



DiaSys
**A Method and Tool for Non-Intrusive Runtime Diagnosis of
Embedded Software**

Philipp Wagner

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.

Vorsitzender:

Prof. Dr.-Ing. Hans-Georg Herzog

Prüfende der Dissertation:

1. Prof. Dr. sc. techn. Andreas Herkersdorf
2. Prof. Dr. Martin Leucker,
Universität zu Lübeck

Die Dissertation wurde am 19. September 2018 bei der Technischen Universität München
eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am
18. Juni 2019 angenommen.

Acknowledgements

This thesis is the result of my research at the Chair of Integrated Systems (LIS) at the Technical University of Munich. It was a great experience. I had the time and the freedom to uncover problems, and to find innovative solutions to them. From every-day problems to deep-dives into technical details, this period had it all.

I am grateful to everybody who supported me during that time: Andreas Herkersdorf for convincing me to take on a PhD in the first place, for his continued support, and for the freedom to evolve this work into a direction that was not obvious from the start; Thomas Wild, who read many pages of draft writings and gave very valuable feedback on it; and Martin Leucker, who volunteered to act as second reviewer and whose remarks helped polish this thesis.

Without my colleagues at LIS none of this work would have been any fun. Thanks for the technical discussions, for the silly moments at lunch and for the beers deep at night. Equally important were all the administrative staff at LIS who helped us get our work done. Doris, Gabi and Verena: thank you!

During my time at the chair I supervised 28 student projects. Much of the work done in these projects fed into our evaluation platforms and inspired this thesis. Thank you!

My work on this thesis was funded by the DFG through the project InvasIC, and by the state of Bavaria through the SoC Doctor project. I had a great collaboration with many colleagues in these projects, especially with Albrecht Mayer and Lin Li at Infineon.

Finally, I am thankful for all friends and family who supported me during this work. Thanks to the “Ladies Lunch” crew for a weekly event to look forward to, and thanks to Susanne for having my back when I needed time to complete this thesis.

Cambridge, UK, September 2019

Philipp Wagner

Abstract

Embedded devices have become ubiquitous. Yet, developing software for such devices is challenging, with requirements ranging from high reliability to real-time constraints and a focus on safety and security. To keep up, or even increase, developer productivity, insight into the software as it executes on the chip is essential.

Today, the prevalent method to gather non-intrusive runtime observations is tracing. Systems like ARM CoreSight or Nexus 5001 observe CPUs, memories, and interconnects, compress this data, and send it off-chip, where specialized tools analyze the data to help the developer to understand the software execution. With tracing, the insight into the chip is limited by the off-chip interface. Since its bandwidth is orders of magnitude below the amount of observable data, developers face an observability gap: they need to limit their observations to short time frames, or to parts of the chip.

This work is based on the realization that insight for developers is not proportional to the amount of trace data they can access, but to the number of questions about the software execution that are answered. The goal of this work is to enable developers to ask questions, and to help them obtain the necessary information for the answer.

Towards this goal we present DiaSys, an approach to give developers comprehensive, non-intrusive insight into the software execution on embedded systems. We reduce the observability gap by distilling information out of the observation data directly on the chip in a hierarchical, fully configurable/programmable multi-step analysis process. DiaSys observes the software execution to generate “observation events,” which are then processed by a dataflow application. The execution of the analysis application can be shared between the chip and the host PC to flexibly trade off the cost of on-chip logic with the bandwidth of the off-chip interface.

With DiaSys, developers describe the collection and analysis of observation data as script written in a novel domain-specific language, the Dia Language. This high-level language is based on C and SQL to reduce the barrier of entry. It is target-independent and can be compiled by the Dia Compiler for different configurations of the Dia Engine, our execution environment with on-chip and off-chip components.

We have fully implemented DiaSys, with all hardware components targeting FPGAs. The observation and analysis components increase the size of an eight-core, micro-controller-class memory-less observed system by 23 percent, or less than the DRAM infrastructure in absolute numbers. In three case studies we evaluate DiaSys in a functional debugging/testing scenario, and in two profiling scenarios. We show that DiaSys can reduce the off-chip bandwidth by several orders of magnitude with the stated hardware cost. We further show that the given dimensioning is sufficient to create a sampling-based function profile for arbitrarily fast processor systems.

Contents

Acknowledgements	iii
Abstract	v
Contents	vii
List of Figures	xi
List of Tables	xiii
List of Abbreviations	xv
1 Introduction	1
1.1 Motivation	1
1.2 Contributions	5
1.3 Thesis Outline	6
1.4 Earlier Publications	7
2 Background and State of the Art	9
2.1 Background: How Software is Developed	9
2.1.1 Software Development Methodologies	9
2.1.2 Debugging	11
2.1.3 Software Testing	12
2.1.4 Verification	13
2.1.5 Software Diagnosis Needs a Toolbox	14
2.1.6 Summary: Software Development and Diagnosis	15
2.2 Non-Intrusive Software Observation	15
2.2.1 An Overview on Tracing Systems	16
2.2.2 A Closer Look: ARM CoreSight	17
2.2.3 A Closer Look: Nexus 5001	20
2.2.4 A Closer Look: Infineon MCDS	22
2.2.5 Discussion: Tracing Systems	24
2.2.6 Performance Counters	24
2.2.7 Summary: Non-Intrusive Software Observation	25
2.3 Observation Data Analysis	26
2.3.1 Trace Analysis Software for Embedded Systems	26
2.3.2 Scriptable Event-Based Debugging	27
2.3.3 Query-Based Debugging	29

2.3.4	A Closer Look: DTrace	30
2.3.5	Summary: Observation Data Analysis	31
2.4	Summary: Background and State of the Art	32
3	The DiaSys Approach to Software Diagnosis	35
3.1	Assumptions and Goals	35
3.2	The DiaSys Concept	37
3.3	Diagnosis Applications	38
3.4	Analyzability of Diagnosis Applications	40
3.5	Behavior in Overload Situations	40
3.6	On the Benefits of Programmable Diagnosis	41
3.7	Design Discussion: Limitations and Consequences	42
3.8	Summary	43
4	DiaSys Design and Realization	45
4.1	How DiaSys Works: A First Glimpse From a Developer’s Perspective	46
4.2	The Dia Language: A Domain-Specific Language for Diagnosis Descriptions	51
4.2.1	Goals and Requirements	51
4.2.2	Dia Language Overview	52
4.2.3	Event Specification Sub-Language	52
4.2.3.1	Simple Events	52
4.2.3.2	Observation Objects and Classes	53
4.2.3.3	Observation Events	54
4.2.3.4	Functions in the Event Specification Sub-Language	55
4.2.3.5	Design Rationale	56
4.2.4	Transformation Actor Sub-Language	58
4.2.4.1	General Structure	58
4.2.4.2	Transformation Actor Body	59
4.2.4.3	Design Rationale	61
4.2.5	Summary: The Dia Language	63
4.3	The Dia Compiler	65
4.3.1	Goals and Requirements	65
4.3.2	Introduction to Compilers	65
4.3.3	Design of the Dia Compiler	67
4.3.4	From Event Specifications to Event Generator Configurations	68
4.3.5	Translation of Transformation Actors	69
4.3.6	Type Mapping of Transformation Actors	69
4.3.7	Mapping, Allocation, and Scheduling of Transformation Actors	71
4.3.8	Implementation	73
4.3.9	Evaluation	74
4.3.9.1	Compiler Performance	74
4.3.9.2	Effectiveness of Processing Element Type Mapping	74
4.3.10	Summary: The Dia Compiler	76

4.4	The Dia Engine: The Workhorse of DiaSys	79
4.4.1	Architecture of the Dia Engine	79
4.4.2	Debug Modules	81
4.4.3	The Debug Network: Communication Within the Dia Engine	82
4.4.3.1	Segmenting the Debug Network Into Subnets	83
4.4.3.2	Gateways in the Debug Network	83
4.4.3.3	The Subnet Control Module	83
4.4.3.4	Discussion: The Design of the Debug Network	84
4.4.4	The Debug Network Protocol	85
4.4.4.1	Addressing in the DNP	86
4.4.4.2	Layering of the DNP	86
4.4.4.3	Debug Packets, The Common Data Exchange Format of the DNP	87
4.4.4.4	Accessing Debug Registers	88
4.4.4.5	Event Packets, a Multi-Purpose Data Container	89
4.4.5	On-Chip Hardware Architecture of the Dia Engine	90
4.4.5.1	On-Chip Debug Modules	91
4.4.5.2	On-Chip Debug Interconnect	92
4.4.5.3	Off-chip Connectivity	93
4.4.6	Host Software Architecture	94
4.4.6.1	Host Communication Over ZeroMQ	96
4.4.6.2	Architecture Summary	98
4.4.7	Implementation Overview	98
4.4.7.1	Target Platforms	99
4.4.7.2	Observed System	99
4.4.8	Implementation of the Core Event Generator (CEG)	100
4.4.8.1	Parametrization of the CEG	104
4.4.8.2	Resource Utilization of the CEG	107
4.4.9	Implementation of the Debug Interconnect	108
4.4.10	Implementation of the Diagnosis Processor (DIP)	108
4.4.11	Implementation of the Event Counter (CNT)	110
4.4.12	Implementation of Dia Engine Host Software	110
4.4.12.1	libosd: The Dia Engine Host Software as Library	110
4.4.12.2	Tools to Interact with the Dia Engine	111
4.4.12.3	Python Bindings	111
4.4.12.4	Quality Assurance of the Software Implementation	113
4.4.13	Summary: The Dia Engine	113
4.5	Summary: DiaSys Design and Realization	114
5	Case Studies	115
5.1	The “Chip Design” For Our Case Studies	115
5.1.1	Evaluation System Resource Usage	117
5.1.2	Data Rates for Tracing Systems	119

Contents

5.2	Case Study I: Debugging and Testing of Race Conditions	121
5.2.1	Problem Description	121
5.2.2	Debugging by Hand: Observe Exchanged Messages	122
5.2.3	Discussion: Manual Debugging with DiaSys	123
5.2.4	From Debugging to Testing: A Transaction Checking Test	125
5.2.5	Discussion: Testing	125
5.2.6	Summary: Debugging and Testing of Race Conditions with DiaSys	126
5.3	Case Study II: Function Profiling	128
5.3.1	Creating a Function Profile with Exact Accounting	128
5.3.1.1	Off-chip Profile Generation with Exact Accounting	128
5.3.1.2	On-chip Profile Generation with Exact Accounting	129
5.3.2	Creating a Function Profile with Sampling	129
5.3.2.1	Off-chip Profile Generation with Sampling	130
5.3.2.2	On-chip Profile Generation with Sampling	131
5.3.3	Summary: DiaSys for Function Profiling	131
5.4	Case Study III: Lock Contention Profiling of a Standard Benchmark Application	134
5.4.1	Evaluation Prototype	134
5.4.2	Problem Description	135
5.4.3	Measurement Approach	135
5.4.4	The Dia Script	136
5.4.5	Evaluation	138
5.4.5.1	Output of the Diagnosis Application	138
5.4.5.2	Event and Data Rates	139
5.4.5.3	Discussion	139
5.5	Summary: Case Studies	141
6	Conclusion and Outlook	143
	Bibliography	145

List of Figures

1.1	IoT connected devices installed worldwide.	2
1.2	Code size in selected embedded systems.	3
1.3	Predicted evolution of the logic size, the number of signal I/O pins, and the test I/O speed.	3
1.4	Architecture of today’s tracing systems compared to DiaSys.	5
2.1	The V-model software development methodology.	10
2.2	The Scrum software development methodology.	11
2.3	A schematic view of a common tracing system.	16
2.4	An example of the CoreSight architecture for tracing.	18
2.5	Combination of triggers from two sources in MCDS.	23
2.6	Programming of the trace qualification functionality in MCDS.	23
3.1	A schematic overview on the concept of DiaSys.	37
3.2	A model of a diagnosis application.	39
4.1	An overview of the DiaSys diagnosis method from a user’s perspective.	46
4.2	Graphical overview of the motivational example.	48
4.3	Classes and objects of observation units shown in an example.	53
4.4	General structure of a compiler.	66
4.5	An overview on the Dia Compiler.	68
4.6	The architecture of the Dia Engine in an exemplary setting.	80
4.7	Block diagram of a Debug Module.	81
4.8	Structure of a 16 bit wide Debug Network Protocol (DNP) address.	86
4.9	Layering of protocols in the DNP.	87
4.10	The structure of a Debug Packet.	87
4.11	Sequence chart showing the read access to a register.	88
4.12	The hardware of the Dia Engine shown in an exemplary configuration.	90
4.13	A Debug Packet when transferred over the on-chip NoC.	92
4.14	Wrapping a Debug Packet in a DTD.	93
4.15	Architecture of the Dia Engine as implemented in software on a PC.	95
4.16	Structure of a ZeroMQ message in the host communication protocol.	97
4.17	Implementation of the Core Event Generator (CEG) module in hardware.	101
4.18	Number of probes per SystemTap script.	106
4.19	Resource utilization of one instance of the Core Event Generator.	108
4.20	Block diagram of the Diagnosis Processor (DIP).	109
5.1	The “chip design” used for the case studies on DiaSys.	116

List of Figures

5.2	Distribution of total LUTs to the components of the Dia Engine.	117
5.3	Sequence diagram showing the race condition discussed in Case Study I.	123
5.4	Case Study I: Comparison of data rates	124
5.5	Different methods of program counter sampling.	130
5.6	Case Study II: Comparison of data rates	132
5.7	The software prototype of the diagnosis system used in Case Study III.	134
5.8	A graphical representation of the Dia script to create a lock contention profile.	136
5.9	Graphical representation of the data rate reductions achieved in the case studies.	142

List of Tables

4.1	List of management messages exchanged over host communication protocol.	98
4.2	List of CEG-related trace port signals.	102
4.3	Parameter values chosen for the Core Event Generator (CEG).	104
4.4	Resource utilization of one instance of the Core Event Generator.	107
4.5	Hierarchical resource utilization of the Diagnosis Processor.	109
5.1	Hierarchical resource utilization of our case study design.	118
5.2	Overview on all case studies regarding their off-chip traffic.	142

List of Abbreviations

ABI Application Binary Interface
API application programming interface
AST abstract syntax tree

BRAM Block RAM

CEG Core Event Generator
CPU Central Processing Unit

DI Debug Interconnect
DII Debug Interconnect Interface
DMA direct memory access
DNP Debug Network Protocol
DSL domain-specific language
DTD Debug Transport Datagram

EBNF Extended Backus-Naur form
ELF Executable and Linking Format

FIFO first in, first out (buffer)
FPGA Field Programmable Gate Array
FPU floating point unit

GCC GNU Compiler Collection
GDB the GNU Project Debugger
GPR general purpose register
GPU graphics processing unit
GUI graphical user interface

HDL hardware description language

List of Abbreviations

IoT	Internet of Things
IP	intellectual property
ISA	instruction set architecture
LUT	Look-Up Table
NoC	Network-on-Chip
OSD	Open SoC Debug
PC	program counter
PDG	program dependency graph
SCM	Subnet Control Module
SoC	System-on-Chip
SRAM	static random-access memory
TAP	test access port
TCP	Transmission Control Protocol
UART	Universal Asynchronous Receiver Transmitter
USB	Universal Serial Bus

1 Introduction

Good morning! It's a grayish winter day in 1998. Your alarm clock makes one last tick before its ring wakes you up and triggers the start of your well-practiced morning routine. You silence the alarm clock, turn on the radio at 104.6 MHz, and walk over to the bathroom to brush your teeth and take a shower. At the breakfast table you quickly flip through a couple of pages in the newspaper before it is time to get to work. You get behind the steering wheel and off you go.

It has been a long day at work. A very long one in fact. You start your way back home twenty years later, in the year 2018. You unplug your electric car from the charger in front of the office, get in and start the engine. "Drive me home!" you instruct the car, and the journey starts. A game on your smartphone keeps you entertained while the city flies past the windows. As the car heads up the driveway your house already awaits you. The door opens, lights turn on and the music start playing. It's Angels by Robbie Williams—just where you left off this morning. Welcome home!

Embedded systems surround us. They are everywhere: in our pocket, in our house, in the city infrastructure, in the cars and the trains, possibly even inside our body. At every tick of the seconds hand, millions of lines of software code command powerful processors to execute trillions of instructions. Embedded systems make our lives more entertaining, comfortable, and safe, often without us noticing. That is, as long as they function correctly. If not, we might be annoyed. Or dead.

This thesis is about improving software on embedded systems. It is about making sure these systems run correct, safe, and efficient code. To reach this goal this thesis presents a way to find and analyze software flaws, or bugs, in systems which might be distributed over the world, systems which must function correctly under all circumstances to preserve lives, tiny systems which process more data in a second than a whole data center could process only some years ago.

Embedded systems come with various name tags, such as IoT (Internet of Things) devices, "Industrie 4.0" controller, or smart devices. They have in common that hardware and software are tightly integrated to fulfill a specific task. To do so embedded systems often interact with the outside world through sensors and actuators, even though they are hidden behind the functionality they provide.

The market for embedded systems is growing steadily, research groups predict an annual worldwide growth between 4 and 9 percent [2, 3, 4]. Parts of the market are expected to grow at much faster pace. For example, the number of small connected devices ("IoT devices") is expected to increase by a factor of 4.9 in just ten years to

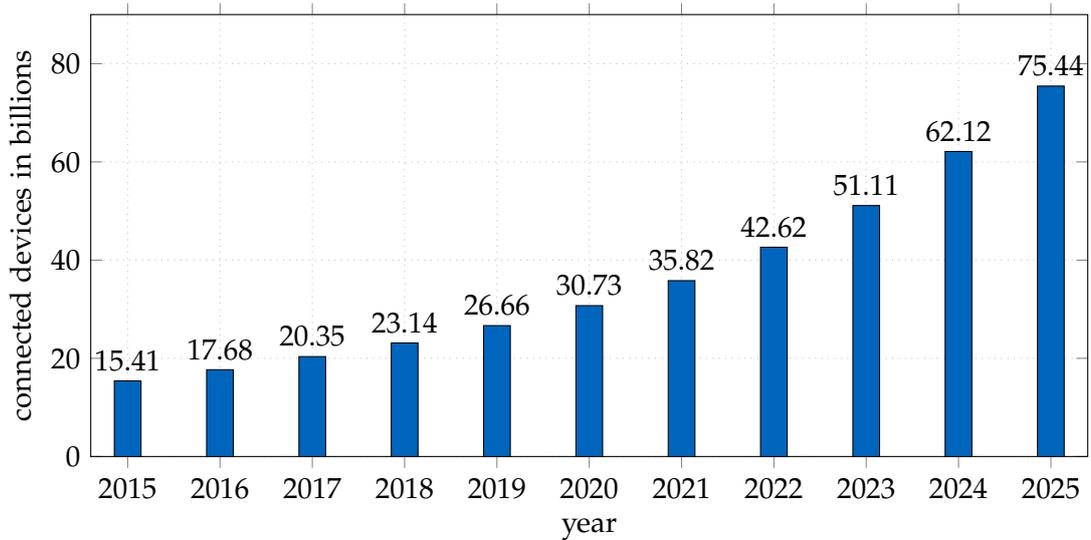


Figure 1.1: Internet of Things (IoT) connected devices installed worldwide in billions. Data from 2015 and 2016, predictions from 2017 until 2025 by IHS Markit [1].

75 billion devices, as predicted by the research group IHS Markit (Figure 1.1). But not only the number of devices grows, also growing is their complexity. Already today, the software within something as “simple” as a washing machine can easily exceed 100 000 instructions, as Figure 1.2 illustrates.

To improve the software running inside this growing number of embedded systems, to optimize performance, to minimize power usage, or to evaluate the effectiveness of algorithms in real-world scenarios, developers need deep insight into the software execution. The most powerful way to gain this insight is through on-chip runtime observation, i.e. the software is observed as it executes on the chip.

An ideal observation system provides complete and non-intrusive insight. Complete insight covers all parts of the chip which participate in the software execution. This includes first and foremost the CPU, but, depending on the use case, also the memory, the input/output interfaces, the peripherals, and the communication infrastructure. Non-intrusive observation does not change the function or the timing of the observed software, and hence allows developers to locate even defects which depend on software timing, such as race conditions.

Today, tracing systems are the primary method to gain non-intrusive runtime insight into embedded systems. A tracing system consists of logic added to the chip, and software running on a PC. The on-chip logic captures observation data in the form of trace streams, which are compressed and sent off-chip. Different trace streams can be produced, depending on the part of the chip being observed. The most common ones are instruction (or program) traces (the control flow of a CPU), data traces, and interconnect (or bus) traces.

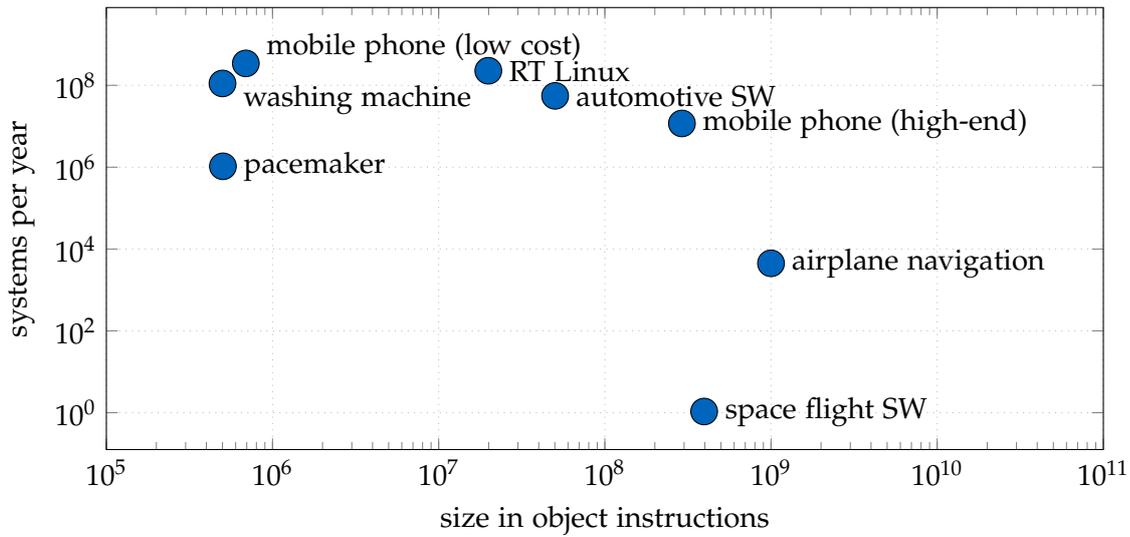


Figure 1.2: Code size in selected embedded systems. Data published in [4] in 2009.

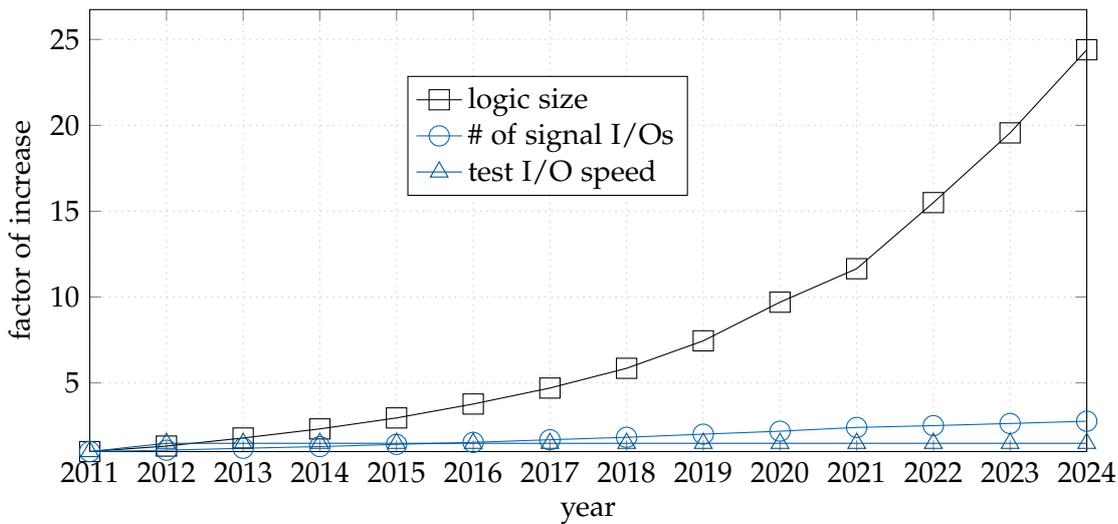


Figure 1.3: Predicted evolution of the logic size, the number of signal I/O pins, and the test I/O speed, normalized to a 2011 baseline, according to the ITRS Roadmap 2013 Edition [5]. Multiplying the number of I/O pins with their speed gives an indication of the available off-chip bandwidth of a chip. The amount of observation data is proportional to the on-chip logic, i.e. the number of transistors in a chip.

Already today, the insight gained through tracing is insufficient in many cases, and it is decreasing. The ITRS roadmap predicts a small linear growth in the available off-chip bandwidth, which does not keep up with the exponential growth of observation data produced on the chip, as Figure 1.3 shows. This gap between the on-chip data a developer would potentially like to observe, and the ability to transfer it off-chip is called “observability gap.”

The observability gap does not only frustrate developers, it also has severe economical impacts due to its direct impact on the development productivity. If developers cannot observe what the software does, they must rely on the observation of effects and guessing—a time-consuming and error-prone approach. A study [6, p. 27] found that in the development of embedded systems reducing the time to market and increasing the quality are seen as the most important challenges. In both challenges debugging and testing play a key role. Even though, as Jones et al. note, “there is a significant shortage of reliable quantitative data on testing efficiency, [and] testing costs” [7, Ch. 5], the available studies can provide insight into the dimension of the problem.

In a study [8, p. 23] for the National Institute of Standards and Technology (NIST) in the United States in 2002 the authors estimate that inadequate infrastructure for software testing costs the U.S. economy US\$ 59.5 billion per year, or 0.6 percent of its gross domestic product (GDP). 40 percent of this cost is borne by software developers (the rest is borne by the users of the software working around the software flaws). This number could be reduced by a third, or US\$ 22.5 billion every year, with improved testing and debugging infrastructure. A study done by Hailpern and Santhanam [9] in the same year found that “in a typical commercial development organization, the cost of [...] debugging, testing, and verification activities can easily range from 50 to 75 percent of the total development cost.” This number has not significantly changed over the decades [10]. For example, Beizer [11, p. 1] summarizes various studies in 1990 by “testing consumes at least half the labor expended to produce a working program.”

To overcome the observability gap, prior work has primarily focused increasing the compression rate of trace streams. However, the efforts in this regard have shown diminishing results in recent years. We take a different approach: We argue that insight is proportional to the questions answered about the software execution, not to the amount of trace data sent off-chip. For most developers, “having full insight” means the ability to obtain an answer to any question he/she has about the software execution—it does not mean access to all possible observation data. In other words, developers are interested in information, not in data.

Based on this idea we aim to create a system that answers similar questions about the software execution as tracing does today, but without being limited by the off-chip bandwidth.

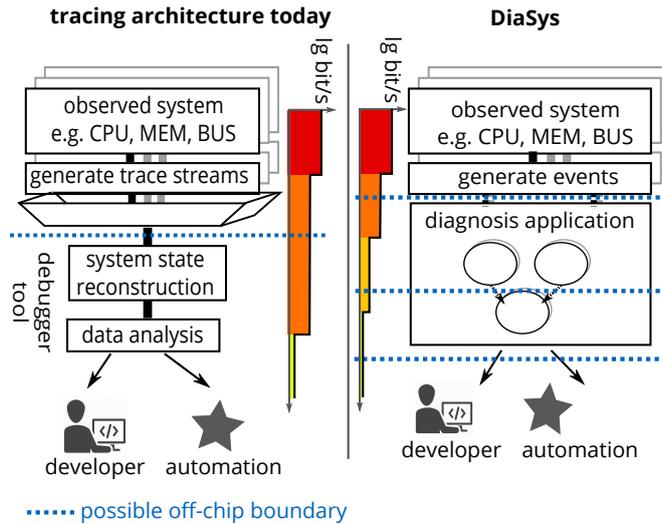


Figure 1.4: Architecture of today’s tracing systems (left) compared to DiaSys (right).

