit, Rudy Lauwereins, and LV Gool. Lococo: Low complexity corner detector. In *Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on*, pages 810–813. IEEE, 2010.

[106] Lucas Teixeira, Waldemar Celes Filho, and Marcelo Gattass. Accelerated corner-detector algorithms. In *BMVC*, pages 1–10, 2008.

[107] Miguel Arias-Estrada and Eduardo Rodríguez-Palacios. An fpga co-processor for real-time visual tracking. In *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, pages 710–719. Springer, 2002.

[108] Marek Kraft, Adam Schmidt, and Andrzej J Kasinski. High-speed image feature detection using fpga implementation of fast algorithm. *VISAPP (1)*, 8:174–9, 2008.

[109] Fouzhan Hosseini, Amir Fijany, and Jean-Guy Fontaine. Highly parallel implementation of harris corner detector on csx simd architecture. In *Euro-Par 2010 Parallel Processing Workshops*, pages 137–144. Springer, 2011.

[110] Enqiang Sun, Dana Schaa, Richard Bagley, Norman Rubin, and David Kaeli. Enabling task-level scheduling on heterogeneous platforms. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, pages 84–93. ACM, 2012.

[111] Pedram Azad. Integrated Vision Toolkit (IVT), 2011.

[112] Jürgen Teich, Alexandru Tanase, and Frank Hannig. Symbolic mapping of loop programs onto processor arrays. *Journal of Signal Processing Systems*, 77(1–2):31–59, 2014.

[113] Éricles R. Sousa, Alexandru Tanase, Frank Hannig, and Jürgen Teich. Accuracy and Performance Analysis of Harris Corner Computation on Tightly-Coupled Processor Arrays. In *Proceedings of the Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pages 88–95. IEEE, October 2013.

[114] Artjom Grudnitsky, Lars Bauer, and Jörg Henkel. Partial online-synthesis for mixed-grained reconfigurable architectures. In *Proceedings of Design, Automation and Test in Europe Conference (DATE)*, pages 1555–1560, 2012.

[115] Artjom Grudnitsky, Lars Bauer, and Jörg Henkel. COREFAB: Concurrent reconfigurable fabric utilization in heterogeneous multi-core systems. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2014.

[116] Fridtjof Stein. Efficient computation of optical flow using the Census Transform. In Carl Edward Rasmussen, Heinrich H. Bülthoff, Bernhard Schölkopf, and Martin A. Giese, editors, *DAGM-Symposium*, volume 3175 of *Lecture Notes in Computer Science*, pages 79–86, Tübingen, Germany, 2004. Springer.

[117] Christopher Claus, Andreas Laika, Lei Jia, and Walter Stechele. High performance FPGA-based optical flow calculation using the census transformation. *IEEE Intelligent Vehicle Symposium*, June 2009.

[118] M. Motomura. A dynamically reconfigurable processor architecture. In *Microprocessor Forum*, 2002.

[119] Volker Baumgarte, Gerd Ehlers, Frank May, Armin Nückel, Martin Vorbach, and Markus Weinhardt. PACT XPP - a self-reconfigurable data processing architecture. *The Journal of Supercomputing*, 26:167–184, 2003.

[120] Frank Bouwens, Mladen Berekovic, Bjorn De Sutter, and Georgi Gaydadjiev. Architecture enhancements for the ADRES coarse-grained reconfigurable array. In *3rd international conference on High performance embedded architectures and compilers(HiPEAC)*, pages 66–81, 2008.

[121] Karthikeyan Sankaralingam, Ramadass Nagarajan, Robert McDonald, Rajagopalan Desikan, Saurabh Drolia, M. S. Govindan, Paul Gratz, Divya Gulati, Heather Hanson, Changkyu Kim, Haiming Liu, Nitya Ranganathan, Simha Sethumadhavan, Sadia Sharif, Premkishore Shivakumar, Stephen W. Keckler, and Doug Burger. Distributed microarchitectural protocols in the TRIPS prototype processor. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 480–491, Washington, DC, USA, December 2006. IEEE Computer Society.