1.2 Contributions

In this work we show that the insight into embedded software can be increased through on-chip analysis of observation data. As realization of this idea we present DiaSys, a method and a tool which non-intrusively collects and analyzes observation data in-situ, i.e. directly on the chip.

We evolve this idea further with three core contributions.

- We contribute a hierarchical multi-step data analysis system with a flexible off-chip boundary as a way to trade off on-chip logic with off-chip bandwidth. We propose to collect and iteratively reduce observation data close to the source, which is enabled by modeling the data analysis process as dataflow application (c.f. Figure 1.4).
- Our system architecture is distributed and composable, built from programmable or configurable components. The data collection components incorporate architectural knowledge and are highly selective to be able to capture also high-volume observations like data/memory traces.
- We introduce a high-level programming language to describe the data collection and analysis as a “script,” making the description reusable, shareable, and composable. Composability creates symbiotic effects when developers from different areas of the hardware and software design contribute knowledge about expected and unexpected behavior within their area of expertise.

To realize DiaSys we have designed and implemented three components.

- *The Dia Language*, a domain-specific programming language to express the collection and analysis of runtime observation data. A script written in the Dia

1 Introduction

Language describes what data to collect when and where, and how to process, combine, or filter it to create meaningful information. Dia scripts are target independent, i.e. the description is portable across different chips.

- *The Dia Compiler*, a tool which transforms a Dia script into a representation suitable for execution on the Dia Engine.
- *The Dia Engine*, an execution platform consisting both of on-chip components, and software components running a host PC. The hardware of the Dia Engine consists of a set of standardized components which can be distributed across the observed chip. Next to data collection components we also include a freely programmable data analysis processor. The software is equally flexible and scalable.

In this work we show that DiaSys can

- be implemented with reasonable hardware and software cost which scales linearly with the observed system,
- be used instead of a tracing system in its two main application areas, the generation of runtime statistics (profiling), and in debugging/testing scenarios,
- reduce the off-chip bandwidth in common application scenarios as compared to today's tracing systems.

A note on terminology: We use the term “software diagnosis” for the combined task of data collection and analysis, and the name DiaSys originated as short form of “diagnosis system.”

1.3 Thesis Outline

This thesis is structured as follows. Section 2.1 presents background information on how software is developed, with a particular focus on the application areas of software diagnosis: debugging and testing. The current state of the art in the collection and analysis of observation data (not only in embedded systems) are presented in Section 2.2 and Section 2.3, respectively.

Based on a solid understanding of the state of the art we present our DiaSys methodology in Chapter 3. The three main components, the Dia Language, the Dia Compiler, and the Dia Engine, are then discussed in Chapter 4. These chapters also include details on the design decisions, the implementation, and its evaluation.

In Chapter 5 representative use cases show DiaSys at work and evaluate various aspects of its design.

Chapter 6 concludes the thesis and gives an outlook on future work.

1.4 Earlier Publications

Subsets of the work presented in this thesis have been previously published in the following peer-reviewed conference and journal papers by the author of this thesis.

- The motivation for DiaSys and its initial idea was presented first in [12] and later in [13].
- The the theoretical foundations presented in Chapter 3, as well as initial versions of Case Study I (Section 5.2) and III (Section 5.4) have been published in [14] and [15].

DiaSys was partially developed within the collaborative project “SoC Doctor,” funded by the Bayerisches Staatsministerium für Wirtschaft und Medien, Energie und Technologie (StMWi). While DiaSys focuses on the description and execution of diagnosis applications on embedded systems, the author of this thesis also closely collaborated with Lin Li at Infineon to explore algorithms and defect patterns which could benefit from the features DiaSys offers. This work was published in [16] and [17].

2 Background and State of the Art

This thesis describes an approach to improve software diagnosis on embedded systems. Before the subsequent chapters go into the details of our method this chapter sets the scene, starting with background information on how software is developed. The chapter then goes on to discuss work related to our approach in two areas: the observation of software execution on a chip, and the analysis of said data to create information which is useful to a developer.

2.1 Background: How Software is Developed

DiaSys performs software diagnosis, the process of obtaining and analyzing runtime observation data from software executions. Software diagnosis is performed in multiple areas of software development. A knowledge of these application areas is essential to understand the requirements and constraints placed on DiaSys. In the following we give a brief introduction into this topic. For a more in-depth discussion we need to refer the reader to the wealth of literature on the topic and its various sub-topics.

2.1.1 Software Development Methodologies

To reliably produce software, various development methodologies have been created.¹ Depending on the size of the project such methodologies can be implicit (e.g. if only single developer works on a hobby project), or very formalized (e.g. when a large team works on a complex project over a long time). Software development methodologies structure a development effort from the definition of requirements until the delivery (and possibly operation) of the software, and guide the team through the design, implementation, and test phases. Multiple methodologies exist and are used in parallel, catering the needs of different development environments and teams, and incorporating “lessons learned” over the last decades.

As a first rough classification, software development methodologies can be grouped into pre-agile, or “heavyweight,” and agile, or “lightweight,” methodologies. Agile methods started to become available in the mid-1990s, but gained more following in the early 2000s.

Pre-agile methodologies are characterized by a linear, top-down, and document-driven approach. Each step in the methodology refines the requirements down to implementable pieces, which are then integrated and tested to ultimately form a

¹Instead of *methodology*, the terms *process*, *method*, or *life cycle* are sometimes used with meaning differing between authors. For the purpose of the discussion in this work a strict differentiation is not essential.

2 Background and State of the Art

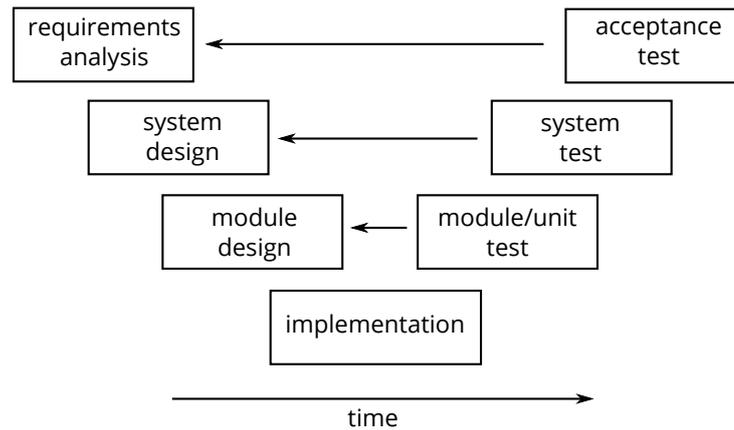


Figure 2.1: The V-model software development methodology.

working software product. Notably, pre-agile methodologies create a working product only at the end of the development process, when all requirements have been covered.

The most prominent examples are the waterfall model and the V-model, both of which are vaguely defined and of which multiple variations exist. Figure 2.1 shows one representation of the V-model. The general “V” structure stems from the match between design and testing phases. Each testing phase (on the right) validates that the implementation fulfills the goals specified in the corresponding design step (on the left). Starting from the requirements (as defined by the customer) the design is iteratively refined in multiple design steps. The result of each design step is documented and used as input for the subsequent step. After each step, the results are considered “fixed,” i.e. they should not be changed. It is therefore necessary to carefully execute the individual steps in depth to avoid problems in later steps of the methodology.

The sequential structure of pre-agile methodologies makes them well-suited for projects with clearly defined, fixed requirements and short project durations. Otherwise, the risk is high to create a product which matches the requirements, but not the needs of the user (anymore).

In the light of raising complexity in software projects and decreasing time to market requirements agile methods arose, which reached a wider audience with the publication of the “Agile Manifesto” in the year 2001 [18]. While agile methodologies differ significantly in detail, they are all characterized by an iterative approach (loop structure), a close collaboration between stakeholders (developers, testers, customers), and a focus on “[w]orking software over comprehensive documentation” [18].

Popular representatives of agile methodologies are Scrum and Extreme Programming (XP). Figure 2.2 illustrates how software is created using the Scrum framework. The requirements in a Scrum project are collected and prioritized in a product backlog by the Product Owner, who represents the customer in a cross-functional Scrum team. Teams consist of typically seven people, including the Product Owner, a Scrum Master (a person who removes roadblocks for other team members), programmers, testers, and other specialists (depending on the project). From the product backlog items are

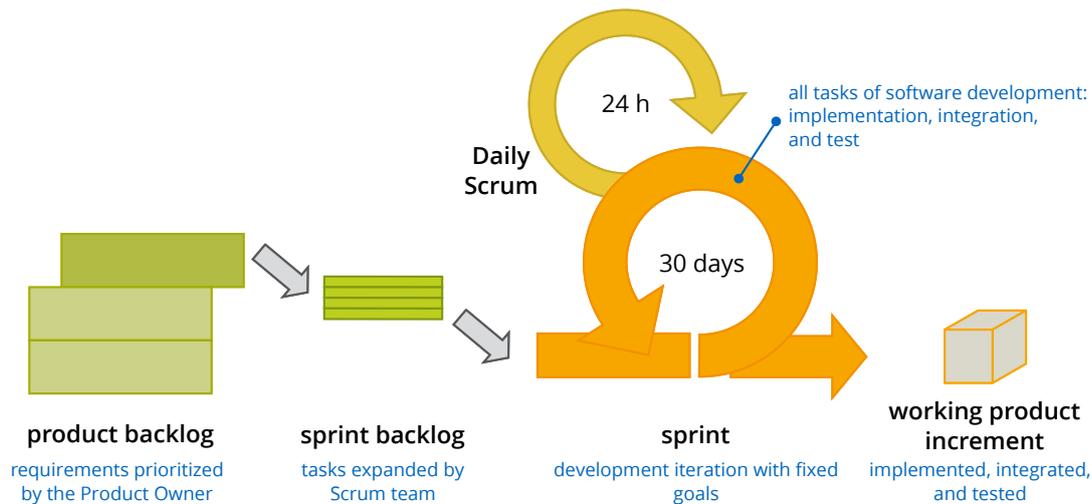


Figure 2.2: The Scrum software development methodology. Figure adapted from [19, p. 8].

taken and implemented during a “sprint,” an interval of typically four weeks. During a sprint the development team is self-organizing. It splits the work into smaller chunks, the status of which is discussed in a short daily meeting (“Daily Scrum”). The result of each sprint is a working product, which could (theoretically) be delivered to the customer, and which iteratively gains functionality as the product backlog is reduced. All steps required to create a working product are performed repeatedly within a Scrum cycle, including design, integration, and various forms of testing.

All software development methodologies contain phases in which code is produced, and phases in which this code is tested to ensure it works and fulfills the design goals. For the purpose of this thesis we are most interested in the phases where insight into the code is required, since that is when software diagnosis is needed. During the development (i.e. “code production”) phase software diagnosis is performed as part of the debugging process. Even more insight into the software execution is needed during the testing and (possibly) verification steps. The next sections describe these three topics in more detail.

2.1.2 Debugging

Debugging is a process in the field of software diagnosis which is performed by a programmer (coder) while implementing software. It is performed to understand the (mis-)behavior of software with the goal of ultimately removing a defect [10, Ch. 8].

Zeller describes the debugging process according to the “TRAFFIC” scheme [20, pp. 5 and 20]. (Similar, yet slightly different, descriptions are presented by other authors, e.g. [10].)

1. Track: Create an entry in the problem database.
2. Reproduce: Reproduce the failure.

2 Background and State of the Art

3. Automate: Automate and simplify the test case.
4. Find origins: Follow back the dependencies from the failure to possible infection origins.
5. Focus: If there are multiple possible origins, first examine the most likely ones using known patterns.
6. Isolate: Use scientific method to isolate the origin of the infection. Continue isolating origins transitively until you have an infection chain from defect to failure.
7. Correct: Remove the defect, breaking the infection chain. Verify the success of your fix.

The “isolate” step is the central and often most time-consuming step in the debugging process.² To isolate a defect developers need to form a hypothesis (e.g. “The problem might be caused by x being below 0.”), and use data obtained at runtime to confirm or refute it. This approach is also known as “scientific method,” the scheme followed by researchers and others to systematically acquire knowledge. It starts by asking a question. Then a hypothesis is formulated, and a prediction about the expected outcome is made. The hypothesis is then tested by conducting an experiment, which can confirm or refute the hypothesis (or have no outcome). Finally, the obtained results are analyzed and integrated into the knowledge base of the scientist (or, in our case, the programmer).

Hypothesis tests, as performed during debugging, are a form of software diagnosis, and a key application area of DiaSys. Even though the scientific method is likely followed by most advanced software developers, few do so explicitly. Most questions and hypotheses are never written down or even explicitly formulated, and many tests do not follow the strict standards required for scientific experiments (especially regarding reproducibility). Typically, hypothesis testing (and debugging overall) is a weakly standardized, manual, and developer-driven process.

2.1.3 Software Testing

Software testing tries to show that software works as intended, or, as Kaner defined it, “[s]oftware testing is an empirical technical investigation conducted to provide stakeholders with information about the quality of the product or service under test” [21]. The stakeholders in this definition are typically the creator of the software (producer or developer), and the customer (consumer or user). While debugging is done by programmers to get their code to work, testing is performed to validate that the software fulfills certain properties. Implementation and testing are often done by distinct people to have a “second pair of eyes” on the problem.

²As Myers et al. note “Of the two aspects of debugging, locating the error and correcting it, locating the error represents perhaps 95 percent of the problem.” [10, Ch. 8]

Just like debugging, testing is an area of software development which is performed regularly, but has not been put onto a solid scientific foundation. As Myers notes, “testing remains among the ‘dark arts’ of software development” [10]. Much of that comes from the fact that software testing is an empirical “art,” which is performed on many abstraction levels using various tools and techniques. Testing approaches range from reviews and module (unit) tests to system and user tests.

For the purposes of this work the relevant aspects of software testing on embedded systems are the following.

- Software tests are often automated to run them repeatedly and reproducibly.
- Testing is performed for functional and non-functional aspects of the software design.
- Higher level tests often interact with peripherals, which either need to be physically present during the test, or their functionality needs to be modeled. For example, a system level test might require the Ethernet interface to be functional to send and receive data from it. If the interface is not physically present during the test, it can be modeled using a bus-functional model (BFM).

As part of a test software diagnosis is used to gain insight into the execution state of a program, and to assert that it matches the expectations. The mentioned aspects of software testing directly translate into design goals for a software diagnosis approach like DiaSys: support for automation, non-intrusiveness (to test non-functional properties), and the ability to perform tests directly on the target device.

2.1.4 Verification

Verification can be seen as the “formal version of testing.” While software testing is an empirical technique, verification provides stronger guarantees and generality as part of the result. Empirical techniques provide a good trade-off between the effort to make sure a program work as intended, and the confidence into the results—at least for many use cases. In safety- or security-critical domains empirical techniques are not sufficient, however. Verification fills this gap with multiple techniques.

Formal verification provides the strongest guarantees (proofs) on certain properties of the software. However, many properties in today’s complex software (especially for software written in general-purpose languages like C) cannot be fully proven, or the creation of a proof would require an unreasonably high effort. Therefore, formal verification is only rarely used in the development of software, with some notable exceptions, for example the seL4 micro kernel [22].

In those cases *runtime verification* can be a suitable approach. Runtime verification checks certain properties using observation data at runtime. While formal verification can give guarantees in the form “this property is always true,” runtime verification can (in most cases) only declare “this property has never been observed to be false.”

2.1.5 Software Diagnosis Needs a Toolbox

The preceding sections have shown the need for diagnosis techniques at various points in the software development process. However, a “one size fits it all” solution to software diagnosis does not exist. Just like a craftsman uses a toolbox full of diverse tools, ranging from general-purpose tools like screwdrivers and pliers to custom-made tools for a single job, software engineers use different tools for software diagnosis at different times.

DiaSys, the diagnosis system we present in this work, is tailored towards non-intrusive runtime observation of embedded systems. It is a powerful tool to have in the toolbox of software diagnosis, but it is not the only one. In fact, with the power of runtime observation comes a certain amount of inherent complexity and some limitations, which sometimes make it the “last resort” in diagnosis. Other forms of software diagnosis make different trade-offs and can be the more productive solution for a given problem. As so often, developers are most productive if they can choose “the right tool for the job.” To describe the environment that runtime diagnosis is best used in, and consequently, areas for which it is not designed, we present an overview of other software diagnosis methods.

A first important category are static analysis tools. Static analysis approaches use (as the name suggests) no runtime information, i.e. they analyze the program (typically its source code) without the actual data inputs. Using techniques like symbolic execution static analysis can produce proofs and argue for all possible execution paths. The obtained results are typically more general, at the cost of higher analysis complexity. Depending on the input (e.g. the programming language the source code is written in), and the question asked, static analysis tools might not be able to produce an answer. Static analysis is especially beneficial if the target system (e.g. the chip) is not (yet) available, since the analysis can be performed nonetheless.

Apart from static analysis many forms of runtime analysis are available, most of which are intrusive and/or work in simulation or emulation (as opposed to “directly on the target chip”). These tools work at different levels of abstraction and make different trade-offs between ease of use, performance, and accuracy of results.

The probably most-used technique to gain runtime insight is code instrumentation. In its simplest form instrumentation is performed by adding code to the program source code, e.g. `printf()` calls in a C program. More advanced approaches exist as well, e.g. library preloading (a way to overwrite calls to external software libraries), or binary instrumentation (as performed by tools like DynInst [23]). Instrumentation can also be used to create an intrusive tracing system. A prominent example is the `strace` tool on Unix-like platforms, which traces (i.e. creates a log) of all system calls performed by an application.

Another often-used intrusive approach to gain runtime insight are run-control debuggers such as the GNU Project Debugger (GDB) or the Visual Studio Debugger. These tools allow a developer to set breakpoints, a point at which the program execution is interrupted. The debugger then hands the control over the application to the developer, who can now inspect the current program state, and re-start the execution.

To conclude, software diagnosis can be performed in many different ways. Depending on the problem at hand and his or her personal experience, a developer chooses a tool suitable for the job. In this work we focus on diagnosis techniques which are non-intrusively performed at runtime. Techniques like static analysis, instrumentation, or run-control debugging do not fulfill these basic requirements; we see them as orthogonal to our approach.

2.1.6 **Summary: Software Development and Diagnosis**

Software development is a large field of research and of even more empirical knowledge. A first structure is provided by software development methodologies. They can roughly be grouped into pre-agile (heavyweight) and agile (lightweight) approaches. Examples for heavyweight approaches are the waterfall and V-models; on the agile side Scrum and Extreme Programming (XP) are well known. All methodologies cover code production (implementation) and testing/verification; software diagnosis is a sub-process in both phases.

During the implementation phase, developers use software diagnosis as part of the debugging process. Debugging is a highly subjective task which very much depends on the experience of the human developer. A productive tool in such an environment is easy to use and does not require substantial effort to set up or maintain. Even if parts of a debugging session are automated little re-use happens: most debugging tasks are one-off jobs.

In testing re-use is essential. Tests are written once and executed repeatedly, often as part of a continuous integration system without human interaction. For testing use software diagnosis tools must be automated, and reliable (to find potential problems in the software execution, not fluctuations in the test execution). Especially in system tests where complex functional and non-functional aspects are tested, tests benefit heavily from running them on the target embedded system. Non-intrusive behavior makes it possible to easier test for non-functional properties, such as the execution time of parts of the code.

This concludes the first section of this chapter, in which we looked at software development in general. This background information helps to position non-intrusive runtime observation within the development flow, and explains some requirements and limitations we place on our work.

We now continue to explore the state of the art in the two main pillars DiaSys stands on: non-intrusive software observation in embedded systems, and analysis of said data for the purposes of software diagnosis.

2.2 **Non-Intrusive Software Observation**

For quite some time now Central Processing Units (CPUs) have been significantly smaller than the human programmers writing software for it (dwarfs included; even a developer's hair is thicker than the structures he or she wants to observe). To

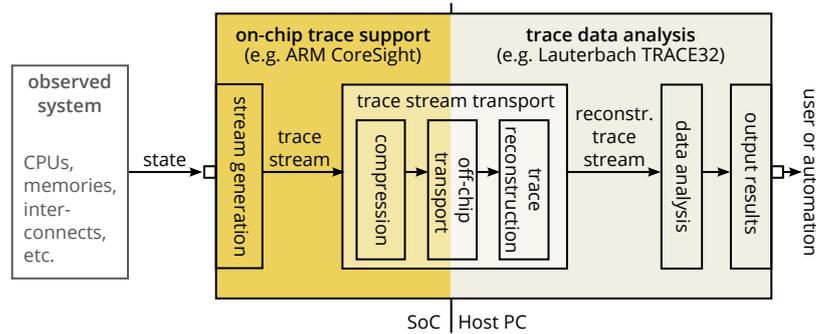


Figure 2.3: A schematic view of a common tracing system like ARM CoreSight or NEXUS 5001.

counteract this mismatch in size chips contain dedicated hardware to give insight into the software execution. In this work we focus on general-purpose, non-intrusive software observation. Two mechanisms provide this kind of insight in today’s chips: tracing and performance counters. We discuss both approaches in the following, including commercial and academic approaches.

2.2.1 An Overview on Tracing Systems

Tracing is the most powerful technique to gain insight the software execution on today’s embedded systems. The basic concept is simple: the software execution is observed on the target device using dedicated logic, and the observation data is sent to an external observer, e.g. a developer sitting in front of a PC. There the trace data can be viewed, or analyzed by dedicated software tools. For example, a developer can step through the recorded program flow, or the tool can calculate metrics out of the data pool, such as code coverage metrics or profiles.

In practice, the achievable insight into the chip is limited by the available off-chip bandwidth: far less data can be transported to the external observer than what is collected at the same time. As Figure 1.3 on page 3 showed, this problem is not getting better any time soon. As mitigation three mechanisms are employed (typically in combination): on-chip buffering, compression, and data selection (also called “trace qualification”).

On-chip buffers bridge the mismatch between the rate at which observations are generated, and the rate at which they can be sent off-chip. Observations are temporarily stored on-chip, and then gradually sent to the host PC at a lower rate. However, this mechanism limits the observation duration. With typical buffer sizes in today’s System-on-Chips (SoCs) in the range of kilobytes to megabytes, observations are limited to typically multiple seconds. Additionally, on-chip buffers are expensive, in that they consume significant amounts of logic area, which in practice places a limit on their size.

A standard way to reduce the size of a data stream is compression. Tracing systems assume that the developer has access to the static program information, i.e. the program binary. Hence, all static information can be seen as redundant and is removed from the trace stream. Depending on the compression scheme, temporal redundancy is also

removed, for example using run-length encoding. The achievable compression rates depend on the implementation and the workload (i.e. the observed software). Program trace compression available in commercial solutions typically requires 1 to 4 bit per executed instruction [24, 25], while solutions proposed in academia claim compression ratios down to 0.036 bit per instruction [26]. Even though data traces contain in general no redundancy, in practice compression rates of about 4:1 have been achieved [24].

The probably most significant way of reducing a trace data stream is data selection: do not collect data which is not interesting to the problem at hand. To facilitate data selection tracing systems typically provide two mechanisms: filters and triggers. Triggers perform temporal selection, they allow the developer to specify when in the execution flow the observation should start, and when it should end. For example, a trace stream can be generated only if a certain function is called (as represented by a program counter). Filters perform spacial selection by defining the data which is collected. Should the trace only contain function calls? Or only executed instructions from CPU 7? With filters developers specify such selection criteria to reduce the data stream to what they are interested in.

All major commercial SoC vendors offer tracing solutions based on this template. ARM provides its licensees the CoreSight intellectual property (IP) blocks [27]. They are used in SoCs from Texas Instruments, Samsung and STMicroelectronics, among others. Vendors such as NXP (formerly Freescale) include tracing solutions based on the IEEE-ISTO 5001 (Nexus) standard [28], while Infineon integrates the Multi-Core Debug Solution (MCDS) into its automotive microcontrollers [29]. Since 2015 Intel also includes a tracing solution in their desktop, server and embedded processors called Intel Processor Trace (PT) [30]. The main differentiator between the solutions is the configurability of the filter and trigger blocks.

In the following we take a closer look at the major commercial tracing implementations.

2.2.2 A Closer Look: ARM CoreSight

The probably most widely used tracing system is CoreSight, which is primarily a specification published by ARM. It is currently available in its third version, released in 2017 [27]. The specification covers mandatory components of any CoreSight-compliant system, protocol and interface definitions, and system architecture requirements. It covers both run-control debug and trace use cases; for the purpose of this work we limit our discussion to the trace components. Based on this specification different vendors can provide implementations; however, it seems that typically vendors include the implementations provided by ARM itself.

The ARM implementation of the CoreSight Architecture specification is marketed as CoreSight SoC and CoreSight Design Kit. These components can be licensed by OEMs and integrated into chip designs, possibly in addition to custom or third-party component implementations. An overview of all available components is given in [31] and [32].

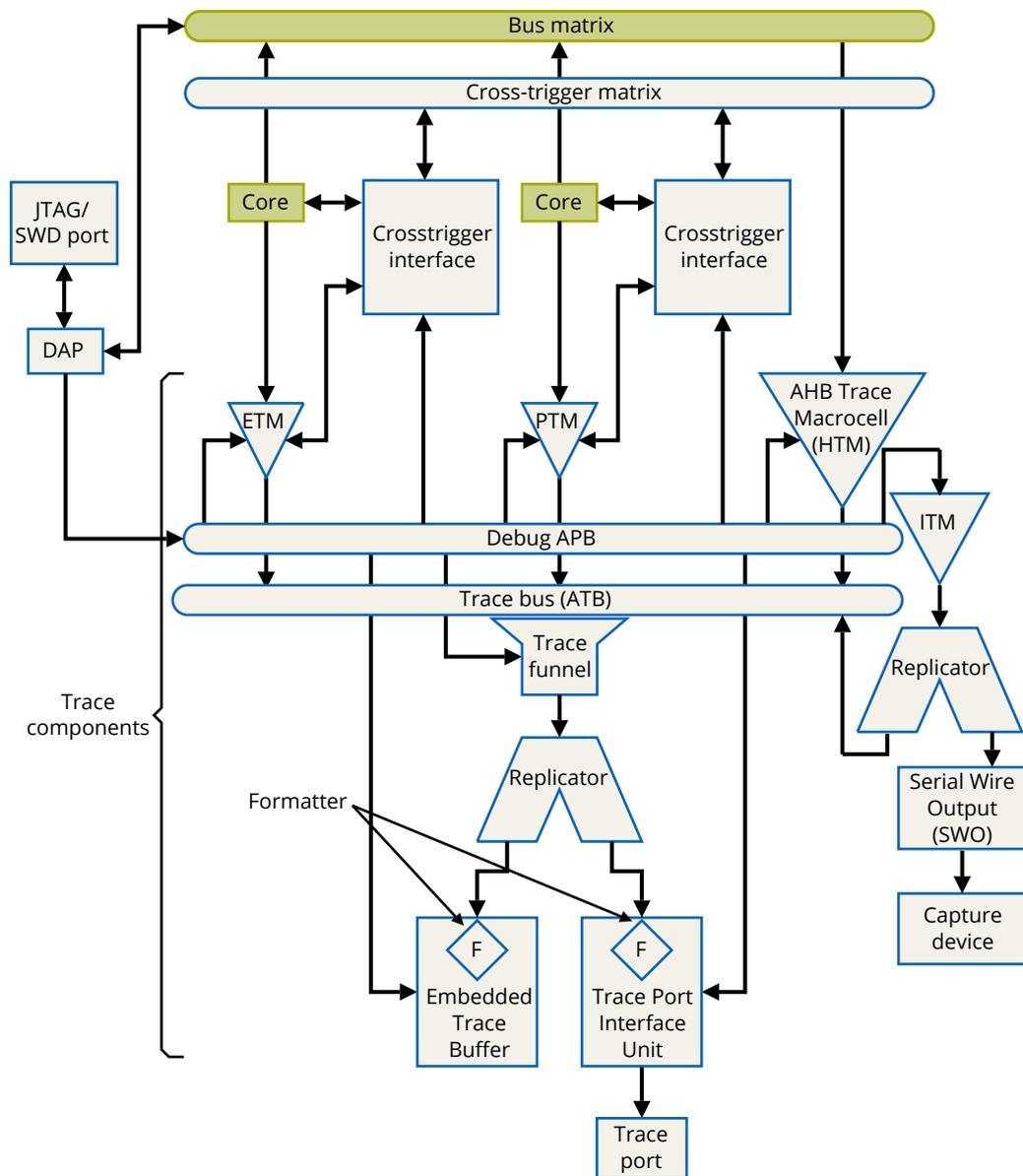


Figure 2.4: An example of the CoreSight architecture for tracing. Drawing adapted from [31, p. 19].

Figure 2.4 gives an overview of a CoreSight system with ARM components. The observed system in this example (green) consists of two CPU cores of different types which are connected through a bus. The CPUs are observed by two CoreSight components which generate an instruction trace (also called program trace): the ETM (Embedded Trace Macrocell) and the PTM (Program Trace Macrocell). Which module is used depends on the CPU family. The HTM (AHB Trace Macrocell) observes the AHB bus, e.g. to observe accesses to the memory or to peripherals. All trace data is sent through a dedicated bus (the ATB trace bus) to a trace funnel, which combines multiple streams into a single one. This stream can then be either temporarily stored on-chip in the Embedded Trace Buffer, or sent off-chip through the Trace Port Interface Unit (TPIU).

In addition to getting trace data off-chip, the tracing system needs to be controlled, configured, and enumerated. For this job the debug infrastructure is re-used. It consists of a low-speed external interface (typically JTAG or SWD), and a dedicated debug bus (APB). In a nutshell, every CoreSight component provides a register-mapped interface which can be read and written through the debug bus. In this way the components can be described (by reading identifier registers), configured (e.g. to set the program counters at which a ETM component should start recording), and controlled. To make it possible for tracing tools on the PC discover all components within a chip CoreSight includes the “reusable component specification” [27]. By implementing a common register set as described in this specification, components can be recognized from the outside, and their internal connectivity can be determined. However, the CoreSight architecture does not require this specification to be followed, and some commercially available chips require the tracing tool to know which components are available.

Triggers and filters can be configured directly with the respective trace modules (such as the ETM). In addition, CoreSight provides an infrastructure to propagate trigger conditions between modules, so-called cross-triggers. With cross-triggers, a trigger condition from one trace module (e.g. the observation of a specific program counter by an ETM module) can result in actions on other debug modules, or the CPU. This can be used, for example, to observe all bus traffic during the execution of a program function. For that to work, an instruction trace module is configured to issue two cross-triggers, one at the beginning and one at the end of a function execution (e.g. based on the observed program counters). The cross-trigger matrix forwards these triggers to the bus trace module (HTM), which activates and deactivates its data recording based on the trigger signals. It must be noted, however, that in asynchronous SoCs no time relationship between the firing and the reception of a cross-trigger is defined; depending on the necessary synchronization stages between the source and the sink the signal can be delayed by multiple implementation-defined cycles.

The most commonly used feature in any tracing system are program or instruction traces, for which ARM provides the ETM and PTM CoreSight components with their CPU core offerings. Four versions of these components are available [33]. The first two versions, ETMv1 and ETMv2 were used prior to the introduction of CoreSight. ETMv3 is the first version compliant with CoreSight, and still used widely today for lower-powered devices (like the Cortex M and R series). As seemingly intermediate

step ARM briefly decided to rename ETM to PTM and the corresponding protocol to PFTv1 (program flow trace). While ETMv3 supports instruction and data traces, PTM only supports instruction traces. As of now, the PTM interlude lasted only for the A9, A12 and A15 processors. The latest version, as of today, switched back to the previous naming and is known as ETMv4. With ETMv4 data traces made a comeback and are now available on the high-performance CPU cores such as Cortex R7, A53, and A57. In addition to supporting newer CPU cores (and their instruction set), the compression ratio improved with newer ETM/PTM versions [34].

Public information about implementation characteristics such as performance and area usage are rare, given that CoreSight is a commercial closed-source implementation. For Cortex R processors, the ARM processor family for real-time applications, [35] claims an area usage of “~ 40 kGates [NAND 2.1-equivalent] for a full debug and multicore tracing (TPIU + ETB)” at “400Mhz on 65nm LP.” The size of the “CPU trace macrocells are between 10 to 20% of the processor gate count.” For Cortex A series, another document [36] claims similar area numbers, and adds that the trace bandwidth can be “[d]own to 0.3 bit per instruction for non-cycle accurate instruction trace.” On the low-end microcontroller core Cortex M3 [25] claims an area use of 7000 gates for a ETM (v3?) cell and a compression ratio of “~ 1 bit/instruction/CPU” and adds that a “data trace from an ARM ETM typically requires 1-2 bytes/instruction.”

Overall, CoreSight has seen steady development since its introduction in 2004. It is an extensible, complete, and efficient debug and trace solution, which is used by many ARM-based SoCs. A large ecosystem of tools from different vendors provides choice on the software side.

Even though CoreSight is in theory vendor-independent, the licensing model of the specification and the implementation do not seem to make it a viable option for non-ARM SoCs.³ For this market other tracing solutions have been developed, especially Nexus 5001, which is discussed next.

2.2.3 A Closer Look: Nexus 5001

Nexus 5001 is a vendor-independent standard for debug and trace on multi-core microcontrollers, with a focus on the interface between the host and the chip.

The work on what later was called Nexus 5001 was started in 1998 by Motorola and Hewlett Packard (as the companies were called back then) [37]. The consortium grew, and by 1999 the first Nexus 5001 standard was published by as IEEE-ISTO 5001-1999. Since then the standard has been revised in 2003 [38] and 2012 [28].

The Nexus standard describes a common, high-level data exchange format, and a choice of port interfaces to connect a chip with a host. To give chip vendors flexibility when implementing a debug and trace solution based on Nexus, the available features are placed into four classes. In Class 1 are basic features for device identification, and run-control debug. Class 2 contains features for program traces, which are extended

³No information about the patent and license situation for implementing the CoreSight specification seems to be publicly available.

with data traces in Class 3. Class 4 extends the data trace functionality with memory substitution features, which essentially redirects memory accesses from the internal memory to the host PC. For each feature certain protocol messages, so called “Public Messages” are defined. For program and data trace messages the standard also defines a simple compression scheme.

In addition to the protocol the Nexus 5001 standard also discusses debug ports, transport mechanisms to connect Nexus-enabled chip with a PC (or an intermediate device). Three categories of ports are supported: dedicated parallel ports (“parallel AUX”), test access ports (TAPs), and high-speed serial ports (“serial AUX”). To enable interoperability, all devices must support a TAP (either IEEE 1149.1, a.k.a. JTAG, or IEEE 1149.7, a.k.a. “compact JTAG”). While the low-speed TAP is sufficient for Class 1 operation, including run-control debug, a higher-speed port is needed for transmitting traces. For this purpose the auxiliary port (AUX) has been introduced; both parallel and serial options exist.

Despite being a vendor-independent standard, Nexus 5001 is too loosely defined to provide “out of the box” interoperability between implementations. Notably the Nexus 5001 standard calls for a significant amount of vendor-dependent functionality, without providing extensive ways of self-description. Additionally, no application programming interface (API) on the software side is defined, and hardware vendors sometimes deviate from the standard in incompatible ways.⁴ Hence, software tools need built-in knowledge about the protocol variant spoken by specific device they are connected to.

The Nexus Forum, the group of authors behind the standard, does not provide IP implementing the standard. Even though some companies (such as HDL Dynamics led by Neil Stollon, a principal author of the specification) are known to provide Nexus-compliant IP, it is unclear if the implementation between chip vendors is (at least partially) shared. This opens the door to further incompatibilities due to implementation bugs or missing features. Furthermore, a compliance test suite does not seem to exist.

Just like for CoreSight, public area and performance numbers for Nexus 5001 are rare. A visual inspection of a die photograph of one of the first Nexus implementations on a Motorola 32 bit MPC565 microcontroller estimates the size of the debug and trace unit to be 0.6 mm² in a 180 nm technology node. The device supports Nexus Class 3, i.e. program and data traces [39, p. 897]. This equals, for comparison, the size of a floating point unit (FPU) or two 4 kB static random-access memories (SRAMs) [24].

Hopkins and McDonald-Maier report in [24] a compression of program traces compared to their raw size to between 9.5 and 11.5 percent (depending on protocol options), which corresponds to roughly 3 bit/instruction. For data traces, their results “indicate that the encoding for the Nexus options had almost no effect on the trace size” [24].

⁴For example, the reference manual of the MPC565 microcontroller notes: “The READI [the implementation name of the debug and trace unit] registers do not follow the recommendations of the IEEE-ISTO 5001 - 1999, but are loosely based on the 0.9 release of the standard.” [39, p. 904]

2 Background and State of the Art

Absent from the Nexus standard are methods for trace qualification, i.e. filters and triggers, including cross-triggers. However, implementors like NXP add custom, non-standard mechanisms to fill this gap.

Today, Nexus-based trace implementations are mainly found in non-ARM devices for automotive applications and (rotating) hard drives [37]. Even though the vendor-independent nature of Nexus makes it well-suited for increasingly heterogeneous chips which integrate components from different vendors, this potential remains to be fully used.

2.2.4 A Closer Look: Infineon MCDS

In its Aurix microcontroller family Infineon includes debug and trace support based on its On-Chip Debug System (OCDS), which is extended by the Multi-Core Debug System (MCDS). Aurix chips mainly target the automotive powertrain market. This market requires high reliability and real-time features, but is also cost-sensitive. Therefore Infineon does not include the advanced tracing functionality in all its chips, but only in a special version of it. The normal chip (called “Production Device”) only contains the OCDS-based basic debugging functionality. Additionally, an “Emulation Device” is produced, which contains the die of the production device together with a further die in a pin-compatible package. The additional die, the “Emulation Extension Chip (EEC),” contains the MCDS functionality, and trace memory in the megabyte range [40].

MCDS is a modular system, consisting of modules to observe CPU cores (POB, “Processor Observation Block”) and the bus (BOB, “Bus Observation Block”). Data traces can be collected through the BOB component. Further auxiliary modules perform timestamping of messages, and other utility tasks. Traces are typically recorded in the trace memory, and once the observation has finished, sent off-chip through a low-speed interface (e.g. IEEE 1149.1, a.k.a. JTAG). Some devices also include high-speed interfaces to stream traces off-chip.

MCDS differentiates itself mainly through its advanced mechanisms for data selection (trace qualification) [42]. Figure 2.5 shows an example of the cross-trigger functionality. In this example, trigger events from two CPUs are combined and a complex trigger is built using a counter element. In this way, trigger conditions can be produced which only fire on the n th trigger event, e.g. after a program counter has been executed five times. The combination of triggers from multiple sources can be programmed using a matrix of AND and OR gates, as Figure 2.6 shows. In contrast to ARM CoreSight, the time relationship between trigger events in MCDS is deterministic. Since MCDS is only implemented in fully synchronous chips, no clock-domain crossings with non-deterministic temporal behavior are needed.

MCDS seems to be implemented currently only by Infineon, even though the implementation has been available for licensing through an IP vendor. No public numbers regarding compression performance or resource usage seem to be available.

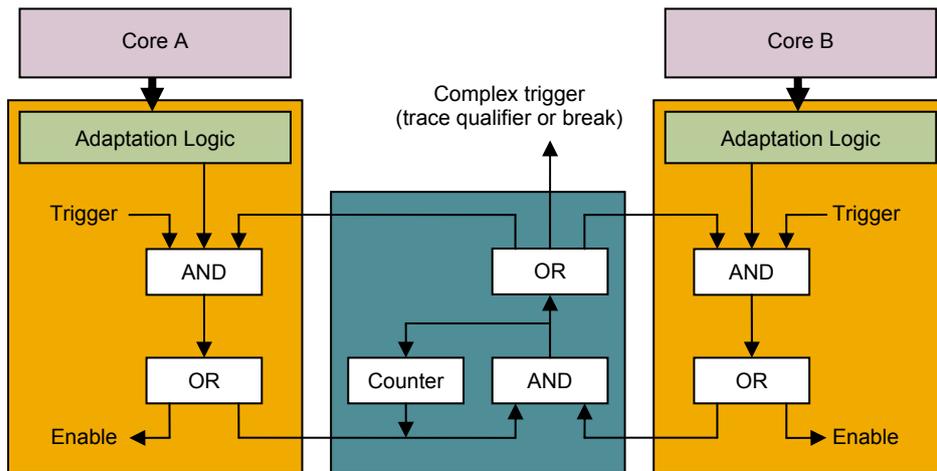


Figure 2.5: Combination of triggers from two sources in MCDS. Graphic from [41].

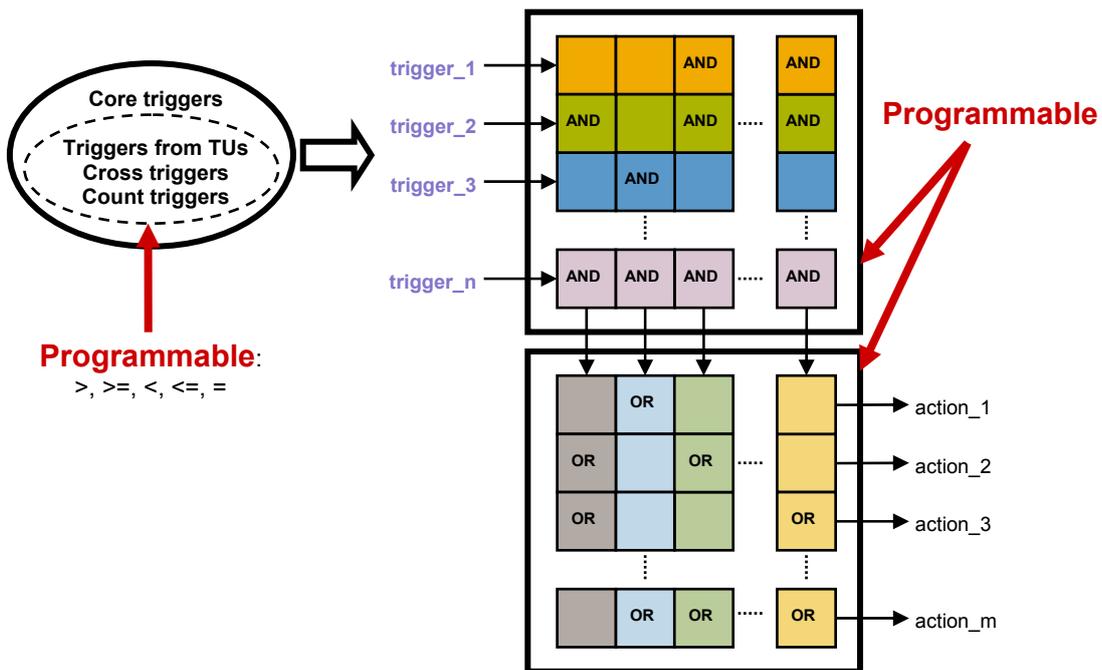


Figure 2.6: Programming of the trace qualification (cross-triggering) functionality in MCDS, as seen from the user side. Graphic from [41].

2.2.5 Discussion: Tracing Systems

Today's tracing systems all follow a common scheme: they observe data on-chip, compress this data with more or less sophisticated compression schemes, combine the streams from individual trace sources, and send the stream off-chip. We presented three commercial implementations, ARM CoreSight, Nexus 5001, and Infineon MCDS. All solutions have a modular structure and been developed and used in products for multiple years now. The main differentiators are the trace compression efficiency, and the methods to select relevant data.

Data selection, also known as trace qualification, is performed through filters and triggers, representing spacial and temporal data selection. Nexus 5001 does not contain data selection mechanisms. CoreSight provides advanced cross-trigger infrastructure also between different clock domains. The MCDS trace qualification mechanisms are most advanced, with support for flexible combination of triggers, and event counters.

Trace compression removes all static information from the trace stream which can be reconstructed on a host PC using the program binary. In addition to the compression algorithm, compression ratios depend on the executed program, the granularity of timestamps, and the overhead introduced by the off-chip interface. Even though no extensive evaluations of trace compression ratios with realistic benchmarks are available, typical numbers for instruction traces are in the range of 1 to 4 bit/instruction. Academic publications claim compression ratios down to 0.036 bit/instruction [26]. However, these numbers are typically for a small set of benchmark programs, or best-case numbers. For an in-depth discussion on trace compression schemes we refer German-speaking readers to [43].

All tracing systems produce heavily compressed traces close to the data source. Uncompressing a trace requires the program binary and significant processing power. Some compression schemes even require an emulation of the functionality of the CPU, since all CPU operations which are not input-dependent (and hence deterministic) are stripped from the trace. While such a compressed trace can be sent off-chip more easily, it cannot be used for on-chip processing, where decompression cannot be performed. The online (real-time) decompression of an instruction and data trace stream in a dedicated device (e.g. an FPGA) is possible (within reason), but requires significant hardware effort, as [44] shows.

Tracing is the most general way of gaining insight into the software execution on a chip, but it is not the only one. Another commonly available feature are performance counters, which are discussed next.

2.2.6 Performance Counters

Performance counters are built-in features of many of today's processors to collect aggregate information about the software execution. In its most general form, performance counters are CPU registers which can be configured to count certain events within the hardware. Almost anything happening on a SoC can be an event: a cache miss, an executed instruction, or an access to a certain memory address are only some

examples. The available events, the number of counters, and other configuration options depend on the CPU implementation. Recent, high-end desktop and server CPUs have shown a trend to integrate more and more performance counter functionality. However, performance counters are not new: already Intel Pentium processors released in 1993 included two performance counters which can count 38 events [45].

As the differences in implementation are small between the different CPU vendors, we limit our discussion to one exemplary implementation: IBM Power. Instruction sets and processor designs of the Power architecture provide extensive support for performance counters through the “Performance Monitor Facility.” The current version of the ISA specification [46, Book III, Chapter 4] describes these. Up to five programmable counters can be configured to count events of a given type. Many types of events can be counted, including various types of executed instructions, interrupts and exceptions, instruction and data cache accesses and misses, address translation events and more. Instead of counting all events of a given type, the data collection can also be randomly sampled. If a counter has reached a programmable threshold, the software can be notified to make use of the data.

In POWER 9 processors, the performance counters can also be used by the In-Memory-Collection (IMC) unit [47]. The performance counter values are grouped by core, thread, or task, and written to memory. The data aggregation is done in software, although the code apparently cannot be modified by end users.

Even though performance counters can typically not be read non-intrusively, their use is minimal-intrusive and gives developers a good way to capture frequent events with low overhead. Today, performance counters are mainly used to analyze and optimize the performance of software. They are not well-suited for other areas of software diagnosis, where the averaged and/or aggregated data hides important information, such as the correlation between events.

2.2.7 Summary: Non-Intrusive Software Observation

In the last sections we discussed tracing and performance counters as ways to observe the software execution in a (mostly) non-intrusive way. Tracing is the most general approach, with tracing hardware integrated into most commercially available SoCs today. Well known implementations are ARM CoreSight, Nexus 5001, and Infineon MCDS. Academic research mostly focuses on the improvement of (program and data) trace compression algorithms.

Another way to gain insight into the software execution are performance counters. These counters are included in many of today’s CPUs and can be configured to count a large amount of events. Since the counting is performed in dedicated hardware, the software execution is not impacted, even at high event rates.

DiaSys takes inspiration from both approaches, as discussed further in Section 3.

2.3 Observation Data Analysis

A human is easily overwhelmed when asked to analyze multiple gigabits of trace data each second. Automated analysis tools perform better in this regard; they can quickly extract useful information out of the vast amount of trace data. Such tools have one common goal: to help a developer better understand the software execution on the target system. The means to achieve this goal, however, vary widely. In the following we present some of these approaches. Approaches used for embedded systems today are typically not very customizable, while systems used for desktop and server PCs are customizable, but intrusive. Still, these approaches serve as a valuable source of inspiration in the design of DiaSys.

2.3.1 Trace Analysis Software for Embedded Systems

Hardware vendors include trace support based on CoreSight, Nexus 5001, or similar approaches, but leave the software side mostly to independent tool vendors. Commonly used tools are TRACE32 by Lauterbach, Universal Debug Engine by PLS, or DS-5 by ARM. The interface between the hardware and software components is the trace, which is first reconstructed using the program binary and other information (c.f. Figure 2.3). While the reconstruction depends on the observed chip (and the used trace compression algorithm), the further analysis is mostly target-independent. Many tools use this abstraction to provide a common user interface for all supported tracing systems, hiding the differences in hardware support from the developer.

Typically, a graphical user interface (GUI) is the main “point of contact” for developers. Through this interface developers can view the decoded trace for manual inspection, or trigger more complex built-in analyses such as the generation of runtime profiles, code coverage reports, or cache access analyses. In addition to the GUI, many tools provide a scripting interface which enables the automation of the user interface functionality (“batch processing”).

The mentioned trace tools can provide the same GUI and re-use code to perform trace analysis because they abstract away the differences between devices. This abstraction works well for the trace stream itself; a reconstructed trace from a Nexus implementation is identical to one from a CoreSight trace system (assuming the same CPU type, of course). However, abstracting away the configuration of a trace system is more challenging. Each tracing system, and each chip, supports different triggers and filters, not to mention complex trigger functionality like counters in MCDS. However, making full use of the data selection functionality a chip provides is vital, as it directly impacts the gained insight into the software execution: the less filtering is applied, the more data must be sent off-chip or buffered on-chip, which either reduces the observation time (if on-chip buffers are used), or prevents observation altogether (if the off-chip interface does not provide sufficient bandwidth).

An approach to make the creation of trigger configurations easier is presented by Braunes and Spallek [48]. Their Trace Qualification Language targets the Infineon MCDS system and can be used to describe events and combine these events in a state

machine to form higher-level events and perform limited actions, such as “record the address currently on the bus.”

To make full use of the provided hardware functionality, some tools integrate more deeply with the target tracing system. For example, the TASKING Embedded Profiler [49] provides insight into reasons for performance bottlenecks by using the advanced features of MCDS. (In fact, this product is based on an Infineon-internal technology demonstration called ChipCoach, to which the author of this work contributed in a joint project.)

To summarize, today’s commercial tools to perform tracing are mostly GUI-based, with optional scripting abilities. They abstract away differences in the trace hardware, which is easier for the trace itself than it is for the trace configuration. The trace analysis (e.g. the creation of a runtime profile) is performed using built-in algorithms, which cannot be changed by the end-user. Scriptable tracing tools, which are not yet common for embedded systems, enable more customization in this regard. We discuss them in the following section.

2.3.2 Scriptable Event-Based Debugging

In run-control debugging, as performed with GDB, a breakpoint is set to halt the program execution at an interesting point in the program flow. Once this breakpoint is hit, the developer takes control and manually inspects the program state. Another name for this debugging approach is “interactive debugging.” Scriptable event-based debugging is the logical continuation of this concept. A breakpoint is now considered an “event,” and instead of manually inspecting the system state developers write small pieces of code which perform this analysis. In essence, the code snippets describe desired or non-desired behavior of the program.

The general concept has been developed since the 1970s, and even though the methods to collect observations were typically intrusive and the used programming languages reflect the choice of languages at the time, this body of work remains highly relevant to our approach today.

One of the first event-based debugging approaches was presented in 1977 by Johnson. With RAIDE [50] debugging scripts can be written which do not depend on the programming language of the source code being debugged. It uses a declarative language to define events, and to write “debugging procedures,” small snippets of code which perform the analysis. These procedures could also be collected in libraries for reuse. Procedures cannot, however, emit further events, and the approach is limited to sequential software.

In 1988 Bates [51, 52] introduces the “Event-Based Behavioral Abstraction” (EBBA), a way to describe the expected program behavior and matching that with observed behavior. It targets heterogeneous, distributed architectures and is especially designed with reusability in mind: for example, a model describing a deadlock situation can be applied to different software to find a deadlock bug. EBBA distinguishes between primitive events, which are generated by the software execution, and high-level events,

2 Background and State of the Art

which model the program behavior by matching primitive events within an event stream using the sequential, choice, concurrency, and repetition event expression operators.

Lumpp et al. present a debugging system which is based on an event/action model [53]. They include a description of a non-intrusive observation system for early parallel computers. The language syntax given in the paper is “currently being developed,” but their syntax example shows the author’s ideas well. An analysis program contains multiple sections: global variables, event definitions, action definitions, bindings of events to actions. Notably, changes to the global variables can trigger further events. The approach has not been implemented, and no work is known which improves on the proposed language syntax.

Dalek [54], presented in 1991, is built on top of GDB and can be used to debug a single, sequential application. It extends the scripting mechanism of GDB with the ability to declare events. Primitive events can be triggered on a breakpoint, and carry data. For example, an event triggered on the call of a function can be accompanied with the first argument to this function. Primitive events can then be further processed by high-level events, resulting in a dataflow-like processing of events. Dalek has been fully implemented (according to its authors), but even though the paper states “[t]he Free Software Foundation plans to incorporate Dalek into a future release of gdb,” this future has not yet arrived.

Event-based debugging has also been implemented in commercial tools, such as the “HP Distributed Debugging Environment” (DDE) [55], a debugger presented in 1994. In this tool custom code can be executed when an event (e.g. a breakpoint or a watchpoint) is hit to decide if the debugger should stop the program execution, or continue.

Coca [56], on the other hand, uses a language based on Prolog to define conditional breakpoints as a sequence of events described through predicates for debugging C programs.

Marceau et al. present MzTake, an interactive scriptable debugger which uses FrTime, a Lisp-inspired dataflow language, to debug Java and Scheme programs [57, 58].

Auguston et al. have continued developing FORMAN [59], into PARFORMAN (“Parallel Foremal Annotation Language”) [60] and a decade later into UFO [61]. It describes an event-based assertion language which is used to specify the expected program behavior, which can then be checked against an actual program execution to detect bugs.

A major inspiration for our work are scriptable or programmable debug solutions which are available on today’s desktop and server machines. Their basic functionality is simple: whenever a defined *probe point* is hit, an event is triggered and an *event handler* executes. The analysis is essentially described as a dataflow program with only a single actor. Common probe points are the execution of a specific part of the program (like entering a certain program function), or the access to a given memory location. The best-known current implementations of this concept are DTrace and SystemTap, which run on, or are part of, BSDs, Linux, and macOS (where DTrace is integrated into the “Apple Instruments” product) [62, 63]. DTrace has been a significant improvement over

```

1 SELECT event-type, timestamp, transaction-id, event-dependent-data
2 FROM R
3 WHERE (event-type = 'message send' or
4        event-type = 'message receive') and
5        process-id = 352 and
6        timestamp > 10:00 AM

```

Listing 2.1: Code example from [64] to query the execution database of a distributed system to obtain all messages sent or received after 10:00 am by process 352.

the state of the art at its time⁵, and is discussed in more detail in Section 2.3.4. Today, the prime scriptable tracing solution within Linux is eBPF/BCC. A limited (and not well defined) subset of C can be used to describe actions, which are then executed in the context of the Linux kernel by a bytecode interpreter. This approach results in efficient execution of action handlers, but much of the infrastructure around it are not yet well-defined or convenient to use.

DTrace, SystemTap, and eBPF/BCC-based approaches all use a C-like language to define event handlers, and custom syntax to define the events. Other works have used SQL or a SQL-like syntax for both tasks, defining events and processing them.

2.3.3 Query-Based Debugging

Query-based debugging models execution traces as database, which can be queried to obtain information about the software execution. Since the advent of SQL in the 1970s using this language to query the “execution database” was a logical choice.

In one of the first works in this regard, Garcia-Molina et al. propose a debugging system which works across distributed nodes [64] (published in 1984). Execution traces from each process are collected and sent to a central server, where they could be analyzed using SEQUEL (as SQL was called back then). Listing 2.1 gives an impression of the query language the authors envision in their approach.