[122] Pierre Palatin, Yves Lhuillier, et al. Capsule: Hardware-assisted parallel execution of component-based programs. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, 2006.

[123] Ying Yi, Wei Han, et al. An ILP formulation for task mapping and scheduling on multi-core architectures. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, April 2009.

[124] Vahid Lari, Frank Hannig, and Jürgen Teich. Distributed resource reservation in massively parallel processor arrays. In *Proceedings of 18th Reconfigurable Architectures Workshop (RAW)*, Anchorage, USA, May 2011.

[125] V. Lari, F. Hannig, and J. Teich. System integration of tightly-coupled reconfigurable processor arrays and evaluation of buffer size effects on their performance. In *Proceedings of the International Conference on Parallel Processing Workshops*, pages 528–534. IEEE, 2009.

[126] Alexey Kupriyanov, Dmitrij Kissler, Frank Hannig, and Jürgen Teich. Efficient Event-driven Simulation of Parallel Processor Architectures. In *Proceedings of the 10th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, pages 71–80. ACM Press, 2007.

[127] M. Kistler, M. Perrone, and F. Petrini. Cell multiprocessor communication network: Built for speed. *Micro, IEEE*, 26(3):10–23, 2006.

[128] C.H. Lu, H.Y. Hsu, and L. Wang. A new contrast enhancement technique by adaptively increasing the value of histogram. In *Imaging Systems and Techniques, 2009. IST'09. IEEE International Workshop on*, pages 407–411. IEEE, 2009.

[129] Johny Paul, Walter Stechele, Ericles Sousa, Vahid Lari, Frank Hannig, Jurgen Teich, Manfred Krohnert, and Tamim Asfour. Self-adaptive harris corner detector on heterogeneous many-core processor. In *Design and Architectures for Signal and Image Processing (DASIP), 2014 Conference on*, pages 1–8. IEEE, 2014.

[130] Johny Paul, Walter Stechele, Manfred Kröhnert, and Tamim Asfour. Resource-aware programming for robotic vision. *arXiv preprint arXiv:1405.2908*, 2014.

[131] Johny Paul, Walter Stechele, Manfred Kroehnert, and Tamim Asfour. Improving efficiency of embedded multi-core platforms with scratchpad memories. In *?1st International Workshop on Multi-Objective Many-Core Design (MOMAC), in conjunction with Architecture of Computing Systems (ARCS), 2014 27th International Conference on*, pages 1–8. VDE, 2014.

[132] Andreas Herkersdorf, Johny Paul, Ravi Kumar Pujari, Walter Stechele, Stefan Wallentowitz, Thomas Wild, and Aurang Zaib. Potentials and challenges for multi-core processors in robotic applications. In *GI-Jahrestagung*, pages 2749–2764, 2013.

[133] Lingkan Gong, Oliver Diessel, Johny Paul, and Walter Stechele. Rtl simulation of high performance dynamic reconfiguration: A video processing case study. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 106–113. IEEE, 2013.

[134] Johny Paul, Walter Stechele, Manfred Kröhnert, Tamim Asfour, and Rüdiger Dillmann. Invasive computing for robotic vision. In *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, pages 207–212. IEEE, 2012.

[135] Andreas Laika, Johny Paul, Christopher Claus, Walter Stechele, Adam El Sayed Auf, and Erik Maehle. Fpga-based real-time moving object detection for walking robots. In *Safety Security and Rescue Robotics (SSRR), 2010 IEEE International Workshop on*, pages 1–8. IEEE, 2010.

[136] Johny Paul, Andreas Laika, Christopher Claus, Walter Stechele, Adam El Sayed Auf, and Erik Maehle. Real-time motion detection based on sw/hw-codesign for walking rescue robots. *Journal of real-time image processing*, 8(4):353–368, 2013.