The idea of using SQL has remained popular over the decades. Presented in 2012 by a group from Microsoft Research, Fay [65] describes itself as a “comprehensive tracing platform that provides both expressive means for querying software behavior and also the mechanisms for the efficient execution of those queries.” [65] The authors have realized that when employed on distributed systems “there is little room for optimization in script-based tracing systems such as [...] DTrace and SystemTap.” Therefore, they created a new language, FayLINQ, which can perform data collection and distributed aggregation over multiple machines in a way which can be optimized towards efficient execution.

Another example for a SQL-like observation collection and analysis language is presented by Mace et al. under the name Pivot Tracing [66]. It combines data collection

⁵An indication for the success of DTrace can be seen in the awards it won, including the first place in the Wall Street Journal’s 2006 Technology Innovation Awards, and the USENIX Software Tools User Group award.

2 Background and State of the Art

```
1 syscall::entry
2 /execname == "firefox-bin"/
3 {
4   @num[probefunc] = count();
5 }
```

Listing 2.2: A DTrace script which counts all system calls performed by the Firefox and groups them by system call name.

through dynamic instrumentation with a high-level SQL-like query language, which introduces a new happened-before join operator to “group and filter events based on properties of any events that causally precede them in an execution” [66].

Finally, a tool used today in commercial settings for large fleets of servers is osquery.⁶ Developed by Facebook, osquery maps system information to relational database tables and makes them accessible through SQL queries. For example, the query `SELECT uid, name FROM listening_ports l, processes p WHERE l.pid=p.pid` returns a list of all processes which are currently listening on a TCP socket, including the ID of the user which is running the process.

2.3.4 A Closer Look: DTrace

DTrace [62] is a scriptable dynamic tracing framework for desktop and server machines. Designed by Bryan Cantril, Adam Leventhal, and Michael Shapiro at Sun Microsystems, DTrace has been published in 2003 to diagnose both kernel and user space applications running on Solaris. It has since seen wider adoption and been ported to FreeBSD, macOS, and partially to Linux.

DTrace follows the event-action approach of diagnosis. A user writes a script which defines probe points and associated actions. When a probe point is hit, the software execution is stalled, and the code within the action is executed. The probe points can be either built into the kernel or user-space applications, or dynamically inserted at runtime. Actions are described in a C-like programming language called D. Even though D looks like C in many regards, it is designed with safety in mind: actions written in D cannot crash the system, making the whole DTrace safe to use in production environments. To provide this guarantee, actions written in D must be statically analyzable; all features which hinder this analysis are not part of the language. Most notably, loops are not allowed in D. On the other hand, convenience features are added to the language, such as associative arrays, easier string handling, and aggregations.

Listing 2.2 shows an exemplary DTrace script. The script probes all system calls (`syscall::entry`) and uses a predicate to filter out only system calls issued by an executable named `firefox-bin` (i.e. the Firefox web browser). Given between the curly braces is the action. In our example, the action makes use of the aggregation

⁶<https://osquery.io/>

functionality (@num) to count the number of system calls, grouped by the system call name (which is given in the DTrace-defined probefunc variable).

The success of DTrace can be most likely attributed to three points.

- DTrace is safe to use and incurs little overhead.
- The language has found the a good balance between expressiveness and safety.
- A large number of example scripts exist, collected by Brendan Gregg in the DTrace Toolkit.

Today, DTrace receives less attention. The Solaris operating system (together with Sun Microsystems) is only a shadow of its former glory, and on Linux DTrace never gained wide adoption due to licensing issues. However, it has inspired the design of other approaches on Linux, notably of SystemTap and ktap (even though these solutions do not provide the same safety guarantees as DTrace).

2.3.5 Summary: Observation Data Analysis

The analysis of observation data can be performed in general ways: either, using fixed-function analysis tools, or using scriptable frameworks. In the domain of embedded systems, today the analysis is performed mostly using tools which have certain analysis algorithms built-in. These tools and algorithms can often be lightly customized, but still, the freedom of expression is limited.

A more flexible way to analyze observation data are scriptable approaches. They can roughly be split into two categories: event-action based approaches, and query-based ones. Event-action based approaches evolve the manual run-control debugging approach, in which events are called “breakpoints,” and provide ways to automate the previously manual step, the action that is performed once an event has been issued. First developed in the 1970s, the basic concept has been applied again and again, with DTrace and its successors SystemTap and eBPF/BCC being the current representatives which are in wide industry use. It is likely that this success comes from the simplicity of the concept to typical software developers, which are accustomed to writing sequential code within an “action.”

Query-based approaches gained less traction. These approaches model the observed system(s) as database, which can be queried to collect and aggregate observation data. Typically, the querying is done using SQL or a SQL-like language. With initial works also presented as early as 1984, Fay [65] showed how powerful this approach can be to distribute the data aggregation across a cluster of machines. Mainly for performance optimization work osquery is currently being used in the industry. One can only speculate as to why query-based approaches gained less traction. It seems likely, however, that aggregating SQL queries are unfamiliar to many developers.

2.4 Summary: Background and State of the Art

DiaSys performs non-intrusive software diagnosis at runtime: it observes software running on embedded systems, and analyzes these observations to provide meaningful information to the developer. In this chapter we looked at the environment in which such a system is used, and at work related to our approach.

Observation and analysis, which we combine in the term “software diagnosis,” are part of the larger software development effort. Independent of the chosen software development methodology, be it a pre-agile one like the V-model, or an agile one like Scrum, software diagnosis is needed during both debugging and testing. In debugging, programmers try to understand the program execution with the ultimate goal of locating and resolving a defect in the software. Testing tries to show that a program behaves as designed regarding functional and non-functional aspects.

Debugging is a subjective and mostly manual process performed by a human developer. To be suitable in debugging, a diagnosis system must be flexible, easy to use for a human developer, and require little setup time. Typically, the focus in debugging is on the collection of observation data, while the analysis is performed manually.

On the other hand, effective testing requires automation in the data collection and the analysis. This is especially true today when agile methods like Scrum iterate quickly to produce a potentially shippable product every couple of weeks. In such an environment, the setup cost of automated tests diminishes compared to their execution time, giving an incentive for higher degrees of test automation.

Overall, the discussion of the software development environment showed diverse requirements, which require a flexible general-purpose diagnosis system. We discuss the requirements which guide DiaSys in more detail in Section 3.1.

In order to bring the analysis of observation data into the chip, DiaSys integrates the non-intrusive collection of observation data, and its analysis closer together than existing solutions. We therefore discuss related work from two areas: the non-intrusive collection of observation data from embedded systems, and the (scriptable) analysis of such data.

The first step in software diagnosis is the collection of observation data. For this purpose two main non-intrusive approaches are available today: tracing systems, and performance counters. Most embedded systems today contain a tracing system based on ARM CoreSight or Nexus 5001. They observe functional units like CPUs, memories, or the interconnect, and compress and stream these observations off-chip. The other approach, performance counters, is tailored towards collecting aggregate data of high-frequency events. Such events can be very diverse, ranging from cache misses to executed CPU instructions.

The second step in software diagnosis is the analysis of the collected observation data. Today, observations from embedded systems are typically analyzed with tools with built-in analysis algorithms and limited flexibility. On desktop and server machines, the scriptable analysis has seen wider adoption, with tools such as DTrace having gained wide acclaim in the community.

2.4 Summary: Background and State of the Art

The design of DiaSys is based on a careful study of the related work presented in this chapter. However, no approach is directly suitable to bring the analysis of observation data on-chip. On the observation side, today's tracing systems compress data in a way that makes it unsuitable for on-chip processing, while performance counters are too limited in their functionality. On the analysis side, the well-known event based approaches, like DTrace, use monolithic code blocks as event handler (or action). This prevents the split of the execution between chip and host PC. Older approaches, most notably Dalek [54], provide the ability to describe the event processing as dataflow network. This work can be seen as closely related to our approach regarding the analysis data processing.

Based on a solid understanding of the environment DiaSys operates in, and the related work in the collection and processing of observation data, we present our diagnosis system in the following.

3 The DiaSys Approach to Software Diagnosis

DiaSys has been created to perform on-chip processing of observation data with the goal of increasing the software insight, and reducing the off-chip traffic compared to today's tracing systems. The previous chapter has set the scene by discussing the environment in which DiaSys is employed (debugging and testing), and by discussing related work. In this and the following chapters we present our contribution. We start with a detailed discussion of the assumptions we made, and the goals we want to achieve with DiaSys.

Some of the work presented in this chapter has previously been published in [15].

3.1 Assumptions and Goals

We designed DiaSys with the following assumptions and goals in mind.

For software observation While finding bugs and errors is part of any development process, this work is only concerned with finding functional and non-functional bugs in software running on embedded systems. The SoC itself is assumed to behave as specified, i.e. we are not concerned with hardware bugs or after-production tests.

For embedded systems DiaSys targets embedded systems or SoCs. Even though many concepts could equally be applied to e.g. desktop, server, cloud or HPC environments, we do not further consider these use cases.

We do not place strong restrictions on the type of SoC, or its usage area. We explicitly include SoCs used in hard real-time scenarios, as well as consumer devices which mostly operate in "best effort" mode. We also place no limit on the size of the SoC, ranging from small systems with only one CPU core, to large MPSoCs with hundreds or even thousands of CPU cores.

General purpose The problems a software developer is facing are diverse—and so must be the scope of the diagnosis solution. DiaSys does not target one single type of bug or software problem, but provides a general-purpose (universal, generic) tool to analyze a wide range of problems.

Distributed The diagnosis system must be able to reduce the amount of observation data as close to the source as possible, i.e. close to the observed units. Since the data sources are distributed across the chip, the diagnosis system must also be distributed appropriately.

Non-intrusive The diagnosis system must be non-intrusive (passive). Non-intrusive observation preserves the event ordering and temporal relationships in concurrent executions, a requirement for debugging multi-core, real-time, or cyber-physical systems [67]. Non-intrusiveness also gives a developer the confidence that he or she is observing a bug in the program code, not chasing a problem caused by the observation (a phenomenon often called “Heisenbug” [68]).

Flexible on-chip/off-chip cost split The diagnosis system must be flexible to implement. The implementation of the diagnosis system involves a trade-off between the provided level of observability and the system cost. The two main cost contributions are the off-chip interface and the chip area spent on diagnosis extensions. The diagnosis system concept must be flexible enough to give the chip designer the freedom to configure the amount of chip resources, the off-chip bandwidth and the pin count in a way that fits the chip’s target market.

Relaxed timing constraints The diagnosis system must not assume a defined timing relationship between the individual distributed components. Today’s larger SoCs are designed as globally asynchronous, locally synchronous (GALS) systems with different power and clock domains, where no fixed time relationship between components can be given.

Enable reuse and automation It is rare that a functional or non-functional software issue is truly unique. Hence DiaSys should facilitate sharing and reuse—reuse of knowledge how to diagnose software issues, and sharing of this knowledge across teams, companies, hardware and software generations. Ultimately, we envision that DiaSys enables the sharing of diagnosis applications similar to how a software library and other program code is shared today.

Going one step further is the automated execution of diagnosis applications. If not only the steps to analyze a problem are coded within a diagnosis application, but also the evaluation of results, an automated execution of a wide range of diagnosis applications is possible without human intervention. It has been shown in the past that such automation significantly increases the quality of a resulting product [69].

The assumptions and goals presented in this section guide the design of DiaSys, from the architecture down to the implementation. The design on the highest level is what we present next.

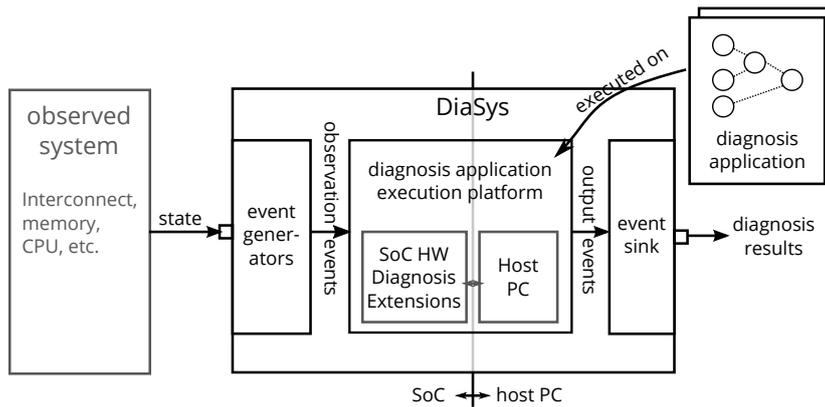


Figure 3.1: A schematic overview on the concept of DiaSys.

3.2 The DiaSys Concept

To perform software diagnosis with DiaSys, multiple components interact in well-defined ways. Figure 3.1 gives an overview of the components which we explain in the following.

The input to the diagnosis system is the state of the observed system over time, the output are the diagnosis results, which can be represented in various forms. The output is created from the input with the help of event generators, the diagnosis application running on an execution platform, and event sinks. Between these components data is exchanged as diagnosis events.

Diagnosis events are the container for data exchanged in DiaSys. In the general case, an event consists of a type identifier and a payload. Events are self-contained, i.e. they can be decoded without the help from previous or subsequent events.

Event generators produce observation events based on state changes in the observed system. Typically, event generators are attached to a single unit in the observed system, e.g. a CPU or a memory. They continuously compare the state of the observed unit with a trigger condition. If the condition holds, they trigger the generation of an observation event.

An observation event is a specialized diagnosis event. The payload contains a partial snapshot of the state of the observed system at the same instant in time as the event was triggered. Which parts of the state are attached to the event is specified by the event generator configuration. For example, a CPU event generator might produce observation events when it observes a function call and attach the current value of a CPU register as payload. An observation event answers two questions: why was the event generated, and in which state was the observed system at this moment in time.

The diagnosis application describes how the observations should be analyzed/processed with the goal of producing higher-level information. We describe diagnosis applications, which are modeled as transformational dataflow application, and their properties in more detail in Section 3.3.

To execute diagnosis applications, DiaSys provides an execution platform. The execution platform can span (transparent to the developer of a diagnosis application) across the chip boundary. On the chip it consists of specialized hardware blocks which are able to execute (parts of) the diagnosis application. On the host PC software components execute of the remaining parts of the diagnosis application. The on- and off-chip part of the execution platform are connected by the off-chip interface. This split design of the execution platform allows hardware designers to trade off chip area with the bandwidth provided for the off-chip interface, while retaining the same level of processing power, and in consequence, system observability.

Event sinks consume output events produced by the diagnosis application. Their purpose is to present the data either to a human user in a suitable form (e.g. as a simple log of events, or as visualization), or to format the events in a way that makes them suitable for consumption by an automated tool, or possibly even for usage by an on-chip component.

Together, event generators, the diagnosis application and the event sink form a processing chain which provides a powerful way to distill information out of observations in the SoC. A key innovation of DiaSys is the use of diagnosis applications to describe the processing of observation data. They are described next.

3.3 Diagnosis Applications

Diagnosis applications are the heart of the diagnosis system, as they perform the “actual work” of interpreting what happens on the observed system during the software execution. Diagnosis applications are transformational dataflow applications. We chose this model to enable the transparent mapping of the diagnosis application to an execution platform spanning across the chip boundary. Our goal is that the developer of the diagnosis application does not need to explicitly partition the diagnosis application into an on-chip and an off-chip part; instead, this mapping can be performed in an automated way. No matter how the diagnosis application is mapped onto the execution platform, the behavior of the application follows identical rules, i.e. the semantics of the application stays the same.

The diagnosis application is a transformational application, in contrast to reactive or interactive applications [70]. This means, starting from a given set of inputs, the application *eventually* produces an output. The application code only describes the functional relationship between the input and the output, not the timing when the output is generated. The application also does not influence or interact in another way with the observed system from which its inputs are derived; diagnosis applications are guaranteed to be non-intrusive.

The structure of diagnosis applications follows the dataflow concept. The complete computation is represented by a directed graph, in which the nodes model the computation, and the edges model communication links. In diagnosis applications we call the graph nodes transformation actors, and the graph edges channels.

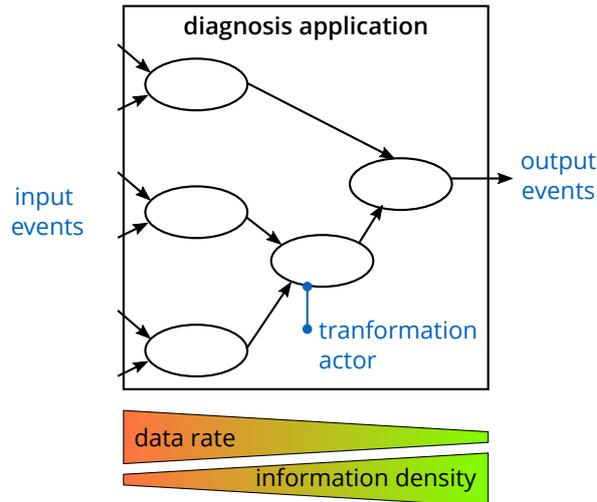


Figure 3.2: A model of a diagnosis application. Diagnosis applications are structured as hierarchical dataflow applications. Across the edges events are sent, which are processed within the nodes (transformation actors). The goal of the processing is the reduction of the event/data rate, which an corresponding increase in information density.

Each transformation actor reads events from $n \in \mathbb{N}_0$ input channels, and writes events to $m \in \mathbb{N}_0$ output channels. The sequence of events consumed on a single channel x is described by a vector of input events $\vec{E}^x = (e_1^x, e_2^x, \dots)$. We call \vec{E}^x , in line with the definitions used by Kahn [71], history of events on channel x .

A transformation actor starts its processing, it “fires,” if a sufficient number of events are available at its inputs. The definition of “sufficient” depends on the individual transformation actor. For example, one transformation actor might always read one event from each input before starting the processing, while another one might always read two events from input 1 and one event from input 2.

When firing, the transformation actor applies an arbitrary transformation function f to the input events. The generated output depends on

- the read input events,
- the ordering of the input events, and
- the internal state of the transformation actor.

Transformation actors may communicate only through the input and output channels, but not through additional side channels (e.g. shared variables). Hence, all state is contained within a transformation actor, and all data exchanged between actors is explicit, which significantly eases the partitioning of a diagnosis application to different execution units. (This is discussed further when we explain the design of the Dia Engine in Section 4.4.)

3.4 Analyzability of Diagnosis Applications

Diagnosis applications built out of such transformation actors are in general nondeterministic, as defined by Kahn [71, 72]. This means, the output not only depends on the history of inputs (i.e. the current input and the state of the actor), but also on the relative timing (the ordering) of events.

Nondeterministic behavior of diagnosis applications is, in most cases, the expected and wanted behavior; it gives its authors much-needed flexibility. An example of nondeterministic diagnosis applications are applications which aggregate data over time, like profiling applications. These applications consume an unspecified amount of input events and store an aggregate of these inputs. After a certain amount of time (e.g. one second), they send a summary of the observations to an output channel.

Unfortunately, it is not possible to perform static analysis of nondeterministic diagnosis applications regarding event rates, bandwidth, or processing requirements. Therefore, if wanted, application authors can create deterministic diagnosis applications, if they restrict themselves to

- always reading the input channels in the same order without testing for data availability first (instead, block and wait until the data arrives),
- connecting one channel to exactly one input and one output of an actor, and
- using only transformation functions which are deterministic themselves.

3.5 Behavior in Overload Situations

Overload situations occur if the observed system triggers the production of more events than the diagnosis system can process. Given the generally unknown input data, and the generally nondeterministic behavior of the diagnosis application, it is not possible to statically dimension the diagnosis system to be able to handle all possible input sequences. Therefore, overload situations are unavoidable in the general (and most common) case. If an overload situation is detected, the diagnosis system can react in multiple ways.

First, the diagnosis system can temporarily stall the observed system. This gives the diagnosis system time to process outstanding events without new events being produced. This approach is only feasible in a synchronous non-realtime system.

A more common approach is to discard incoming data until further processing resources are available. Depending on the diagnosis application, a recovery strategy needs to be formulated. Some applications can deal easily with incomplete input data, e.g. diagnosis applications creating statistics. Others are not able to work with an incomplete input sequence and in consequence fail to be executed properly.

It is up to the implementation of DiaSys to define a suitable strategy to handle overload situations.

3.6 On the Benefits of Programmable Diagnosis

The data analysis in DiaSys is performed by the diagnosis application. While it is possible to create a diagnosis application through a graphical user interface, or hard-code it into a software diagnosis application, we focus in our work on programmable or, as we call it from now on, script-based diagnosis.

The main benefit of a script-based diagnosis approach as proposed in DiaSys is the increase in developer productivity. Developers need to spend less time on repetitive, manual tasks, and have more time to find creative solutions to novel problems. Script-based diagnosis shares work between human and machine in a way which allows everybody to do what they can do best. Machines are best at following a script to the dot, over and over again. And humans are best at inventing, at finding new solutions to new problems.

Splitting work between human and machine in this way is not new, of course. Since the early days of industrialization the concept has been applied repeatedly, profoundly changing our industry and our society along the way.

The increase in productivity comes from increasing automation. The first step towards automation is documentation: writing down the steps which must be taken to diagnose a software problem. As a subsequent step, this documentation must be made machine-readable and machine-executable, i.e. the documentation is now “code,” or as we call it in this context, a “script.” Once a diagnosis script has been written it can be executed repeatedly, making it easier to run diagnosis scripts more often, unattended, and to create reproducible results.

To better understand the benefits and limitations of automation we can have a look at a widely studied example close to our topic of diagnosis: automated software testing (AST). A large literature review study showed [73] clear benefits of automation: less human effort is required, which reduces the time spent on executing tests, and increased the reliability, reusability and the confidence in the result.

But where there is light, there must be shadow, or limitations. The study [73] also includes an overview on those. First and foremost: automation cannot replace a human. Second, creating a fully automated solution requires time and skills, which in turn means that automation is not the answer to every problem.

In DiaSys, the goal is not to make the human developer superfluous, but to free his or her time for tasks which require skills a computer does not possess: to create novel and creative solutions. DiaSys does not take an “all or nothing” approach. Instead, DiaSys provides a gradual path to more automation and enables developers to find a trade-off between automation and manual work that fits their needs. Reusability is another challenge: the return on invest in writing a diagnosis script is significantly larger if the same script can be used in multiple projects on multiple hardware platforms. We address these challenges with the Dia Language and the Dia Compiler, which are discussed in the next chapter.

3.7 Design Discussion: Limitations and Consequences

The presented diagnosis system is designed to fulfill the requirements outlined in Section 3.1. In the following, we discuss the consequences of the design decisions, which can limit the applicability of the DiaSys concept in some cases.

No access to resident state. DiaSys, like all non-intrusive tracing systems, is only able to observe state changes, not resident state. For example, only memory access can be observed, but it is not possible to access the memory contents itself. This limitation is due to the non-intrusiveness of DiaSys: accessing resident state on a SoC requires interfering with the observed SoC, i.e. by multiplexing accesses to the memory and hence changing the (timing) behavior for the functional SoC. This limitation is especially visible when DiaSys (like any tracing system) is compared to run-control debugging, which draws its power from the ability to access all state available on the SoC once the CPU is halted (i.e. once a breakpoint is hit).

No recording of a full trace stream. By transforming the observed system state close to the source into denser information the off-chip bottleneck can be circumvented. As a downside this lossy transformation thwarts two usage scenarios of today's tracing systems: off-line analysis and reverse debugging.

In many of today's tracing systems it is possible to capture a trace once, store it, and run different analysis tasks on it. If major parts of the captured data are dismissed early, this is not possible anymore. Instead, the analysis task must be defined (as diagnosis application) before the system is run. If the problem hypothesis changes and a different diagnosis application is required, the observed software must be executed again. The severity of this limitation strongly depends on how hard it is to reproduce a bug or behavior across runs.

A special use of recorded trace information is reverse debugging [74]. Reverse debugging (sometimes also called time travel debugging) is the process of stepping backwards through a program flow. This requires a full instruction trace (and often a data trace) being present, which DiaSys by design does not record.

No cross-triggers. Another feature present in many of today's tracing systems, which is explicitly not supported by the diagnosis system, are cross-triggers. Cross-triggers are a mechanism in the tracing system to start or stop the observation, or to observe different components, based on another observation in the system. For example, memory accesses could be traced only after a CPU executed a certain program counter. Cross-triggers are most useful if their timing behavior is predictable. For example, memory accesses are traced "in the next cycle" after the specified program counter was executed. In GALS SoCs, such timing guarantees cannot be given; for a diagnosis application spanning across a SoC and a host PC, it is equally impossible to give (reasonably low bounded) timing guarantees. We make this property explicit by modeling the diagnosis system as a transformational system, not a reactive system. The

commercially available tracing systems today are less specific about this. For example, ARM CoreSight uses a handshaking protocol for cross-triggers delivered across clock boundaries, which guarantees safe delivery of the signal, but does not guarantee any latency.

Instead of relying on cross-triggers to collect data from different sources at the same instant in time, we capture this data already when creating observation events through event generators. The payload of observation events is the only way to pass multiple state observations with a defined timing relation to the diagnosis system. For example, an event generator attached to a CPU can trigger an event based on a program counter value, and attach current contents of certain CPU registers or stack contents to it. Using this method, it is possible to generate for example an event which informs about a function being called, and which function arguments (stored in CPU registers or on the stack) have been passed to it. We show an example of such an event generator as part of our hardware implementation in Section 4.4.8.

3.8 Summary

DiaSys is our approach for non-intrusive software diagnosis on embedded systems. It combines the on-chip observation of software with a flexible data analysis approach, which can be performed (partially) on-chip. Following the event-action model of debugging, software observations are modeled as events, which can be triggered at configurable points in the program execution.

The transformation of observation data into useful information is performed by a dataflow program. This dataflow program consists of multiple transformation actors arranged in a directed tree structure. The semantics of the dataflow program are rather loose by necessity: stricter semantics, which would provide better static analyzability, prevent common use cases of software diagnosis, such as the aggregation of an unspecified amount of events over a fixed period of time (as commonly done in profiling).

We explained the components that make up DiaSys and took a closer look at diagnosis application, a concept at the heart of DiaSys. In the next chapter we move closer towards the implementation, by first describing the three components that form the DiaSys implementation: the Dia Language, the Dia Compiler, and the Dia Engine.

4 DiaSys Design and Realization

The power of DiaSys comes from the generality of the concept, and the flexibility inherent in it. We realized DiaSys in a way which carries this flexibility across the layers of design and implementation.

Before we dive into the details we start with a big picture overview shown in Figure 4.1. The primary goal of DiaSys is to answer questions, questions that a developer has about the execution of software on an embedded system. Examples for questions include “How often is function $x()$ called?” “Could I get a histogram of execution times for this function?” or “How many lines in the code are actually executed?” From a high level point of view DiaSys takes a question as input, and returns an answer as output.

To ask DiaSys a question developers need to formulate it in a way that DiaSys can understand: they write a little program called Dia script in our own domain-specific language (DSL), the Dia Language. But writing a Dia script with only a question is not enough, unfortunately. Developers also need to write code describing what DiaSys needs to do to answer the question they asked. This code is the diagnosis application that was discussed in Section 3.3. In more technical terms, a Dia script written by a developer defines where in the system observations are made, when they are made, and how they are processed to answer the developer’s question.

Once the Dia script has been created it can be passed on to DiaSys which then performs the analysis. To do so the Dia Compiler first transforms the Dia script into code of a diagnosis application and configurations that the execution environment understands. This execution environment, the Dia Engine, then observes the software execution on the chip and analyzes generated observation data according to the code in the script. Finally, the results are presented to the developer, who can interpret them and either refine the question, i.e. start the process over, or end the diagnosis process.

In Section 4.2 we look at the Dia Language, a developer-friendly DSL to write Dia scripts, little programs which consist of a diagnosis application and associated event generator configurations. Then we present the Dia Compiler in Section 4.3, a tool for preparing Dia scripts for execution on the Dia Engine, which we discuss subsequently in Section 4.4.

Before we go into the details of the components, however, we start with a motivational example.

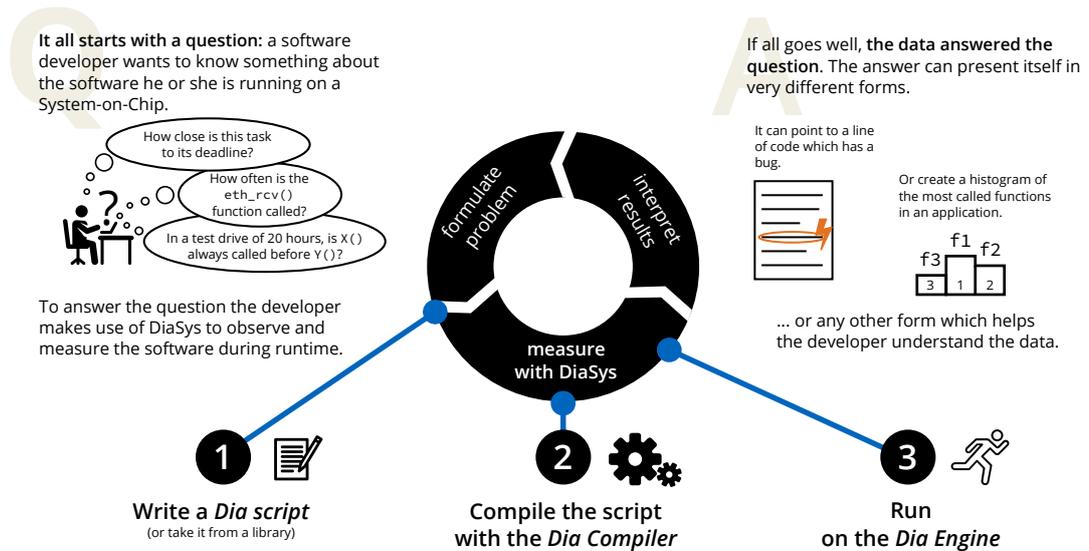


Figure 4.1: An overview of the DiaSys diagnosis method from a user's perspective.

4.1 How DiaSys Works: A First Glimpse From a Developer's Perspective

As shown in the big picture overview in Figure 4.1 a developer's entry point to DiaSys is a question formulated as Dia script. Listing 4.1 presents a small example of such a Dia script. It has been written to answer the question "How often is the program calling the function `push(linked_list ll, int item)`, and in how many of those calls is the argument `item` above the value 50?"

When would such a Dia script be used? Imagine the observed application to be a measurement software. It reads sensor values from an external interface and pushes (appends) the obtained values to a linked list data structure. From this data structure another part of the program can pop (remove) the values to process them further. For the sake of this discussion let the sensor values represent temperatures in degrees Celsius. By using DiaSys the developer now wants to know how often the temperatures exceed 50 °C.

Let's take a closer look at the Dia script in Listing 4.1. The first part of the script are event definitions, starting with the keyword `EVENT`. All communication in DiaSys is done by exchanging events, and these events needs to be defined first. Two types of events exist: events that are generated as result of observations, and events that are created as (intermediate) result of a data processing action.

The definition of the event `ev_cnt` (lines 1–5 in Listing 4.1) showcases how an SQL-like language can be used to describe which units in the SoC are observed, when an observation event is generated, and what data will be "bundled" with it. In addition, two more events are defined in our example. `ev_get_sum` is an event triggered by

4.1 How DiaSys Works: A First Glimpse From a Developer's Perspective

```
1 EVENT ev_cnt
2   TRIGGER
3     AT Cpu WHERE core_id = 0
4     WHEN pc = pc_from_elf("binary.elf", "push.call")
5     CAPTURE uint32_t r5 AS "item"
6
7 EVENT ev_get_sum
8 EVENT ev_sum CONTAINS uint16_t sum_hi, uint16_t sum_lo
9
10 count_events(in ev_cnt, in ev_get_sum, out ev_sum) {
11   static uint16_t sum_hi = 0;
12   static uint16_t sum_lo = 0;
13
14   dia_ev_t* in_event = dia_ev_wait(ev_cnt | ev_get_sum);
15   switch (in_event->type) {
16     ev_cnt:
17       if (in_event->item > 50)
18         sum_hi += 1;
19       else
20         sum_lo += 1;
21       break;
22     ev_get_sum:
23       ev_sum_t* sum_event = dia_ev_new(ev_sum);
24       sum_event->sum_hi = sum_hi;
25       sum_event->sum_lo = sum_lo;
26       dia_ev_send(sum_event);
27       break;
28   }
29 }
```

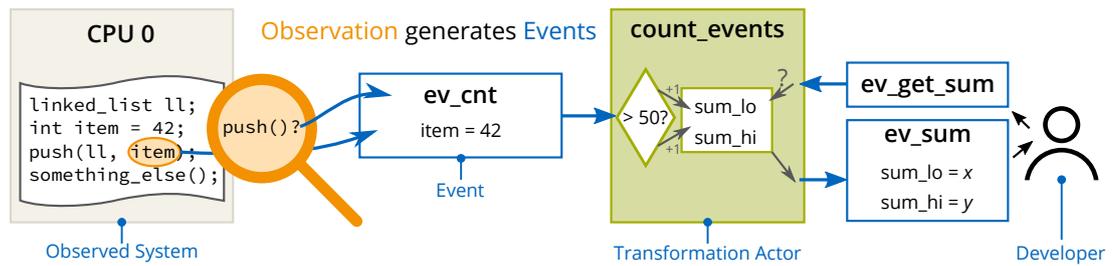
Listing 4.1: A simple Dia script which describes an analysis to count the number of calls to the `push()` function on a CPU depending on the value of the function argument `item`.

the developer when he or she wants to obtain the result of the analysis. DiaSys then responds with an `ev_sum` event which contains the requested information.

What happens based on these events is described in the `count_events` block, which is called a “transformation actor” in the Dia language. Transformation actors are written in a C-based programming language. In our example, the transformation actor waits for an arbitrary input event (line 14), and then, depending on the type of the input event, either counts it (and ends the processing; lines 16–21), or creates a new `ev_sum` event and sends it (lines 22–27).

This small example already highlights some of the unique features of DiaSys. First and most obvious: Dia scripts build on knowledge many embedded developers already have: programming in C. Beyond the script DiaSys abstracts away many implementation details of the observed system and the diagnosis system. For example, the presented Dia script would not look different if changes were made to the processor architecture, the interconnect, or to the way the chip connects to the PC. It does not matter how and where the data aggregation is performed, i.e. if `count_events` is executed on the chip

How a developer sees the processing in DiaSys



... and what happens behind the scenes.

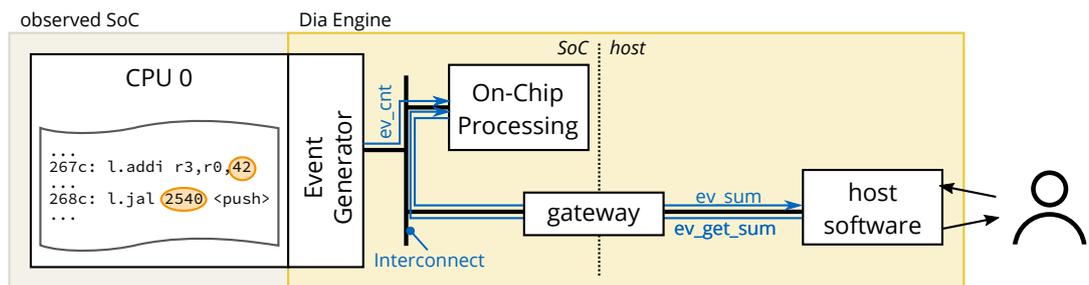


Figure 4.2: Graphical overview of the motivational example in Listing 4.1 as seen from a developer's perspective (top), and how DiaSys ultimately executes the observation and data analysis job (bottom).

or on the host PC. What the developer must know, of course, is the observed system: which CPUs are available, which memories are available, etc.

A second differentiator of DiaSys is the fact that observations can result in events which carry state data. In our example, the `ev_cnt` event does not only signal "the function `push()` has been called", but it says "the function has been called with this value of `item`." It is obvious from our example that data values, e.g. the arguments of a function, are highly valuable when performing diagnosis. Yet state of the art tracing systems often do not provide this information.

After the developer has finished writing the Dia script DiaSys takes over to produce the desired answers. First, the Dia compiler checks and transforms the script into something that the Dia Engine can understand and run. This step involves a compilation and a mapping sub-step, in which the script is matched with the capabilities of the execution hardware (what observation hardware is available, which on-chip processing blocks can be used, etc.). The execution of the script is shown in the bottom part of Figure 4.2. An event generator (a hardware module part of the Dia Engine implementation) is attached to the observed CPU. Before the execution starts the event generator is configured to create a new `ev_cnt` event when it observes a call to the program counter representing the function `push()`. It is also configured to attach the value of the CPU register `r3` to the event, since this register contains the value of `item`, the second argument to `push()`.

4.1 How DiaSys Works: A First Glimpse From a Developer's Perspective

The event then travels over a dedicated interconnect to an on-chip processing unit, where the event is counted according to the code given in `count_events`. How this on-chip processing unit is implemented does not matter at this stage, as long as it provides the functionality asked for. In our implementation we include special-purpose (fixed-function) processing units, as well as freely programmable ones.

The developer interfaces with the system through software running on a host PC. Through this software the developer can send and receive events, in our example the user sends the `ev_get_sum` event and gets the `ev_sum` event back.

This concludes our first look at DiaSys based on a small motivational example. It helps to keep this full picture in mind when we dive into the individual subtopics, starting with the Dia Language.

4.2 The Dia Language: A Domain-Specific Language for Diagnosis Descriptions

DiaSys improves the insight into on SoCs by moving (part of) the data analysis into the chip. The data analysis is modeled as diagnosis application, a transformational dataflow application. The input to a diagnosis application are observation events, which are generated properly configured by event generators.

To give developers a convenient way to “program” diagnosis applications, and to configure the event generators, we have created the Dia Language. It is a DSL, which builds on top of concepts and languages embedded developers typically know.

A standalone piece of code written in the Dia Language is called “Dia script” in the following. This section describes the language, starting with goals and requirements guiding its design.

4.2.1 Goals and Requirements

The Dia Language was designed to meet the overall goals of DiaSys as outlined in Section 3.1. In addition the following goals and requirements are specific to the Dia Language.

Make developers highly productive. Writing a script using the Dia Language should not be a burden to developers, but increase their productivity. In consequence the Dia Language should make use of concepts familiar to embedded software developers, and provide a consistent experience.

Support gradual increase in automation. The ultimate goal for DiaSys is to increase developer productivity. This goal cannot not reached by full automation of all diagnosis tasks, but by making sure a developer’s time is well spent. As [75] shows a trade-off needs to be found: some tasks are best executed by hand (e.g. if they are not well defined, or occur only rarely), while other tasks are best automated. DiaSys should support a gradual increase in automation. A Dia Script may be used to only collect observation data, to perform initial processing on it, or to fully measure and evaluate the results—all depending on the developer’s needs.

Be explicit. The Dia Language should prefer explicit information over implicit conventions. The event and transformation actor-based design of DiaSys clearly shows communication between the parts of the system. The Dia Language should follow down this path to increase readability of Dia scripts for less experienced developers. As a side effect being explicit reduces the guesswork required by the compiler and hence simplifies its implementation.

Based on these goals and requirements we developed the Dia Language to be used in Dia scripts. In the following we discuss the language design in detail.

4.2.2 Dia Language Overview

Upon a closer look, the Dia Language consists of two sub-languages: one to declare events, and one to describe the diagnosis application, i.e. the transformation/processing of events. Both parts are loosely coupled: the event specification sub-language describes which events exist in the system, what data they carry and possibly how they are generated. The transformation actor sub-language is used to describe the code within the transformation actors, i.e. how events are processed. Additionally, by specifying input and output events, the transformation actors define the channels (edges) in the dataflow network.

A Dia script always starts with the event definitions, followed by the transformation actors. In the following we look first at the event specification sub-language.

The language discussion in this section is mostly using a non-formal grammar and examples to build up an understanding of the language without getting lost in details.

4.2.3 Event Specification Sub-Language

The event specification sub-language is the part of the Dia Language used by developers to describe all events flowing in the diagnosis system. Its syntax is inspired by SQL [76] and can be read almost like an English sentence. DiaSys differentiates between two types of events: *observation events* are generated by observing a unit in the SoC, and *simple events* carry information between transformation actors (i.e. within a diagnosis application).

4.2.3.1 Simple Events

The description of a simple event only needs two things: a unique name, and a specification of the data fields it contains. Simple events can contain additional data (payload), but do not have to. If an event is used only to signal that something happened or should happen (e.g. a counter should be incremented, or reset) the event does not contain data.

The event specification of a simple event then takes the following form.

```
EVENT <event-name> [CONTAINS <payload-fields>]
```

The only required part is the event name (<event-name>). After the CONTAINS keyword a comma-separated list of payload fields can be given in the form <datatype> <fieldname>.

Before we can move on to discuss the other type of events, observation events, we need to answer a preliminary question: “How can a developer reference an individual part of the SoC, e.g. the CPU on the top-left?”

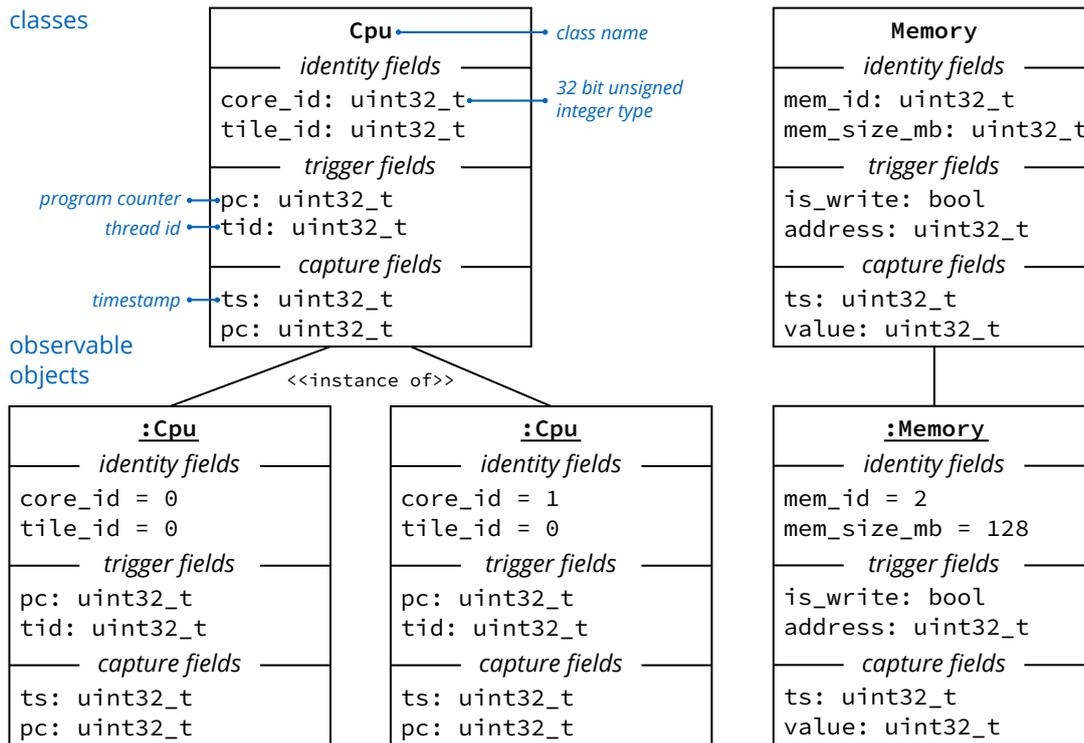


Figure 4.3: Classes and objects of observation units shown in an example.

4.2.3.2 Observation Objects and Classes

To specify an observation event a developer needs a way to reference parts of the SoC, e.g. the specific CPU he or she wants to observe. Developers also need to know what data can be used to trigger on, and what data can be captured with the event. The Dia Language cannot define these things upfront, since they are dynamic and depend on the specific SoC. Therefore, the event specification sub-language makes use of classes and objects of observed units, following well-known concepts from object-oriented programming.

Figure 4.3 gives an overview of the concept by example. It shows three observable objects in the SoC, two CPUs and one memory. The observable objects are grouped into classes; the observed objects are an instance of a class. In the example the classes **Cpu** and **Memory** are shown. Each class has three types of fields: identity fields, trigger fields, and capture fields. All fields have a name and a data type. Identity fields describe and distinguish the object (instance). Trigger fields describe on what data the class of units can trigger. Finally, capture fields define which data can be collected together with an event.

It depends on the Dia Engine implementation which classes of observed units are available. When the engine connects to a target SoC it searches for observable objects, identifies them and makes them available to be used within a Dia script. As a

consequence, classes and objects may or may not be available depending on how and where a Dia script is executed. Whatever objects are available can be used to specify observation events, the topic we discuss next.

4.2.3.3 Observation Events

Compared to simple events, more information is needed to specify an observation event. In particular, the following four pieces of information must be present:

- What is the name of the event? (<event-name>)
- *Where* in the SoC should the observation be made? (<class> together with <object-select-expr>)
- *When* should the event generation be triggered? (<trigger-expr>)
- *What data* should be collected if the event is triggered? (<capture-fields>)

In the Dia Language these four pieces of information are combined into an event declaration which takes the following form (indentation and line breaks are optional and added for clarity):

```
EVENT <event-name>
  TRIGGER
    AT <class>
    WHERE <object-select-expr>
    WHEN <trigger-expr>
    [CAPTURE <capture-fields>]
```

To write the event declaration the Dia Language makes use of the fields specified in the observation objects. Each field type is used in a different part of the event declaration. First, after the AT keyword the class is given (e.g. Cpu). After the WHERE keyword the identity fields select one or multiple object(s) of this class. The WHEN keyword precedes the trigger expression. In this expression any of the trigger fields of the associated class can be compared to arbitrary values to construct a trigger condition. Finally, the captured data (i.e. the data attached to an event) is specified following the CAPTURE keyword by giving a list of one or multiple capture fields.

After the WHERE and WHEN keywords an expression is expected. Just like in SQL, an expression can consist of multiple sub-expressions combined through the AND and OR keywords. Each sub-expression is given in the format <field-identifier> <operator> <const-expr>. Valid <field-identifier>s are discussed below. <operator> can be any of the comparison operators known from SQL and C: = or == (equal), <> or != (not equal), > (greater than), < (smaller than), >= (at least), and <= (at most). Finally, <value> can be either a literal (e.g. a number), or a function call returning a value. (Functions are discussed in the next section.)

With expressions being defined we can focus again on the event specification, and discuss the individual parts in more detail.

- `<event-name>` can be any valid identifier. The identifier must be unique within the Dia script.
- `<class>` identifies the class of objects. Which classes are available is implementation-dependent.
- `<object-select-expr>` is an expression to select an object of `<class>`. In this expression the `<field-identifier>` can be any identity field available to the class.
- `<trigger-expr>` defines the trigger expression. Here `<field-identifier>` can be any trigger field of the class, however sub-expressions can be only combined using the OR keyword, the AND keyword may not be used. (This design choice is motivated below in Section 4.2.3.5.)
- `<capture-fields>` is a comma-separated list of `<capture-field>` elements. Each `<capture-field>` is described as `<datatype> <field-identifier> [AS <alternative-name>]`. `<field-identifier>` can be any capture field available to the class. To provide a more convenient name within a transformation actor, a field can optionally be renamed using the AS keyword.

This ends the discussion of observation events. We next look at a convenience feature, which significantly simplifies writing an observation event description.

4.2.3.4 Functions in the Event Specification Sub-Language

One of the design goals of the Dia Language is to create a highly productive experience for developers. Writing an observation event specification like `WHEN pc = 0x1234` is neither intuitive nor productive: developers typically don't think about program counters, but about function calls, returns and lines in the source code. They can, of course, manually look at the program binary to match a function call to a program counter value, but that is rather tedious. With functions in an event specification this lookup process can be automated. Developers can now write `WHEN pc = pc_from_elf("binary.elf", "main.call")` and the program counter representing the call to the `main()` function in the program binary `binary.elf` is looked up automatically. A function call can be used in place of a literal value in the trigger expression (`<trigger-expr>`), the object select expression (`<object-select-expr>`), and the specification of capture fields.

The Dia Language itself does not specify which functions are available, it is up to the implementation to provide such functions (possibly as extension points).

With the discussion of functions we have completed the description of the event specification sub-language as part of the Dia Language. In the following we discuss the design decisions and their implications.

4.2.3.5 Design Rationale

In the following we discuss the design decisions which led to the current design of the event specification sub-language, not only to serve as a justification, but also to provide a basis for future iterations of the language. Languages, in programming as in real life, are a living entity, they evolve as they are used to provide more efficient communication.

The most obvious discussion question is “Why is a SQL-like language used?” A look at the evolution of the language helps to answer this question. The initial inspiration came from the DTrace [62] and SystemTap [63] languages. These languages, as presented in Section 2.3.4, tightly couple the event handler with the trigger conditions (which are called “probe points” there). For example the following code shows a probe definition in SystemTap syntax which triggers on either the call of a kernel function, and once every second. (DTrace uses a different syntax, but follows essentially the same design.)

```
probe kernel.function("tcp_accept").return, timer.s(1) {  
    // do something  
}
```

Three things should be noted. First, the captured data (i.e. the data that can be accessed within the function body) is not specified explicitly, since the body can access (almost) any data available in the system once the handler is called. Second, what data values are actually observed and compared is implicit: the function trigger compares the program counter, while the timer trigger compares some form of time data. And third, no location where the observation is made is given; it is always assumed that the current CPU(s) are observed. Observing a memory or a NoC router is not in the scope of SystemTap or DTrace.

Following our goals to create a solution which is suited for SoCs, and to prefer explicit descriptions over implicit ones, we need to improve the language into these directions. Even though used on a different abstraction level, SQL addresses our requirements well. If the data generated by the software execution on a SoC is seen as a database, using SQL to select a subset of this data is an obvious choice, as the related work in Section 2.3.3 shows. Using SQL or a SQL-like language does not create additional barriers of entrance for embedded developers. First, (simple) SQL is well-known to many developers who have interacted with databases in some way. Second, even for developers who have never used SQL before the event specifications are easy to understand, since SQL can be read almost like an English sentence.

A natural follow-up question would be: “Why is it a SQL-like language, and not (a subset of) standard SQL?” The short answer: SQL can be both too powerful and too limiting at times. To discuss this answer further we assume in the following basic understanding of SQL.

1. SQL is a large standard and contains many features which are not applicable to our use case. Hence, in any way, a subset of SQL must be defined for our use case. For example, JOINS, subqueries and similar features cannot be supported.

2. SQL blurs the line between data selection and processing. The aggregate functions (e.g. `AVG()`, `SUM()`, etc.) of SQL are a powerful way to combine and process data. In DiaSys, however, we keep data selection (observation) and processing separate to be able to perform the processing in a distributed manner.
3. The `WHERE` clause of SQL is too powerful and ambiguous. In DiaSys observation events need to be specified in two dimensions: in the spacial dimension (which unit in the SoC should be observed), and in the temporal dimension (when should an event be triggered). In SQL both dimensions are combined in a `WHERE` clause, resulting in a (hypothetical) query like

```
SELECT ts FROM cpus WHERE core_id = 1 AND pc = x'1234'
```

to trigger on program counter 0x1234 at core 1 and capture the timestamp (`ts`). This mixing of dimensions in the `WHERE` clause can be seen as reducing the usability, but is not a significant problem.

What is problematic is shown in the following example:

```
SELECT ts FROM cpus
  WHERE (core_id = 1 AND pc = x'1234') AND (core_id = 2 AND pc =
  ↪ x'5678')
```

DiaSys cannot execute this query as triggers cannot be `AND`-combined. If the statement would read

```
SELECT ts FROM cpus
  WHERE (core_id = 1 AND pc = x'1234') OR (core_id = 2 AND pc =
  ↪ x'5678')
```

the execution would be possible. If standard SQL syntax was used the developer gets no clear indication from the language what constructs are allowed, and which ones are not.

For all these reasons, we chose to take inspiration from SQL and keep the event specification sub-language close to it, but take the freedom to modify the basic concepts of SQL according to our needs.

Now that we have explained our use of a SQL-like language we can focus on the details of it. One topic which has been mentioned before is the fact that DiaSys does not allow combining triggers using an `AND` operation (i.e. the `<trigger-expr>` may not contain the `AND` keyword). This limitation can be explained by the asynchronous nature of DiaSys: to support large, asynchronous SoCs, DiaSys does not allow cross-triggers or similar statements which assume an “at the same time” relationship (c.f. Section 3.1).

Another detail of the event specification sub-language is its strong typing, something not seen in SQL. We do this, following our goal to be prefer explicit over implicit functionality, to simplify the implementation and to help developers better understand the data and amount of data exchanged with an event. Since thinking about data types is common for embedded software developers writing C code we do not expect this design decision to create significant usability hurdles.

A last discussion point is our choice of comparison operators, which follow both the SQL and the C standard. SQL uses = and <> for equal and not equal comparisons while C uses == and !=. The event specification sub-language looks similar to SQL, hence the SQL-style operators seem a natural fit. However, the transformation actors, written into the same script file, use C syntax. To reduce the confusion we allow both SQL-style and C-style operators in the event specifications.

This concludes the discussion of the event specification sub-language, in which developers write the first part of a Dia script. The second part of a Dia scripts describes how events are processed. The syntax used for this description is the transformation actor sub-language, which is described next.

4.2.4 Transformation Actor Sub-Language

The transformation actor sub-language is the second sub-language of the Dia Language (next to the event specification sub-language). It describes the diagnosis application, i.e. the processing or data analysis of observation data within a Dia script. The language based heavily on C, with only small modifications done to make it suitable for our use case.

4.2.4.1 General Structure

Transformation actors have a structure similar to a function definition in C.

```
[<attributes>] <ta-name>(<arg-list>) {  
  <ta-body>  
}
```

The following three elements make up a transformation actor.

- <attributes> is an optional space-separated list of attributes helping the compiler to perform the compilation and mapping process. Only the `_on_host_` attribute is supported currently, forcing the mapping of a transformation actor to the host PC.
- <ta-name> is an identifier naming the transformation actor.
- <arg-list> is a list of input and output events the transformation actor takes. It is a comma-separated list of <arg> arguments, which consist of a <direction> <event-name> tuple. <direction> can be either the keyword `in` for events sent to the transformation actor, or `out` for events sent out from the transformation actor. <event-name> is expected to be a valid event name, i.e. an event which has been defined before using the using the event specification sub-language.
- <ta-body> is the body of the transformation actor, i.e. the part which contains the actual processing code. The syntax, which is a dialect of C, is described below.

4.2.4.2 Transformation Actor Body

The body of the transformation actor describes the processing performed on input events which can lead to output events. The code is written in a C dialect, which is valid C in itself. In the following we describe the language in a practical way by describing the syntax for specific scenarios. It should be noted that beyond what is shown here all C features can be used.

Referencing events Within the body of a transformation actor events need to be referenced. For each event defined in a Dia script two identifiers are made available.

- An (integer) constant `<event-name>`, which can be used when the “name” of the event is needed in the code.
- A data type `<event-name>_t` (i.e. the event name with `_t` appended to it) which is used when, unsurprisingly, the data of the event is to be accessed. This data type is called `<event-type>` in the description below.

Waiting for a single event In its simplest form a transformation actor waits for exactly one event to happen. To perform a blocking wait for a specific event the function `dia_ev_wait()` can be used.

```
<event-type>* <identifier> = dia_ev_wait(<event-name>);
```

The function expects the name of the event as argument. It returns a pointer which can be cast to the event-specific data type. An example shows this construct in action:

```
lock_call_t* my_lock_call = dia_ev_wait(lock_call);
```

Waiting for all events in a list of events Waiting for multiple events to all happen does not require special syntax as the functionality can be achieved by placing multiple calls to `dia_lock_wait()` after each other. The ordering of the wait statements does not influence the behavior if the events do not depend on each other.

In the following example the code waits for two events and only then continues with its processing.

```
lock_call_t* my_lock_call = dia_ev_wait(lock_call);  
lock_ret_t* my_lock_ret = dia_ev_wait(lock_ret);
```

A slightly different syntax is needed when not waiting for all events, but for any event.

Waiting for any event in a list of events The DiaSys processing model as described in Section 3.3 allows for non-deterministic behavior of transformation actors by waiting for any event in a list of events. In code this functionality is realized by passing a OR-combined list of event names to the `dia_ev_wait()` function. The function blocks until it receives any event (within the list of events) and returns it with the `dia_ev_t` data type. (The exact type is not known at compile time.)

```
dia_ev_t* <identifier> = dia_ev_wait(<event-name> | <event-name> | ...);
```

A practical example which waits for either a count or a reset event shows this concept at work. To determine which event was actually received the type field of an event can be used. (These fields are further discussed in the next paragraph.)

```
dia_ev_t* my_event = dia_ev_wait(count | reset);
switch (my_event->type) {
    case count:
        // got a count event, do processing on it
        break;
    case reset:
        // got a reset event, do processing on it
        break;
}
```

Access to event data The previous example already showed that all event data types have a type field with the `<event-name>` constant. This field can be used to distinguish event types, which all “inherit” from a base data type, `dia_ev_t`.

```
typedef struct {
    uint64_t type;
} dia_ev_t;
```

The event-specific data types `<event-type>` make all fields specified in a CONTAINS (simple events) or CAPTURE (observation events) clause available as members.

For example, an event declared as `EVENT sum_event CONTAINS uint32_t sum` results in the `sum_event_t` data type as shown below.

```
typedef struct {
    uint64_t type; /* == sum_event */
    // CAPTURES fields start here
    uint32_t sum;
} sum_event_t;

// usage example
sum_event_t* my_sum = dia_ev_wait(sum_event);
uint32_t sum = my_sum->sum;
```

Creating and sending an event Transformation actors can create and send output events. To create a new event the function `dia_ev_new()` is used, and to send an event `dia_ev_send()` is used. The sending of an event does not end the processing of the transformation actor (i.e. return from it) to allow multiple events to be sent.

```
<event-type>* <identifier> = dia_ev_new(<event-name>);  
dia_ev_send(<identifier>);
```

The code below shows how to use this syntax to create an event, assign a value to its data field, and send it.

```
// sum_event as defined previously  
sum_event* my_sum = dia_ev_new(sum_event);  
my_sum->sum = 27;  
dia_ev_send(my_sum);
```

Actor state Transformation actors can have local state, according to the DiaSys processing model. This state is represented by static variables in C. The following example code shows how to make use of local state to implement a counter transformation actor.

```
ta_count(in cnt_event) {  
    static uint32_t counter_state = 0;  
  
    dia_ev_wait(cnt_event);  
    counter_state += 1;  
}
```

This completes our presentation of the special syntax used within transformation actors which goes beyond standard C. Beyond what has been shown all features of standard C can be used. In the following we discuss the design decisions which lead to what has been shown.

4.2.4.3 Design Rationale

The design of the transformation actor language is motivated by three goals. The language

1. should feel familiar to embedded software developers,
2. serve as a foundation for future experimentation, and
3. be analyzable at compile time.

Within this set of goals we evaluated multiple languages and implemented some ideas as prototype. Ultimately, we settled with the C dialect as it is presented in the previous section.

The goal that the language should be familiar to embedded software developers results from the higher-level goal to create a productive language (c.f. Section 4.2.1). Even though diagnosis can take a significant amount of time in the development process, it is still a side task next to the actual writing of code. Writing a Dia script will in many cases only occupy a fraction of a developer's day, and hence the effort of learning a new programming language will pay off only slowly. The language the overwhelming majority of embedded developers are familiar with is C, as statistics show. In the TIOBE Programming Community Index, which is "an indicator of the popularity of programming languages", C has constantly held either the first or the second place since 1988 [77]. Similarly, in a survey done by IEEE Spectrum the first place in the category languages for embedded programming held by C (followed by C++, which is very close to C in the embedded domain) [78]. For this reason, we decided to use C or a language close to C for the description of the transformation actors.

The second goal calls for a language which can be used as a foundation for experimentation. Right now we do not know enough about how developers will make use of the Dia language. This prevents the creation of a high-level domain-specific language, as productivity comes from specialization, essentially following the mantra "do one thing, and do it well." Until we know more about the usage patterns of the Dia Language we want to enable progress, not stand in the way of it. By making the transformation actor language a (possibly overly) general-purpose language it can serve as a base for add-ons, specializations, and other experiments. These experiments may take different forms: voluntary restriction to only some features, additional libraries, or even an additional, source-to-source compiled language layer.

Finally, in the third goal we aim for a language which can be analyzed well at compile time. The transformation actors written in the Dia Language can be mapped to different execution units, some of which provide very specialized functionality. In order to decide at compile time if a piece of code can be executed by such a unit, or if it needs to be executed on a fully-fledged general purpose processor, the compiler needs to fully understand the code. The first step into this direction is already the limitation of side effects to the transformation actor boundaries, i.e. no implicit dependencies or side channels are present between actors. Within an actor a statically typed language significantly helps analyzability, as does the absence of exceptions. All these features are present in C, making it suitable to reach our goal. At the other hand, some C constructs can be very hard or impossible to reason about at compile time. The most significant challenge in this regard is aliasing, i.e. the access of a memory location through different (pointer) variables. Additionally, statically understanding the semantics (i.e. the meaning) of C code is hindered by the low orthogonality of the language, i.e. multiple ways exist to describe the same outcome ($i++$ vs. $i = i + 1$ vs. $i += 1$ is just simplest example).

The discussion of the three goals shows a conflict, especially between the first two goals (developer-friendly foundation for experimentation) and the third goal (analyzability). In the process of creating the Dia Language we considered, partially implemented and evaluated different approaches, from a complete new language design with a syntax similar to C [79] to an extension of C which adds a limited amount of new constructs [80]. Both approaches showed different deficiencies regarding expressiveness, and significantly increased the implementation effort. Ultimately, we settled for the language design presented in the previous section, which makes the full power of C available in transformation actors.

At the same time we encourage authors of Dia scripts limit themselves to a suitable subset of C. At this point in time, we do not know exactly how this subset is defined, but in general the advice is “Limit the code to well-defined, compile-time-understandable, and simple constructs.” Several groups have attempted to create such subsets of C, typically for safety or security purposes. Good examples are the C secure coding rules described in ISO/IEC TS 17961 [81], MISRA C [82], or the SEI CERT C Coding Standard [83].

4.2.5 Summary: The Dia Language

The Dia Language gives developers a voice when writing diagnosis scripts, or “Dia scripts.” These scripts describe where and when to observe the software execution on a SoC, and how to process these observations to yield information, which ultimately helps the developer to understand and improve the software execution. By describing the analysis process in a script as opposed to using a state-of-the-art graphical user interface developers can increase the portability and enable the sharing of diagnosis knowledge.

The main theme in the design of the Dia Language is improving developer productivity. This theme runs through both parts of the Dia Language, the event specification language and the transformation actor language. The former sub-language describes the data observation using a SQL-like syntax. The latter describes the processing performed on the observation data using a C dialect.

The design of the Dia Language was performed in an environment with a lot of unknowns, which is not an uncommon setting for programming language designers. (In fact it is an ongoing debate in the programming language community how a “proper” evaluation of languages should be performed [84, 85].) In our case, we see the design of the Dia Language as a first step which enables further experimentation and improvements. This will be especially necessary once more experience has been gained regarding how the language is actually used by developers. The design rationales given in Section 4.2.3.5 and 4.2.4.3 describe our motivations in order to build up a knowledge base upon which further improvements can be built.

The Dia Language is the interface between the developer and DiaSys. The first component which “takes over” a Dia script is the Dia Compiler, which is described next.

4.3 The Dia Compiler

Between a script written in the Dia Language and its execution on the Dia Engine sits the Dia Compiler. Its job is to transfer a Dia script into something the Dia Engine can execute. To do so it needs input from both sides: the Dia script from the developer, and information about the Engine and its configuration, which depends on the observed SoC. The compiler abstracts away the specifics of the Dia Engine to allow developers to write target-independent scripts. Depending on the available execution units in the Dia Engine the compiler can perform semantic analysis on the code to perform better mapping to special-purpose execution units (such as counters).

How the compiler performs these jobs is described in this section. We start by outlining the goals, non-goals and requirements which guided the design and implementation of the Dia Compiler.

4.3.1 Goals and Requirements

In addition to the overall goals of DiaSys as described in Section 3.1 the Dia Compiler was developed with the following goals and requirements in mind.

Abstract away differences in the Dia Engine. The Dia Engine is a modular system which can be combined and configured in a way which fits the observed system best. The Dia script, on the other side, does not make strong assumptions about its execution platform. Bridging this gap is the main goal of the Dia Compiler.

Explore mapping of transformation actors to special-purpose execution units. In addition to providing general-purpose processing elements to analyze observation data (both on-chip and on the host PC) the Dia Engine can be equipped with special-purpose processing elements. To make use of such elements, such as hardware counters or state machines, the compiler needs to detect code which can be executed on such elements. While this task is impossible in the general case, the compiler should show the feasibility of such a semantic analysis and mapping based on well-defined code patterns. We limit this exploration to the mapping of complete transformation actors; no attempt is made to e.g. split the description of a transformation actor in the Dia Language into two processing nodes.

Show general feasibility. The compiler implementation should have proof-of-concept quality. Neither is the performance of the compilation of output expected to be fully optimized, nor is the usability a focus (e.g. regarding error messages). Addressing these points is beyond the scope of this work.

4.3.2 Introduction to Compilers

While we expect the reader of this work be familiar with the design of embedded systems, the inner workings and design of compilers may not be well-known. Readers

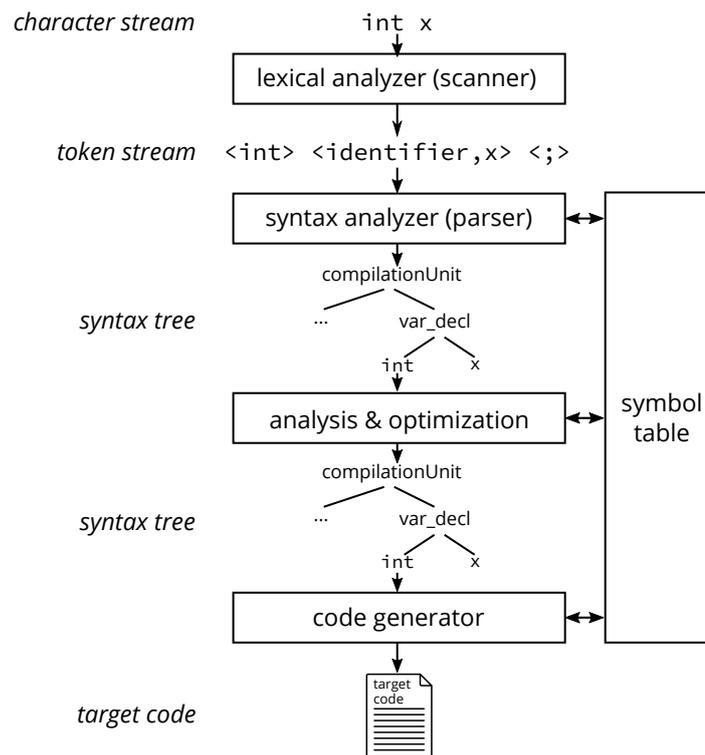


Figure 4.4: General structure of a compiler.

experienced in compiler design may safely skip beyond this section. This section aims to be a very quick introduction to the topic of compiler design as needed for a better understanding of the subsequent sections. Compilers have been developed and studied for decades, and the general structure of a compiler is modified and extended depending on the use case.

In the most general terms a compiler translates between a source language and a target language. Typically, the source language is a high-level language like C, and the target language is an assembly language. The compilation process is split into two parts: the analysis and the synthesis part, also called front-end and back-end.

Figure 4.4 gives an overview of the inner structure of a generic compiler.

First, the lexical analyzer (also called lexer or scanner) splits the stream of source code into a sequence of tokens, each representing an atomic unit of the language.

Then the parser validates the token sequence to check if it conforms to the language grammar. As result a tree representation of the source program is generated, called abstract syntax tree (AST).

The tree representation of the program is used in the semantic analysis phase to check the program against additional rules, and to build up a symbol table. The symbol table is a data structure in the compiler which keeps track of all symbols, identifiers like function or variable names, and their scope.

In an optional next step the syntax tree and the symbol table can be used together to perform various optimizations. These optimizations transform the syntax tree to reach various optimization objectives, such as reduced code size, increased performance, or efficient use of target-specific functionality.

As last step the syntax tree is used to generate the target code, which is the result of the compilation process.

We end our discussion of compilers already at this point. For more information interested readers are referred to [86], [87], and other literature on the topic. (The first two books also served as reference for this section.) We continue with a description of the design of the Dia Compiler.

4.3.3 Design of the Dia Compiler

The Dia Compiler is designed following the established patterns of compiler design, as discussed in the previous section. Figure 4.5 gives an overview of the structure of the Dia Compiler.

The input to the compiler are two files: the Dia script and a platform configuration file. While the former is written by the developer, the platform configuration is dynamically generated by the Dia Engine when it connects to a target system. It contains information about the observable units in the SoC, the attached event generators, and the on-chip and off-chip processing elements.

Out of these two input files the compiler generates multiple output files. First, it generates a configuration file for all event generators in the Dia Engine, which describes when and where to trigger the event generation, and where to send the resulting events to. Second, it generates configuration for all special-purpose on-chip processing elements when transformation actors are mapped to such elements. And third, the compiler generates C code for all remaining transformation actors, which should be mapped to general-purpose processing elements (either on-chip or off-chip).

To generate the output from the input the compiler performs multiple steps. First, it parses the Dia script into a tree intermediate representation (and shows error messages if the script is syntactically invalid). The compiler then uses the platform configuration to validate the script, e.g. to check a Cpu addressed in an event specification actually exists in the target system. Once a script has been validated it can be optimized. The only optimization which we perform is a semantic analysis to determine if transformation actors can be mapped to dedicated execution units. Once all mapping options available the transformation actors are mapped to the available units (as defined by the platform configuration). Finally, the output files are generated and the compilation process ends.

Based on this high-level overview we shed in the following a light onto individual subtopic in the design Dia Compiler. For reasons of brevity we do not discuss design decisions which do not differentiate our compiler design from typical designs. (The full compiler design is disussed in [80].)

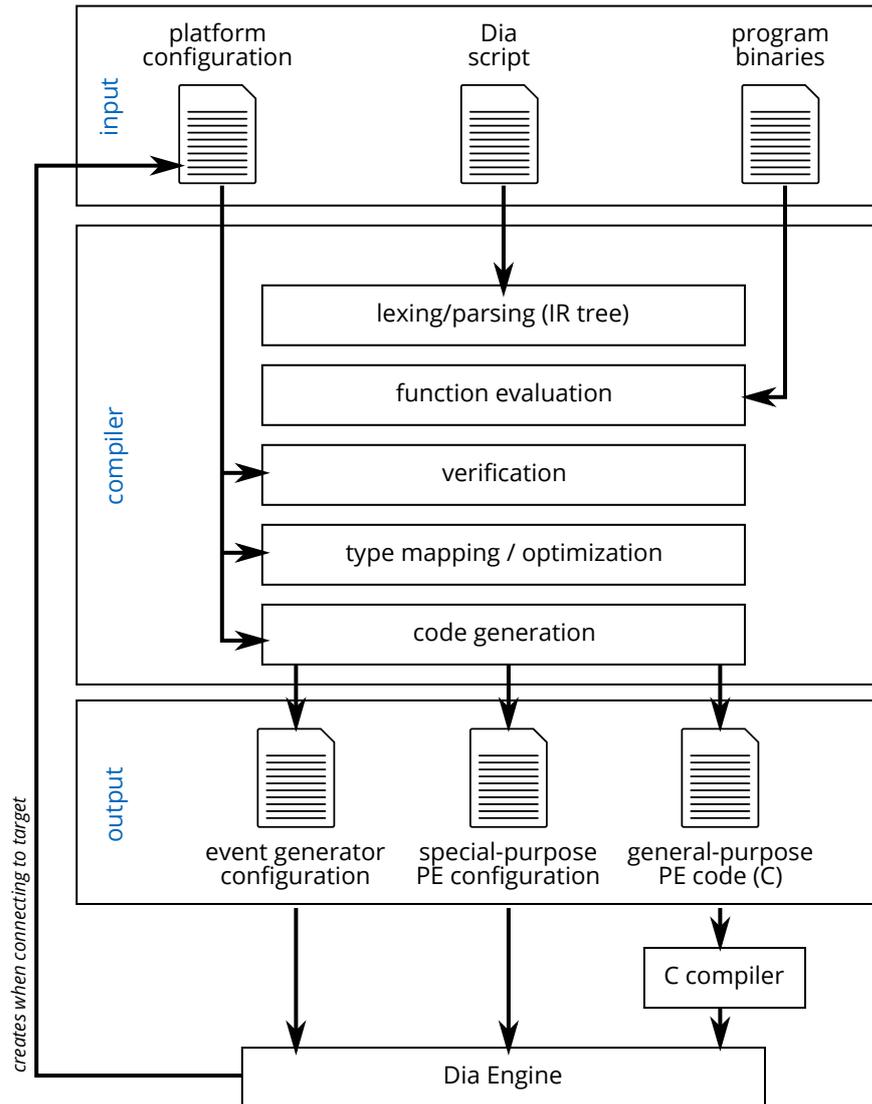


Figure 4.5: An overview on the Dia Compiler, with its inputs, outputs and internal processing steps.

4.3.4 From Event Specifications to Event Generator Configurations

Events are specified using the syntax described in Section 4.2.3. First, all functions (as described in Section 4.2.3.4) are called, and their return value is inserted into the AST in place of the function call. Then all location and data select statements in observation event descriptions are resolved and verified using the description of the available observation units in the hardware configuration file. For example, a statement `core_id > 2 AND core_id < 5` could be resolved to select the CPU cores with IDs 3 and 4. At

the same time, observation event specifications which do not resolve to event generators or valid observed units are reported to the developer.

Once the AST has been verified it is used in two ways. First, the event names are added to the symbol table to make the event names known globally. As a second step a configuration file for the event generators is produced, which is used by the Dia Engine. For each observation event the necessary configuration (i.e. the trigger and capture data configuration) is added to this file.

This concludes the transformation of event specifications, in the next step the transformation actors are compiled.

4.3.5 Translation of Transformation Actors

Transformation actors consist of declaration and a body of C code. The declaration is similar to a function declaration in C, but differs in that it specifies the direction of the event input and output arguments, and that it does not have a return type. The full syntax is described in Section 4.2.4. To be able to perform a deeper analysis of the transformation actor body, the Dia Compiler parses not only the declaration, but also the body of the transformation actor according to the rules of the C11 grammar [88]. Hence, after the parsing step the AST contains the full Dia script down to the individual tokens that make up the body of the transformation actor.

This representation is then verified to check the correct usage of events: only declared events may be used as input or output. And within the body of the transformation actor only events which have been specified in the declaration may be used (e.g. in `dia_ev_wait()` or `dia_ev_send()` statements). The verification of the full C code is beyond the abilities of the Dia Compiler, and can be better performed by a fully-featured C compiler. We therefore do not perform deeper inspection of the body of the transformation actors.

After the verification step all events and transformation actors are known and the dataflow graph of the Dia script can be created. In the general case, transformation actors are mapped to general-purpose processing elements, i.e. an on-chip or off-chip CPU. Since we require a C compiler for such processing elements to be present, we do not perform further compilation steps on the body of the transformation actor, but pass the C code on to a C compiler.

To map a transformation actor to a special-purpose execution unit more work needs to be performed by the compiler, which is described next.

4.3.6 Type Mapping of Transformation Actors

One of the stated goals of the Dia Compiler is to evaluate the feasibility of mapping transformation actors to special-purpose execution units. In the Dia Language, a transformation actor is described using a C dialect, which allows the description of arbitrary processing in the general case. Hence, in the general case, the code of a transformation actor can only be executed on a general-purpose processing element, a CPU. However, in many cases transformation actors describe operations which

are common, such as counting events, or creating statistics. If such commonly used transformation actors can be mapped to special-purpose execution units, as opposed to general-purpose ones, savings in logic area, or increases in performance can be realized.

Before such a mapping can be performed, the code of a transformation actor must be analyzed if it matches the feature set of the special-purpose execution unit. Such an analysis cannot be perfect, and does not need to be, as long it is conservative. C is a highly complex language, which severely hinders (or even prevents) a full semantic analysis at compile time. False negatives only result in suboptimal mapping decisions, if general-purpose execution units are available as fall-back. False positives, however, must be avoided.

The limitations of such an approach are similar to the those of hardware description language (HDL) synthesis tools: if common (and documented) coding patterns are followed the code is detected as such and can be mapped to special-purpose units. If these patterns are not followed general-purpose units are used, degrading the quality of the result, but not the correctness.

Within the area of compiler research the detection of idiomatic similar code has been studied extensively under the term “clone detection” [89, 90]. The approaches can be grouped based on the input and data structures they work on, from a textual analysis, token-based and tree-based approaches to semantic approaches. The approaches can further be categorized based on the type of clones they detect. For our use case we are ideally interested in “type 4” clones, defined as “[t]wo or more code fragments that perform the same computation but are implemented by different syntactic variants” [90]. However, as [90] shows, only approaches performing semantic analysis can reliably detect type 4 clones. Semantic analysis is typically performed using a program dependency graph (PDG), which is not available in our compiler design.

We therefore chose an approach which is less powerful, but able to work with the AST-based intermediate representation our compiler has available. Multiple AST-based clone detection approaches have been proposed, starting with the work of Baxter et al. [91], which is implemented the CloneDR product¹. In our implementation we built on this concept and used tree pattern matching to identify sub-trees in the AST which match a set of patterns. A pattern describes code which can be executed by a special-purpose execution unit. Baxter and subsequent research works describe multiple optimizations which can be performed on the matching processing to make it more robust and increase its performance. In line with our goal to create a proof-of-concept implementation we opted for a “naïve” implementation without such optimizations.

To detect mappable code we perform tree-pattern matching on the AST, or more specifically tree template matching. This type of tree matching permits the pattern to contain wildcards (or “don’t cares”). Tree pattern matching has been studied extensively since the 1960s with many algorithms, optimized for various use cases, being available [92]. In our work we make use of existing implementations to perform the tree matching and do not further try to optimize the algorithms.

¹<https://www.semanticdesigns.com/products/clone/>

```

1 <waitSpec>
2 <waitSpec>
3 <typeSpecifier> <Identifier> = <additiveExpression>;
4
5 <eventCreate>
6 <Identifier>.<Identifier> = <assignmentExpression>;
7 <sendSpec>

```

Listing 4.2: A tree template matching a transformation actor which waits for two events, adds (or subtracts) a payload field within them, assigns the result of the operation to a new event, and sends it out.

The actual pattern matching is performed in two steps: first all bodies of transformation actors are selected from the parse tree using a XPath expression. XPath is a query language for tree structures originally developed to select parts of a XML document [93]. However, subsets of it have seen wider adoption to describe a query on a tree structure. In our case, the XPath expression `//actor/compoundStatement*` selects all bodies of transformation actors from the parse tree.

Then various tree pattern templates are tested on each body of a transformation actor. To simplify the creation of such templates they can be described in textual form using “tags,” which can represent either a token or a rule reference in the grammar ². For example, the template in Listing 4.2 waits for two events, adds (or subtracts) them, assigns the result to the data of a new event, and sends out this event.

This example highlights two properties of our approach: first, the matching can be performed on different levels of granularity. And second, matching results need to be further validated.

The tag `<additiveExpression>` represents a subtree in the parse tree, it can represent, for example the code `2 + 3` or the code `var1 - var2`. Using coarse-grained templates helps to cover more semantically identical, but syntactically different code. But at the same time coarse-grained templates can also match code which is not semantically equivalent and cannot be mapped to a given special-purpose unit. Therefore, all matching results must be validated if they can really be mapped. The more fine-grained a template is the less validation needs to be performed.

If a transformation actor is mappable to a special-purpose execution unit the respective subtree is annotated, which makes this information available to the mapping and output generation steps.

4.3.7 Mapping, Allocation, and Scheduling of Transformation Actors

After the previous compilation steps the application structure is known to the compiler, and a type mapping has been performed, i.e. it is known to which processing elements a transformation actor can be mapped to. What is missing is the actual allocation of execution units, the mapping of the Dia script to them, and the scheduling of their

²For a full description see <https://github.com/antlr/antlr4/blob/master/doc/tree-matching.md>.

execution. Since the problem is NP-complete heuristics need to be applied when searching for a solution.

We build our heuristic on the following facts and assumptions.

- The activation rate (or the event rate) of a transformation actor is not known at compile time. Events in DiaSys are generated by observing software and from user input; the event generation is not controlled by DiaSys, but given by an uncontrolled external environment. This prevents event rates from being known at compile time.
- The execution time of a transformation actor is assumed to be unknown at compile time. Transformation actors are written in a C dialect, which severely hinders the compile-time analysis with respect to the execution time. Even though mechanism to determine an estimate of the execution time, or the worst-case execution time, exist, we currently do not perform this analysis.
- It is assumed that each transformation actor reduces the event rate between input and output events. This assumption is based on the typical use case of DiaSys, where each transformation actor aggregates data to create denser, “more valuable” information.
- Each special-purpose processing element can be used by most one transformation actor (1:1 mapping). Each general-purpose processing element can be used by n transformation actors, where n is a static value assigned to a general-purpose processing element at design time. Hence, for the purposes of the allocation step, each (physical) general-purpose processing element can be seen as n “virtual” elements.

For mapping and allocation the Dia Compiler uses this heuristic: *First, use all special-purpose on-chip processing elements. Then use all remaining on-chip processing elements, before using off-chip ones.*

This heuristic brings the processing close to the data sources (i.e. the event generators) hence potentially reducing the off-chip traffic. And it prefers computationally denser special-purpose processing elements over general-purpose ones. In case that the mapping is ambiguous, i.e. in cases where multiple possible mapping options exist, the behavior is implementation-dependent, i.e. it should be considered to be “random” by the user of the compiler.

All transformation actors are executed using run-to-completion scheduling. Given that transformation actor are expected to be small chunks of code which are triggered repeatedly by incoming events, and that latency within the DiaSys processing model is generally undefined and hence of less importance, this scheduling algorithm is able to maximize the utilization of processing elements by reducing scheduling overhead. The arbitration between runnable transformation actors (i.e. actors which have received a sufficient amount of events to be executed) is implementation-defined, typically using a FIFO mechanism.

The chosen algorithm has limitations especially when multiple mapping options exist. Consider a case where two transformation actors are described in a Dia script, both of which can be mapped to a special-purpose processing element. But only one such special-purpose element is present in the Dia Engine. In this case, it is beneficial to map the most frequently used transformation actor to the special-purpose unit, and map the other one to a general-purpose one. Such a decision cannot be made at compile time as long as the event rates are unknown. A similar scenario is concerned with the mapping of transformation actors to on-chip or off-chip processing elements. Ideally, the compiler maps the actors on-chip which yield the largest reduction in off-chip traffic—a decision which, again, cannot be made at compile time due to unknown event rates.

To overcome these limitations runtime information needs to be collected and used in the process. While we expect such dynamic scheduling and mapping to improve the quality of the compilation result such an investigation was out of scope for a proof-of-concept implementation of the Dia Compiler.

Until the compiler gains this functionality, the developer can use transformation actor attributes to explicitly specify the mapping. We currently use this functionality to force the mapping of some transformation actors to the host PC by specifying the `__on_host__` attribute.

4.3.8 Implementation

The Dia Compiler has been implemented to evaluate the design trade-offs between implementation complexity and performance. Writing a compiler, especially a compiler which supports a complex high-level language, is a non-trivial task and requires substantial development effort. To reduce the amount of work put into the “basic” tasks of compiler construction, i.e. the creation of a lexer, parser and methods to traverse through trees, several “parser generators” exist. The most widely known free and open source options are `lex/yacc`, `flex/bison`, and ANTLR (“Another Tool For Language Recognition”) [94]. The Dia Compiler is built with ANTLR version 4 (ANTLR 4), mainly because it is the most convenient tool to use, a decision which is in line with the goal to create a proof-of-concept implementation of the compiler. In contrast to some other options ANTLR combines the lexer and parser generator in one tool, takes a very general and easy to write grammar as input, and provides a lot of utility classes to iterate through parse trees and abstract syntax trees. ANTLR 4 supports multiple “language targets,” i.e. programming languages used to write the compiler in. (As of mid-2018 Java, C#, Python 2/3, JavaScript, Go, C++ and Swift are supported.³) When the development of the Dia Compiler started only the Java target was usable, hence the Dia Compiler is written in Java. Even though the code should (through its use of Java) run on any operating system, only Linux x86_64 systems were used in the development phase and hence are the only tested target.

³<https://github.com/antlr/antlr4/blob/master/README.md>

The convenience of ANTLR does not come for free: the lexing and parsing is typically slower than compared to other tools, and of course significantly slower than a hand-written parser/lexer. (Since we created no alternative implementation this claim has not been validated by us, but is based on the experience other developers made with ANTLR.) However, the performance we get is within reasonable bounds and not critically deteriorating the user experience (as shown in the evaluation in Section 4.3.9).

An extremely helpful feature of ANTLR which our implementation makes extensive use of is its support for tree listeners and visitors. Instead of writing large amounts of code to iterate over a tree structure, listeners and visitors make it easy to perform actions based on the presence of certain tokens in the parse tree. Additionally, the built-in support for using XPath to search in the parse tree, and template matching to perform subtree matching significantly simplifies the implementation of the special-purpose mapper.

4.3.9 Evaluation

The Dia Compiler was created as a proof-of-concept implementation to evaluate where the bottlenecks and challenges lie. Our evaluation (beyond the correctness of the result) focused on the compilation performance and especially on the mapping of transformation actors to special-purpose execution units.

4.3.9.1 Compiler Performance

A Dia script is compiled on the fly when it is executed, after the connection to the target system has been established (only then the available observation units are known). Therefore, the compiler must be fast enough to not impact the developer productivity.

We evaluated the execution time and memory usage depending on the size of the input Dia Script.⁴ A small Dia script of roughly 1 kB takes less than 1 s to compile. This time increases linearly with the size of the input script. The compiler throughput, the size of the input Dia script divided by the execution time, saturates at approximately 75 kB/s when processing multiple megabyte of data.

The memory usage in our measurements stayed below 200 MB, but did not show a clear dependency on the input file size. We can only speculate as to the reasons, but the nondeterminism of the Java garbage collector might play a role.

Overall, we can conclude that the compiler, even though it is not particularly fast, is sufficiently fast for our use case. Typical Dia scripts in our evaluations were between 1 kB and 10 kB in size, resulting in a compilation taking not more than 1.5 s.

4.3.9.2 Effectiveness of Processing Element Type Mapping

Section 4.3.6 discussed the ability of the Dia Compiler to identify code constructs in a transformation actor which can be mapped to special-purpose execution units (such as

⁴The measurements were performed on a x86_64 Intel Core i7-3770 CPU running at 3.40 GHz with 16 GB memory being available. The system ran Ubuntu Linux 16.04.

dedicated counters). Since, as discussed, a full-blown semantic clone detection is not feasible within the scope of this work, we used an approach based on tree matching. In this approach code templates are compared with the user code to determine if it follows a common structure which is known to be executable by a special-purpose unit.

To gain initial insight into the effectiveness of this approach we performed a limited user study in which participants were asked to write a part of a Dia script to fulfill a given task. The answers were then collected and evaluated to answer two questions. First, do the templates we created before the study match the code written by the users? And second, how would the templates need to be modified to match the user's code? How many templates would be needed to match all user inputs?

The questions were handed out on paper, and the participants were informed about the goal to find differing syntax for coding the same algorithm. In the first part the Dia Language was introduced by giving an example of how to write a script which implements an event counter (both with code and textual explanations). Then two tasks were given.

The first task asked the participant code a transformation actor which calculates a moving average with a window size of three. The code to wait for input events, and to send out output events, was already given. Missing was code to remember the last three values of an event's data and to calculate the average out of it.

The second task asked the user to write a complete transformation actor to calculate the difference between the payload of two events.

The last part of the questionnaire asked for a self-assessment of the participant. Did he/she understand the question? How difficult did they find reading and creating a script? Finally, the participants were asked to rate their programming experience and state their main programming language.

The tasks were chosen with the focus on two aspects. Calculating a moving average in C syntax can be performed in various ways. The primary goal of this task was to evaluate how many syntax options the users would come up with. The calculation of a difference between two variables can be performed in less ways. Since this task required the participant to write more "boilerplate" code, i.e. the full transformation actor, our evaluation focus in this task was the matching performance when considering larger code fragments.

In the study eight (8) people participated, all of which were either staff members or students at the Chair of Integrated Systems. Half of the participants stated between one and three years of programming experience, the other half had at least three years of experience. All but one participant uses C or C++ as their primary programming language. Nobody found the process of reading or writing a Dia script "hard," and everybody with three or more years of programming experience found it to be "easy."

Since all coding was done on paper, the answers were edited for minor mistakes (such as omitted semicolons (;) to make them compilable. In the first task one submission had to be excluded from the evaluation as the code did not compile. The code was then compiled using our compiler and the matching performance was observed.

Our first research question was: "How many solutions are matched by the predefined set of templates?" For the moving average actor a single template matched 29 percent

(2 out of 7) solutions. For the difference actor a single template matched 63 percent (5 out of 8).

The second research question was: “How many templates would be needed to match all submitted code?” For the moving average actor, the solutions were differing so much that only an individual template for each solution would have created a 100 percent match. For the difference actor two templates would have already resulted in 88 percent of match rate, while three templates would have created full coverage.

The results can be interpreted in various ways. First, non-novice developers experienced in C/C++ did find reading and writing Dia scripts easy. This indicates that our choice of syntax for the transformation actors is intuitive for this group of developers.

Second, the matching of code to describe simple constructs can be performed reliably, as the difference actor task showed. With only three fine-grained templates all given answers could be matched; by using fine-grained templates only a small amount of validation needs to be performed on a successful match.

Third, matching of more complex algorithms, such as the moving average actor, shows the limits of the tree pattern matching approach for the Dia Language. The conclusion could be two-fold: either the language provides too little orthogonality, i.e. allows too many syntactic variants to describe identical semantic concepts. Or the matching approach is not powerful enough, and a semantic clone detection approach could be used.

Finally, while the survey gave interesting insights, its conclusions cannot be generalized. The sample size was very low (eight participants), and the participants do not represent the full spectrum of potential users of DiaSys. Additionally, the information at the beginning of the questionnaire that we look for syntax variations might have invited participants to write more “exotic,” but syntactically valid, code. This was confirmed in personal interviews after the evaluation was performed.

4.3.10 Summary: The Dia Compiler

The Dia Compiler translates Dia scripts, written in the Dia Language, into code that can be executed by the Dia Engine. The compiler was designed and implemented to show the general feasibility of the processing chain from Dia scripts to their execution. As a secondary goal, the compiler serves as an experimentation platform to gain insight into the optimizations which can be performed on a Dia script. In particular we evaluated the identification of transformation actors describing given algorithms with the goal to map them to special-purpose execution units in the Dia Engine. The implementation was done in Java using the ANTLR compiler framework and targeting Linux x86-64 systems.

The evaluation shows that the compilation process can be performed fast enough not to hinder the diagnosis workflow of a developer. Furthermore, a small user study showed that the identification of small mappable code fragments can be done reliably using a tree pattern matching approach. However, for more complex algorithms this approach does not result in good matching performance. This result could be addressed in two ways: The Dia Language could be modified to provide fewer options in its syntax

to describe the same algorithm. Or a different approach to detect code clones could be used, most likely also creating a need for an additional intermediate representation. As both mitigations are not exclusive, it is likely that a sweet spot lies somewhere in between the two approaches.

This concludes our look at the Dia Compiler. The output of the compiler is consumed and executed by the Dia Engine, which is discussed next.

4.4 The Dia Engine: The Workhorse of DiaSys

The *Dia Engine* is the work horse of DiaSys, it is the platform on which Dia scripts are executed. For that the Dia Engine must do two things. First the software execution on an embedded system must be observed to collect data. And then the collected data must be analyzed as described in the Dia script.

The observation of the software must by nature happen where it is executed: on the chip. The data analysis, however, can theoretically be performed anywhere: on the chip, on a host PC, or on a dedicated device in-between (an option we do not discuss further). The Dia Engine provides this flexibility in its design and its implementation.

During the design and development process we recognized that the features of the Dia Engine are useful in other projects and for other people as well. We therefore created the Open SoC Debug (OSD) project with the primary goal to create a specification for a reusable debug and tracing system, together with an open source reference implementation. In its current form, the Open SoC Debug (OSD) specification is closely aligned with the Dia Engine described in this thesis. Hence the description of the Dia Engine in this thesis focuses on the novel and “interesting” aspects of the architecture and its implementation.

The full specification of OSD is available online [95], as are the reference implementation of the software and hardware components. The web site <http://www.opensocdebug.org> links to all available material, download links, and documentation.

The first part of this chapter presents the overall architecture of the Dia Engine and motivates its design decisions. The second part then describes the implementation of the architecture both in hardware and in software, and discusses the implementation-specific design decisions which have been made.

4.4.1 Architecture of the Dia Engine

The Dia Engine collects and analyzes observation data from software executed on an embedded system. What data is collected where, and how and where it is processed varies heavily between implementations and use cases. We therefore designed the Dia Engine in a very flexible way. All data gathering and processing is performed by components called Debug Modules. Debug Modules can communicate with each other over the Debug Network. A number of Debug Modules are provided with the Dia Engine to cover the most common use cases, e.g. the observation of a CPU, or the processing of observation data.

As an introduction Figure 4.6 presents the architecture of the Dia Engine in an exemplary configuration. The shown setup consists of multiple on-chip and off-chip Debug Modules, which are connected by the Debug Network. The Debug Network is segmented into two subnets, one on-chip subnet (left) and one off-chip subnet (right). The subnets are connected by a gateway.

The example includes various types of Debug Modules. Within each subnet a Subnet Control Module (SCM) provides management and discovery functionality. The Core

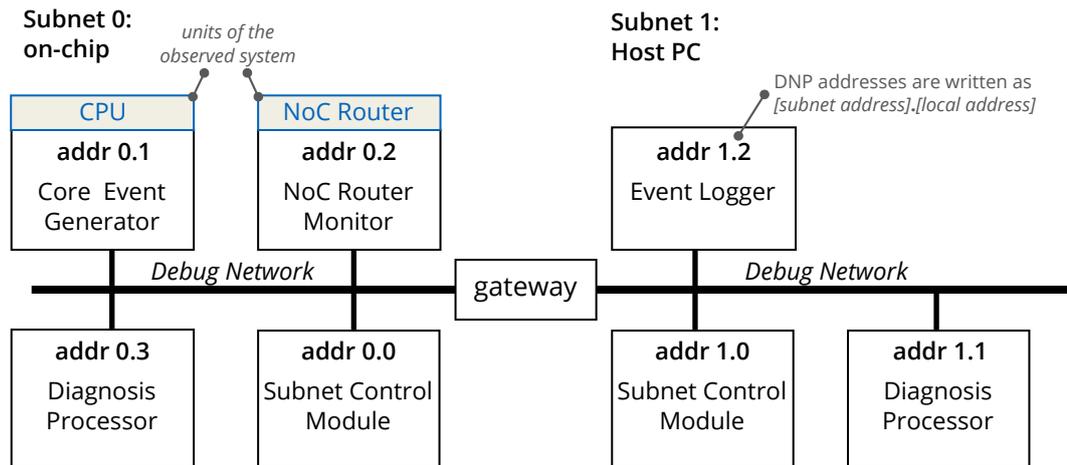


Figure 4.6: The architecture of the Dia Engine in a simple exemplary setting with two subnets connected through a gateway. Each subnet is shown with the required Subnet Control Module at the local address 0 and several exemplary Debug Modules. See Figure 4.12 for a more advanced scenario with multiple cores.

Event Generator and the Noc Router Monitor are Debug Modules which collect data from components in the observed system, here a CPU and a Network-on-Chip (NoC) router. The Diagnosis Processor, which can be placed on-chip or off-chip, processes the observation data, and the Event Logger shows the analysis results to a developer.

The example in Figure 4.6 provided an overview of the Dia Engine. In the following we discuss the individual components in more detail.

Debug Modules are general encapsulations of data gathering or processing functionality within the Dia Engine. Debug Modules can be used for widely different purposes: they can observe units in the SoC, e.g. a CPU, but they can also process observations or control the target system, e.g. to reset it or to initialize its memories. The architecture of the Dia Engine does not mandate where Debug Modules must be placed or how they are implemented: in hardware or in software, on-chip, off-chip, or in another device.

All Debug Modules are connected through the **Debug Network**. The Debug Network is a packet-switched, connectionless network which provides bi-directional and directed (unicast) communication between Debug Modules. A **DNP address** uniquely identifies the Debug Module within the network.

The Debug Network does not mandate a specific network topology. For implementation and routing purposes the Debug Network can be segmented into **subnets**, which are connected through **gateways**. In the most common case subnets are used to separate the on-chip part(s) of a Debug Network from the off-chip (host) part.

Within the Debug Network the **Debug Network Protocol (DNP)** is used for data exchange. This network protocol can be layered on top of other communication protocols and used with various physical interconnect types. For example, DNP can be used on-chip over a NoC with low overhead, or on a host PC on top of TCP/IP.

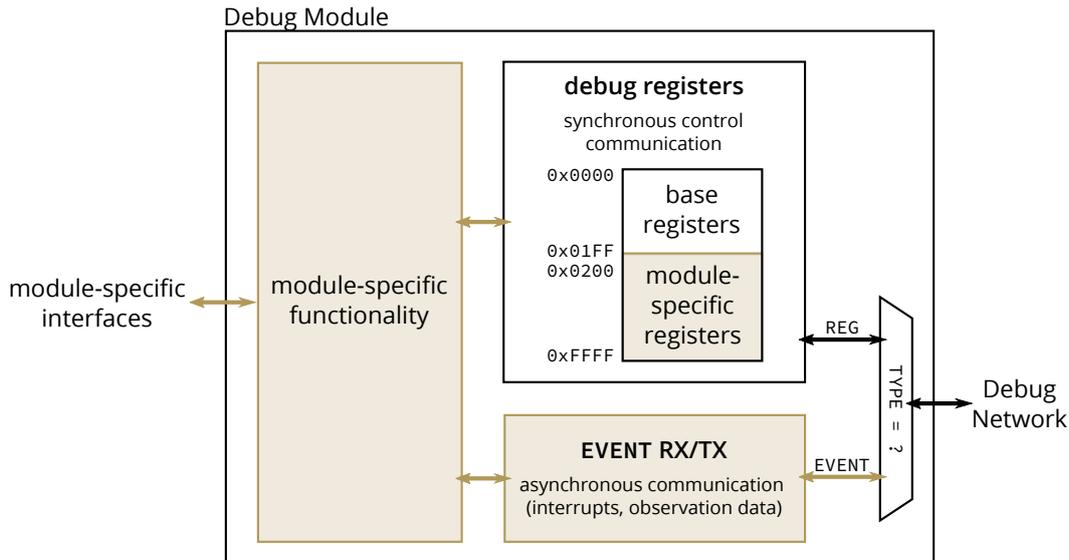


Figure 4.7: Block diagram of a Debug Module. Only the base registers are required, all other blocks (shaded in the figure) are optional and their use depend on the specific purpose of the Debug Module.

The data transferred within the Dia Engine falls into two categories with distinct characteristics: control and streaming traffic. Synchronous, request-response based communication between Debug Modules is characterized as control traffic, while the transfer of observation data, e.g. from a CPU, are best described as streaming traffic. The Debug Network and the DNP are designed to handle both types of traffic within the same network and protocol while taking care of the individual needs of the two traffic types.

Control communication between Debug Modules is modeled as register access, which is performed over the Debug Network by **register access packets**. **Event packets** are used to exchange unsolicited (“asynchronous”) and often high-bandwidth data between Debug Modules. The data format specified within the DNP for event packets places few restrictions on its contents. This allows Debug Modules to efficiently pack data in a way which reduces its size and hence increases bandwidth utilization.

This concludes the overview of the Dia Engine architecture. The following sections take a closer look at the Debug Network, the Debug Network Protocol, and associated infrastructure components, before moving on to implementation topics.

4.4.2 Debug Modules

Most functionality provided by the Dia Engine is encapsulated within Debug Modules. Debug Modules share a common structure, as shown in Figure 4.7. An interface to the Debug Network is present in every Debug Module. Over this interface both register access and event data can be exchanged. However, only accesses to certain registers,

called base registers, must be handled by every Debug Module. The 200 base registers are the lowest common denominator between all Debug Modules. Three base registers are used to describe the module through a vendor and module identifier and a version number. A further base register is used to activate and deactivate sending of event packets from the module.

Debug Modules often extend the base register set with additional registers for module-specific functionality. Up to 65336 register addresses are available for this purpose, and module-specific registers can be 16, 32, 64 or 128 bit wide. (Base registers are always 16 bit wide.)

All remaining blocks in a Debug Module are optional. Depending on how the Debug Module is used it may or may not send or receive event packets. It may interface with other components (e.g. a CPU on a chip, or a graphical user interface on a desktop PC), or it may be self-contained (e.g. in case of a Debug Module which analyzes observation data). We present specific implementations of Debug Modules later in this chapter.

4.4.3 The Debug Network: Communication Within the Dia Engine

The Debug Network provides bidirectional any-to-any connectivity between Debug Modules, even across chip and device boundaries. It is a connectionless, packet-switched network in which data is exchanged according to the rules of the Debug Network Protocol (DNP).

The Debug Network is designed to provide interoperability between Debug Modules while leaving enough room for optimizations when implementing it (e.g. on a host PC or within a SoC). For this reason all implementations of the Debug Network need to guarantee the following properties.

- The network must guarantee the delivery of every packet if the destination is generally reachable. All injected packets must reach its destination after some time, no packets may be dropped.
- The network must guarantee strict ordering of packets with the same (source, destination) tuple. If source *A* sends multiple packets to destination *B*, the packets must arrive at *B* in the order *A* has sent them. No ordering guarantees must be given regarding packets sent from, for example, source *C* to destination *B*.

Within those boundaries implementations have significant freedoms.

- No specific network topology is mandated. (We have implemented both star and ring topologies.)
- No specific routing algorithm is required, it is up to the implementer to choose a suitable one.

Three aspects of the Debug Network deserve further discussion: subnets, the subnet controller, and gateways.

4.4.3.1 Segmenting the Debug Network Into Subnets

To a Debug Module the whole Debug Network is opaque in that it transparently provides any-to-any connectivity between all Debug Modules, wherever they live. Internally, however, the Debug Network can be segmented into “islands” called subnets. Splitting up the Debug Network significantly eases its implementation. Within a subnet an implementor can freely choose a network topology, routing algorithm, and the protocol stack below the DNP. (The last aspect is discussed in more detail in Section 4.4.4.2.)

In the most common scenario two subnets are used: the on-chip segment of the Debug Network is organized as a first subnet, and the software on the host PC is organized as a second subnet. However, subnets have uses beyond such a simple scenario. They are in general useful wherever a system should be seen as a whole from a debug point of view, but where it in reality consists of different connected components, potentially even provided by different vendors. One such advanced scenario is sketched in the discussion in Section 4.4.3.4.

Debug Modules are addressed by their DNP address, which contains the subnet as part of the address. This makes it straightforward for a router to decide if a packet should be routed within its own subnet, or sent to another subnet. If another subnet should be reached the traffic is sent through a gateway.

4.4.3.2 Gateways in the Debug Network

Gateways connect the different subnets of the Debug Network. A gateway is a point-to-point connection between two subnets (typically between two “gateway routers” in a subnet). Gateways are invisible to the Debug Modules as they are not assigned a DNP address.

In addition to connecting subnets, gateways can be used to convert data between transmission protocols. In the typical example of two subnets, one on-chip and one on a host PC, the off-chip connection is hidden behind the gateway. On the chip side the gateway is connected to the debug NoC, and on the host side the gateway connects to a TCP network. All necessary data conversion in between is handled in a way that is not visible to the Debug Network.

In addition to subnets and gateways, Debug Networks require a third component: the Subnet Control Module.

4.4.3.3 The Subnet Control Module

The Subnet Control Module (SCM) is a Debug Module which is always present at local address 0 of a subnet, and which provides subnet-wide functionality. This functionality can be grouped into three categories: description, enumeration and implementation-specific functions.

The SCM describes a subnet through an implementation-defined vendor and device identifier. A debugger can use this information to modify its behavior depending on

which target it connected to, or to simply display it to a user. The current status of the chip can also be part of the description: “Are the clocks in the system running?” “Is the chip in a secured mode?”, etc.

Enumeration is the process by which Debug Modules learn which other Debug Modules are present in the network. The SCM is the only module in a subnet which is always present and has a well-known address. It therefore facilitates the enumeration process by serving as a first point of contact and by keeping track of all other Debug Modules within the subnet.

Finally the SCM can be used to control subnet-wide functionality. The exact functionality is implementation-defined. For example, in our on-chip implementation of the Dia Engine the SCM can reset the observed system and stall and start its CPUs.

With the presentation of the SCM we completed our journey through the Debug Network. The next section continues by motivating key design decisions and widens the view to how the Debug Network enables novel functionality.

4.4.3.4 Discussion: The Design of the Debug Network

The presented architecture of the Debug Network is tailored towards the needs of the Dia Engine and differs significantly from today’s commercial approaches for debug connectivity as discussed in Section 2.2.1. The two main differentiators are the use of packet-switched, connectionless communication, and the removal of the strict distinction between host and device. Both design decisions are primarily motivated by the desire to enable a transparent mapping of data processing elements to the host, or to the device.

The first differentiator is the use of packet-switched, connectionless communication, both of which can be explained by the gained flexibility. If for example a Debug Module wants to send observation data to a different processing unit it only needs to change the destination address of its data packets, thanks to the packet-switched nature of the network. The module also does not need to initialize or tear down communication channels, communication can start immediately.

This flexibility is paid for with larger packet overhead, since every packet needs to contain full routing information. (In our design the routing information is the destination DNP address.) We minimize the impact of this overhead in the design of the data exchange format, which makes it possible to reuse the routing information across protocol layers. For example, in our implementation the destination address is used on the DNP layer as well as on the NoC transport layer. This optimization is further discussed as part of the hardware architecture in Section 4.4.5.2. If in the future the overhead of certain transmission paths should be reduced further, routers can be extended to create a “fast path” between two nodes, essentially creating a fixed communication channel between two nodes, which allows removing the source and destination information from the Debug Packet during the transmission (and adding it back at the destination node to restore the full packet).

As a second main differentiator the Debug Network architecture does not give special meaning to a SoC or to the host PC. Instead of hard-coding their semantics, the Debug

Network consists of multiple subnets, all of which follow the same semantics. Beyond the typical scenario of two subnets (one host subnet, and one on-chip subnet), more advanced scenarios are possible, for example the following two.

- Diagnosis beyond die or chip boundaries: The integration of multiple chip dies within a package, or the close coupling of multiple chips to achieve a given functionality is becoming more common. The closer the coupling, the closer are also the interactions in software, and the more pressing is the need to perform software diagnosis across die or chip boundaries.
- Dedicated diagnosis dies or chips: Already today Infineon (and potentially other chip vendors) produce two versions of their SoCs. The standard version contains only the SoC itself in a package, while special debug chips (called “emulation device” by Infineon) bond two dies together within one package: the SoC itself, and a reusable die containing advanced tracing functionality and memory. Similarly a SoC die could be produced containing a subnet of Debug Modules focusing on data collection. A second die could then provide a subnet of Debug Modules performing advanced data analysis. Cost savings can be realized by reusing this “diagnosis die” for multiple SoCs, and only packaging it with the SoC when the advanced analysis functionality is actually needed (e.g. for development chips).

Both example scenarios show how the design of the Debug Network fosters reuse, helps to decrease cost, and increases insight into today’s complex SoCs.

The design of the Debug Network, which was motivated in this section, enables any-to-any connectivity between Debug Modules. But not only connectivity, but also a “common language” is required for a successful communication. This common language, the Debug Network Protocol, is described next.

4.4.4 The Debug Network Protocol

The Debug Network is designed to enable seamless communication between Debug Modules, wherever they reside (on-chip, off-chip), or how they are implemented (in hardware or in software). The network itself provides connectivity between the components, which is a necessary prerequisite, but not sufficient. For successful communication the components need to speak a common “language,” they need to use the same communication protocol. For the purposes of the Dia Engine the communication protocol must fulfill three requirements.

1. The protocol must provide a common data exchange format for both (request-response) control traffic, as well as for asynchronous, streaming traffic.
2. The data exchange format must be agnostic to how a Debug Module is implemented or where it resides.
3. It must be possible to layer the protocol on top of other transmission layers.

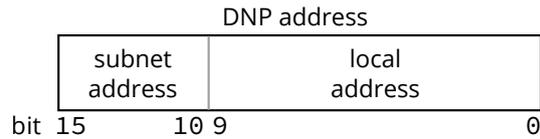


Figure 4.8: Structure of a 16 bit wide DNP address. The most significant 6 bit represent the subnet address, the remaining 10 bit represent the (subnet-) local address.

Based on these requirements we created the Debug Network Protocol (DNP). In addition to the data format the specification also defines a communication protocol, the rules which guide the interaction between Debug Modules. In the following we start with a discussion of the general aspects of the DNP: how Debug Modules are identified within the network, how DNP is layered on top of other protocols, and how the DNP supports the segmentation of the Debug Network into subnets. This discussion is followed by an in-depth description of the data format and protocol rules governing the DNP.

4.4.4.1 Addressing in the DNP

Each Debug Module is assigned a DNP address which uniquely addresses it. To ease the routing of data between subnets a DNP address is split into two parts: a subnet-local address and a subnet address. (This approach is comparable to the implementation of subnets in TCP/IP.) Figure 4.8 shows this split for a single DNP address. The most significant 6 bit of a 16 bit wide DNP address identify the subnet and are hence called “subnet address.” The remaining 10 bit are called “local address” and identify a Debug Module within a subnet. This structure enables routers within the network to easily determine which packets should be routed within the own subnet, and which packets must be routed through a gateway to reach another subnet. By assigning 10 bit to the local address each subnet is restricted to $2^{10} = 1024$ Debug Modules. The complete network can consist of $2^6 = 64$ subnets, and a total of $2^{16} - 2^6 = 65472$ non-infrastructure Debug Modules (excluded is one Subnet Control Module for each subnet).

4.4.4.2 Layering of the DNP

Figure 4.9 shows a typical layering of protocols when a Debug Module on a chip communicates with a Debug Module on a host PC. On the chip the data is transmitted over a NoC, where each DNP packet is transmitted as one NoC packet. To transfer the data between the chip and the host each DNP packet is wrapped in a Debug Transport Datagram (DTD). The DTD data format is designed for efficient point-to-point transmission of debug packets in a stream. A DTD stream can be transmitted over any off-chip interface a chip provides; in the exemplary setup we used USB 3.0. Not shown in the figure are the protocol layers that Universal Serial Bus (USB) itself consists of. Once the data stream reaches the host PC a DNP packet is wrapped inside

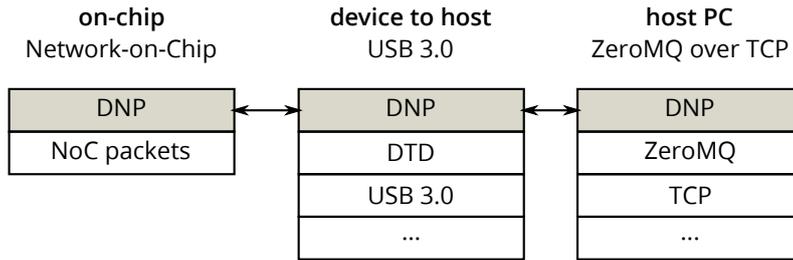


Figure 4.9: An exemplary layering of protocols in the DNP for a common communication scenario between on-chip and off-chip components.

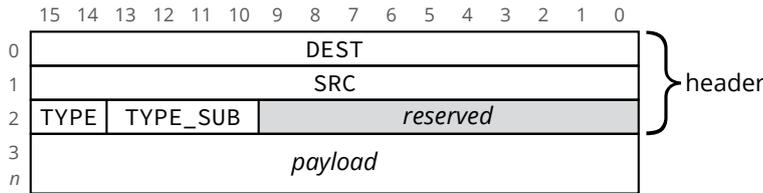


Figure 4.10: Structure of a Debug Packet. The first three 16 bit words are the header, consisting of the destination address, the source address, the packet type and the packet subtype. All remaining words are occupied by payload.

a ZeroMQ message. ZeroMQ is a lightweight communication protocol which itself builds on top of and abstracts from various other protocols, such as TCP.

While the shown layering in Figure 4.9 is typical, it is in no way the only option: all layers below the top DNP layer can be replaced, and they often are. For example, our implementation of the Dia Engine can also make use of UART, JTAG or Ethernet for off-chip communication. On the host TCP can be replaced with, for example, in-process or inter-process shared memory communication for increased performance.

4.4.4.3 Debug Packets, The Common Data Exchange Format of the DNP

The DNP describes a common format for all data exchanged within the network called Debug Packet. Debug Packets are structured as sequence of 16 bit words. Unless specified otherwise the byte ordering is big endian.

The structure of a Debug Packet is shown in Figure 4.10. The first and second word are the DEST and SRC fields, containing the DNP address of the packet destination and the packet source. The third word is used for packet flags. The two most significant bit (TYPE) specify the packet type, followed by four bit for the packet subtype (TYPE.SUB). Ten bit remain in the third data word which are currently unused and reserved for future extensions. Starting with the fourth data word is the packet payload, which may be empty. The number and semantics of the payload words depend on the packet type and possibly its subtype.

Two packet types are defined: TYPE = 0b00 (REG) is used for register access packets, while TYPE = 0b10 (EVENT) is used for event packets. These two packet types enable

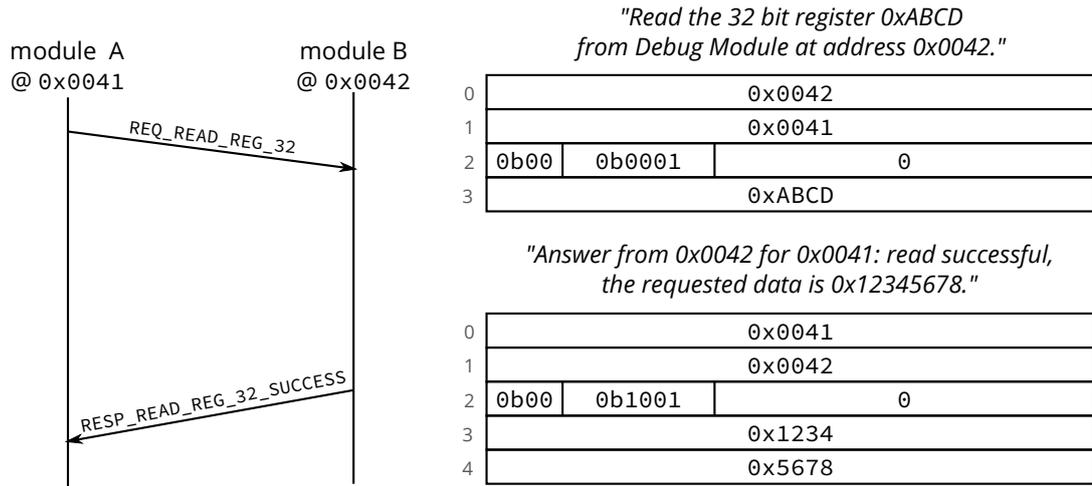


Figure 4.11: An example scenario showing the successful read access of a 32 bit wide register at the module at DNP address 0x0042 by module 0x0041.

distinction between the two main traffic types: register access packets are used for request-response oriented, low-volume traffic, while event packets are used for unsolicited (asynchronous) and often high-volume messages. Typical use cases for event packets are signaling between Debug Modules (e.g. "interrupt x was fired") and the streaming of observation data between modules.

The maximum number of words that make up a Debug Packet can be limited by the implementation. This maximum length must be at least 12 words, i.e. all implementations must support Debug Packets consisting of 12 words or less. Specifying a maximum packet length is useful if an implementation needs to buffer a full packet somewhere in the system. Knowing the maximum size of a packet allows for examples chip designers to instantiate a buffer in hardware which is in any case able to hold a full packet. The requirement to set this length to at least 12 words is due to the fact that register access packets can be up to 12 words in size, and splitting such packets is not allowed by the protocol. The maximum packet length for any given subnet can be queried from the Subnet Control Module.

In the following sections the two types of Debug Packets are discussed in detail: register access packets and event packets.

4.4.4.4 Accessing Debug Registers

Every Debug Module can be configured, controlled and described by accessing a set of registers called debug registers. These registers can be read and written over the Debug Network through standardized messages, which are Debug Packets of the type REG (coded as 0b00). Accessing a debug register requires two packets, a request and a response packet as shown in the example in Figure 4.11. In this example, a 32 bit wide debug register inside the Debug Module with the DNP address 0x0042 is read from

the Debug Module at address 0x0041. The operation is successful, and the requested data is returned.

Depending on the type of access (read/write and register width) and the result of the operation (success or error) different request and response packets have been defined. The different packets can be distinguished by their `TYPE.SUB` value, a full list of which is available in the Open SoC Debug specification [95].

Register accesses are synchronous operations: a sender may not initiate another register access request until the response has been received. If asynchronous communication is required event packets can be used, which are described next.

4.4.4.5 Event Packets, a Multi-Purpose Data Container

Debug Packets used for register accesses are defined strictly from both a protocol and a data format point of view. For event packets that is not the case: the payload of an event packet can be used for any purpose, and there are no rules how and when packets are produced or consumed. This makes event packets a suitable container for a wide range of communication between debug modules. Typical uses include the transmission of asynchronous interrupt messages, the streaming of observation data, or the exchange of other large amounts of data between modules (such as memory contents).

Event packets set the `TYPE` field in the packet header to `EVENT` (coded as 0b10). Two packet subtypes (values of `TYPE.SUB`) are defined: `EV.CONT` and `EV.LAST`.

The two subtypes are used to support split event transmissions. As discussed earlier, Debug Packets are limited to an implementation-defined maximum size. Split transmissions transmit data over the Debug Network which does not fit into a single packet, but should still be kept together as if it was one packet. In this case the payload is split into multiple packets with identical source, destination and type fields. The first packet in a split transmission sets `TYPE.SUB` to `EV.CONT`, signaling that more packets are to follow. Only the last packet in the transmission sets `TYPE.SUB` to `EV.LAST`.

Supporting split event transmissions within the DNP significantly eases the implementation and documentation of Debug Modules. Since their semantics are defined in the scope of the DNP split packets can be handled in a generic fashion within a Debug Module. This makes the existence of split packets “invisible” to a sender or receiver on a higher level. Event packets which are too long can be split before the transmission, and reassembled upon reception, making it possible to describe event packets of arbitrary length according to the needs of the data transmission, and without taking care of the packet length restrictions.

The discussion of the event packets completed the implementation-agnostic overview of the Dia Engine. The following sections move closer to the implementation by presenting the hardware and software architecture, followed by an even deeper dive into the implementation itself.

To observe the software execution on this system the Dia Engine adds three Debug Modules: two Core Event Generators observe the software execution on the CPU cores, and a NoC Router Monitor (NRM) observes the NoC router. The observation data can be processed on-chip within the Diagnosis Processor (DIP). All Debug Modules can communicate with each other over the on-chip debug interconnect, which is implemented as unidirectional ring NoC (blue). Also shown are the debug NoC routers together with their DNP address. From a Debug Network perspective the on-chip debug interconnect is a subnet, which requires a Subnet Control Module (SCM). In our example the on-chip subnet is given the subnet address 0. The router at address 0.0 is different from the other routers in that it has not only ports for input, output and local traffic, but also a gateway port. All packets which do not have a destination within the local subnet 0 are sent through the gateway port to the host gateway. This component converts NoC packets into Debug Transport Datagrams (DTDs) and ultimately sends this data to the host. Incoming data from the host is likewise unpacked from DTDs into NoC packets and injected into the on-chip ring NoC.

At this point we could end the discussion—but the Dia engine can do more. The Memory Access Module (MAM) highlights how the Dia Engine supports software development beyond the collection and analysis of observation data. During the development it is often necessary to access the memories of the system, typically to load software into it. The Memory Access Module (MAM) gives developers this memory access, which is, of course, an intrusive operation and should not be done during the production use of the SoC.

Based on this introduction by example the following sections take a closer look at the key components of the hardware architecture: Debug Modules, the on-chip interconnect, and the off-chip connectivity.

4.4.5.1 On-Chip Debug Modules

On-chip Debug Modules closely follow the architecture described in Section 4.4.2 regarding their registers and interfaces to other components in the system. These topics are not discussed here any further. Instead we focus on two issues, both of which are related to time: clocking and synchronous behavior.

Larger SoCs often consist of multiple clock domains, chiefly to reduce the power consumption and to cater to a thermal budget. Since the Dia Engine (and particularly the Debug Network) spans across the whole chip, it also spans across multiple clock domains. This makes it necessary to cross clock domain boundaries at some point. While in theory the clock domain crossing can be performed anywhere, the recommended crossing opportunity exists within a Debug Module at the network interface. In this case the on-chip debug NoC is part of a different clock domain than the Debug Module, which is then in the same clock domain as the unit it observes. Crossing the clock domain boundary at this point is recommended because the network interface is a well-defined and common interface to all Debug Modules. Its FIFO characteristics simplify the building of the clock domain crossing hardware from standard and mature IP. Clock domain crossing are easy to get wrong and are notoriously hard to test.

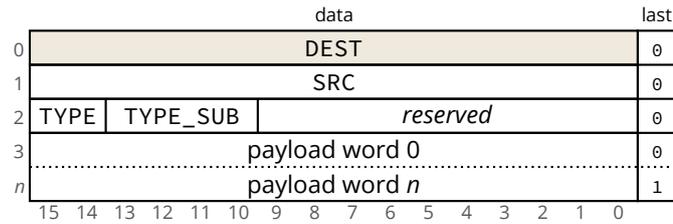


Figure 4.13: A Debug Packet when transferred over the on-chip NoC. Data words are mapped directly into NoC flits, an additional last bit indicates the start and end of the packet on the NoC. The NoC routes the packet based on the DEST field, i.e. the first flit of the packet.

Building on top of tested components, and increasing reuse are essential for productive engineering and high-quality results.

The second time-related topic in the implementation of Debug Modules is their role as “synchronous island.” The architecture of the Dia Engine (like all of DiaSys) is designed to cope well with large asynchronous SoCs. We achieve this goal by requiring no timing relationship (ordering) between events generated from different sources. However, in some cases having a well-defined time relationship between two data items is necessary. For example, the CPU Event Generator must be able to report the contents of a CPU register at exactly the time when a given program counter was executed. Within a Debug Module (and only there) collecting data with such a well-defined timing relationship is possible. Section 4.4.8 shows based on the example of the Core Event Generator how a specific implementation of that concept is realized.

4.4.5.2 On-Chip Debug Interconnect

The on-chip interconnect connects all Debug Modules and forms the on-chip subnet of the Debug Network. In our implementation we chose a unidirectional ring NoC as on-chip interconnect. The NoC is buffered, packet-switched and wormhole routed. Flits are 16 bit wide and each hop takes one cycle, which results in a NoC bandwidth of up to 2 byte/cycle.

These design choices are driven by our desire for a simple design which fulfills the performance and resource consumption constraints of our evaluation platform. (The evaluation platform is presented in more detail in Section 4.4.7 on page 98.) Depending on various factors, including the number of observed units, their operating frequency, and the chip’s target market the design choices will likely differ.

In Section 4.4.4.2 we discussed how the DNP can be layered on top of other protocols. When being exchanged over the on-chip NoC interconnect one Debug Packet is wrapped into one NoC packet. NoC packets consist of a sequence of flits, in every clock cycle one flit is transferred one hop. In typical network protocol stacks (such as TCP/IP/Ethernet) each protocol layer adds its own headers to a packet, thereby increasing the packet size and the overhead (i.e. the difference between transmitted data and payload). We optimized the transmission of DNP over NoC to result in only very little overhead

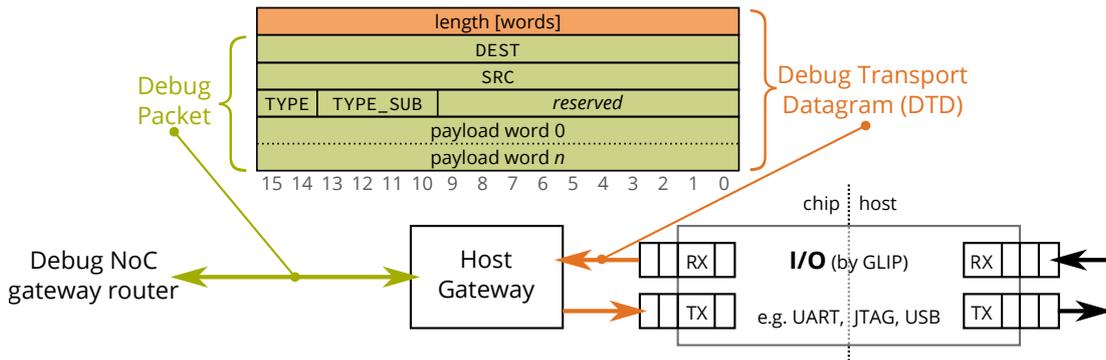


Figure 4.14: A Debug Transport Datagram (DTD) is wrapping a Debug Packet for transmission over the off-chip interconnect.

through multiple design decisions. First, we have a one-to-one mapping between Debug Packets and NoC packets, i.e. every NoC packet contains one Debug Packet. Second, the data width of the NoC (16 bit) matches the word width of a Debug Packet. Third, the destination address used to address network participants on the DNP layer is equal to the address in the NoC. The last property makes it possible to reuse the DEST field in the first word of a Debug Packet as destination address within the NoC (and hence as routing information).

The only information missing in a Debug Packet is the length of the packet, i.e. the number of words it consists of. Within the NoC this information is added in the form of a last bit added to every data word, indicating if the word is the last word in a packet. With this additional bit a flit is 17 bit wide, as shown in Figure 4.13. Together the optimizations reduce the overhead of transferring a Debug Packet over the NoC to only 6.25 percent, or one additional wire, independent of the size of the packet. Since no modifications of the payload in the data stream are necessary a simple implementation in hardware is possible.

4.4.5.3 Off-chip Connectivity

After gathering observation data and possibly performing initial processing on-chip the data is typically sent to a host PC. The host PC also communicates with the chip to initiate and control the analysis. All this communication is performed through the off-chip interface. Unfortunately “the” off-chip interface does not exist: every device provides different connectivity options, such as serial connections (UART), Ethernet or USB. With those different physical interfaces also come different protocols which must be followed to exchange data. To keep the hardware architecture of the Dia Engine flexible and retargetable to different devices we abstracted the off-chip interface through a FIFO interface. The actual off-chip connection is wrapped into a module which provides two FIFOs as an interface, one for receiving data (RX) and one for transmitting data (TX). The FIFOs are 16 bit wide, hence data needs to be provided as

16 bit wide data stream. Preparing data for transmission from and to the FIFO interface is the job of the host gateway.

The host gateway, as shown in Figure 4.14, connects on the one side to the gateway router of the debug NoC. (Figure 4.12 shows the architecture with more context.) Within the host gateway each Debug Packet as received over the NoC is converted into a Debug Transport Datagram (DTD). To be compatible with the FIFO interface the data width must be 16 bit, but a NoC flit is 17 bit wide. The host gateway hence codes the packet length information differently than the NoC. While the NoC uses the last bit to indicate the length of the packet, the DTD codes this information as a separate data word which precedes the Debug Packet. After this initial data length word the Debug Packet is sent unchanged, as shown in Figure 4.14. The DTDs are then passed on to the off-chip FIFO interface.

The details of how the data is exchanged between host and device FIFOs differ significantly depending on the actual off-chip interface (e.g. UART or USB). In the process of implementing the Dia Engine we have also developed several off-chip communication interfaces which we collected in the Generic Logic Interfacing Project (GLIP). Since the implementation details do not add value to the discussion of the Dia Engine we refer the interested reader to the web site at <https://www.glip.io>.

By treating the interface between chip and host as abstract FIFO we can continue our discussion at the host side. There software reads the data from its side of the FIFO and processes it further, something we explore in detail in the next section.

4.4.6 Host Software Architecture

The Dia Engine is unique in that it extends beyond the chip boundary. The host software is an equal part of the Dia Engine and its architecture follows the same design as the on-chip part. Debug Modules perform processing, a subnet of the Debug Network connects the Debug Modules on the host, and a gateway connects to the on-chip subnet of the Debug Network. But, of course, software is not hardware, a desktop PC is not a SoC. Implementing the Dia Engine architecture in software on a desktop PC requires a series of design decisions that tailor it to such an execution platform. These design decisions, and the resulting architecture of the host implementation of the Dia Engine, are explained in this section.

In addition to the general design goals of the Dia Engine, the host software architecture was created with a focus on the following aspects.

- The modularity and extensibility of the Dia Engine should also be visible in the software implementation. The resulting implementation should not be “one big block,” but a collection of communicating pieces.
- The communication between components should have high throughput and low overhead. The communication path with the lowest bandwidth should always be the connection between device and host.

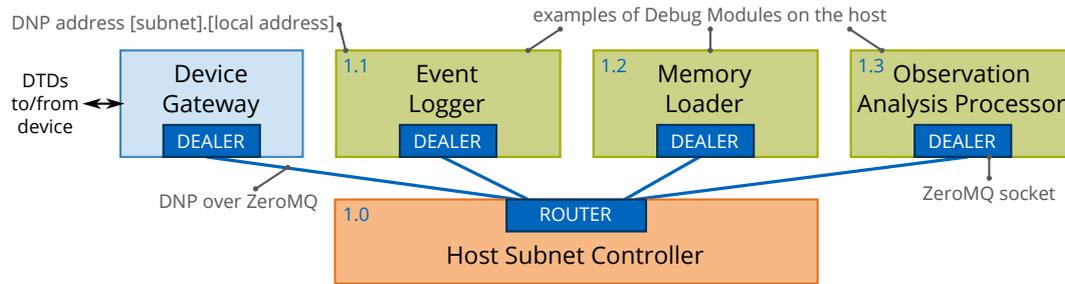


Figure 4.15: Architecture of the Dia Engine as implemented in software on a PC. A minimal configuration consists of the host subnet controller and the device gateway. The host subnet controller performs at the same time the duties of a Subnet Control Module and of a router of all host traffic, which exchanged over a ZeroMQ DEALER and ROUTER sockets (blue). The three Debug Modules shown on top in green are examples.

- The Debug Network on the host should be dynamic, it should be possible to add and remove network participants during runtime.
- The implementation should be able to make use of the computational power provided by today's PCs, especially its concurrency.
- The software design should follow and enable the best practices of software engineering, which make the creation of a reliable, high-quality implementation possible. This especially includes a strong focus on testability of functional and non-functional aspects.

These goals are reflected in the software design as presented in Figure 4.15. The central point of communication is the host subnet controller, which serves two purposes: it acts as Subnet Control Module (SCM) for the host subnet of the Debug Network, and it is the central communication hub. The Debug Network on the host is organized in a star topology, and the host subnet controller is the “center of the star,” i.e. the module which serves as connection point for all other modules and routes messages between them. The data exchange itself is handled by ZeroMQ, a library and protocol for messaging on top of other protocols like TCP. The host communication and ZeroMQ are explained in more detail in Section 4.4.6.1.

Debug Modules on the host implement the same specification as Debug Modules on the chip: they are addressed by a DNP address, they implement base registers and optionally the module-specific register set. Debug registers are read and written by REG messages as specified in the DNP, EVENT packets can equally be sent and received. While on the chip most Debug Modules are used to collect observation data, the Debug Modules on the host are mainly used for processing and control functions. Figure 4.15 shows three Debug Modules as examples: an event logger module writes incoming event packets into a file, events which typically represent the result of a Dia script-triggered observation task. The Diagnosis Processor fulfills the same task as its on-chip counterpart: it processes observation data and creates new events as result. And the

memory loader communicates with the on-chip Memory Access Module (MAM) to initialize the on-chip memories with program code.

The use of ZeroMQ together with an object oriented software design makes it possible to instantiate the components shown in Figure 4.15 in different ways to form an application. The components can be used within one (operating system) process. In this case they typically communicate over shared memory. Or every component can be made its own process. In this case the communication is typically handled over TCP or intra-process shared memory. To change the communication mechanism no code changes are required, ZeroMQ makes it possible to switch between them at runtime.

The next section takes a closer look at the communication on the host, and how the design decisions enable a modular, flexible and highly performant design.

4.4.6.1 Host Communication Over ZeroMQ

No matter where the Debug Network is implemented: the topmost layer is always the Debug Network Protocol (DNP). Below that we use ZeroMQ on the host.

ZeroMQ is a distributed messaging protocol and implementation provided as a software library. It provides an abstraction between the exchange of messages and the actual implementation of the message transport. More information, documentation, and the code itself is available at <http://zeromq.org/>.

The use of ZeroMQ is beneficial for two reasons. First, ZeroMQ makes it possible to abstract from the underlying transmission mechanism, and hence switch between TCP, in-process or intra-process shared memory communication without changing a single line of code. Second, the implementation of ZeroMQ is of very high quality and optimized in many ways for reliability and performance. While communication in general is relatively simple to implement, aspects like setting up a connection, handling unexpected disconnects, or retransmissions tend to be more difficult and are error-prone. By using ZeroMQ we can focus on the exchange of data and let the library handle the majority of tricky edge cases.

Communication in ZeroMQ is performed over sockets. Even though they share a name, these sockets are not to be confused with TCP or operating system sockets. ZeroMQ provides different types of sockets, which can be combined to create certain messaging patterns (e.g. broadcast or bidirectional communication). In our case we make use of two socket types, DEALER and ROUTER sockets, to create a network with bidirectional communication in a star topology, as shown in Figure 4.15. The “center of the star,” the host subnet controller, provides a ROUTER socket. All other participants in the network connect to the central point by using a DEALER socket.

As shown in Figure 4.9 on page 87 host communication over ZeroMQ is layered below the DNP. In addition to exchanging Debug Packets the host communication layer also handles the dynamic addition and removal of Debug Modules, which also includes the assignment of DNP addresses to joining Debug Modules. All this communication is handled between the ZeroMQ sockets by exchanging messages according to the host communication protocol as shown in Figure 4.16. ZeroMQ messages can consist of an arbitrary amount of frames, in our protocol three frames are used. The first frame

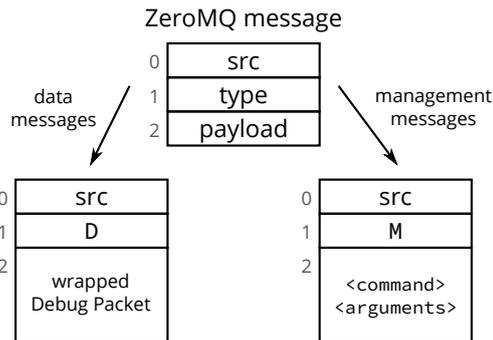


Figure 4.16: Structure of a ZeroMQ message in the host communication protocol. Messages consist of three ZeroMQ frames. The first frame is the “identity frame,” which contains the message source. The second frame determines the type of the message. Data messages (type D) carry a single Debug Packet, management messages (type M) contain a management command and possibly an argument to it.

is the “identity frame,” a ZeroMQ-provided mechanism to identify the sender of a message based on a unique address assigned to it when connecting. (The destination is not explicitly visible in the message on this level.) The second frame determines the message type, and the third frame contains payload depending on the message type.

Messages of type D are data messages which contain a Debug Packet as payload. Most messages exchanged on the host during the runtime of the Dia Engine are of this type.

Messages of type M are management messages. These messages are used to provide network management functionality below the DNP layer. This includes adding a Debug Module to the Debug Network (and by that, assigning a DNP address to it), removing modules, or informing the network participants about routing changes (e.g. an added gateway to another subnet).

Management messages warrant further discussion, together with the mechanisms that provide the dynamicity of the Debug Network on the host. A full list of management messages is presented in Table 4.1, in the following we take a closer look at two scenarios.

The first scenario we look at is adding a Debug Module to the Debug Network on the host. When a Debug Module connects to the host subnet controller its DEALER socket has already an identity, i.e. an address on the ZeroMQ-layer. It does not yet, however, have a DNP address assigned to it. The Debug Module therefore sends a management message `DIADDR_REQUEST` to the host subnet controller to “introduce itself,” and to request the assignment of a free DNP address. Upon reception of this message the host controller picks an unused DNP address within the subnet and sends it to the requesting module. It also updates its internal routing table by adding a mapping between the newly assigned DNP address and the identity of the Debug Module. Disconnecting a Debug Module on the host happens in a similar fashion using the `DIADDR_RELEASE` message.

Message	Source	Destination	Reply	Description
DIADDR.REQUEST	Debug Module	subnet ctrl.	<dnf-address>	Request a new DNP address from the subnet controller.
DIADDR.RELEASE	Debug Module	subnet ctrl.	ACK or NACK	Release the DNP address assigned to the source.
GW_REGISTER <subnet-addr>	device gateway	subnet ctrl.	ACK or NACK	Register the source of this message as gateway for all traffic intended for subnet <subnet-addr>.
GW_UNREGISTER <subnet-addr>	device gateway	subnet ctrl.	ACK or NACK	Unregister the source of this message as gateway.
ACK	any	any	none	Generic acknowledgement "operation successful."
NACK	any	any	none	Generic error message "operation failed."

Table 4.1: List of management messages exchanged over the ZeroMQ-based host communication protocol.

The second scenario involves a device gateway. After it has established the connection to the chip the device gateway registers the subnet of the Debug Network used on the device with the host gateway. For that it sends a `GW_REGISTER` message to the host controller, which (if successful) registers the device gateway as destination for all messages going to the subnet. When disconnecting from the subnet the gateway can unregister itself with the message `GW_UNREGISTER`.

4.4.6.2 Architecture Summary

In the previous sections of this chapter we presented the architecture of the Dia Engine on different levels. Starting with the implementation-independent architecture in Section 4.4.1 we moved closer to the implementation by discussing the on-chip hardware architecture in Section 4.4.5 and the software architecture in Section 4.4.6. We have shown how the general design goals of the Dia Engine tickle through the abstraction hierarchy to create a coherent implementation. We continue this journey in the following sections by looking at the implementation itself.

4.4.7 Implementation Overview

The Dia Engine as described is fully implemented both in hardware and in software. The discussion in this thesis focuses in the following on the key challenges and their solutions that were encountered while creating the implementation. In addition we

provide evaluations and measurement data which support the design choices made along the way.

Beyond that the details of an implementation are best studied by reading the source code itself, which is available online as part of the Open Soc Debug source code.

We first discuss two auxiliary topics, the platform that the development and testing happened on, and the architecture of the observed system. We then continue with the description of the key modules that make up the Dia Engine.

4.4.7.1 Target Platforms

Both the hardware and the software implementation are written in a highly portable way to be able to run the same code on different target platforms. For our evaluations we especially made use of the following targets, and the implementation is optimized to run on those platforms.

The hardware has been mainly been developed and evaluated on two FPGA boards, the “Xilinx Virtex UltraScale FPGA VCU108 Evaluation Kit” and the “Digilent Nexys 4 DDR” board. The VCU108 contains a Xilinx Virtex Ultrascale FPGA (XCVU095) with 1,176 thousand logic cells and 60.8 Mbit on-chip memory. It is accompanied by 8 GB of DDR4 memory and a large variety of off-chip interfaces. More information on the VCU108 board is available at <https://www.xilinx.com/products/boards-and-kits/ek-u1-vcu108-g.html>. The Nexys 4 DDR board is roughly ten times smaller, containing a Xilinx Artix 7-series FPGA (XC7A100T) with 101 thousand logic cells and 4.7 Mbit on-chip memory. On the board 128 MB DDR2 memory are available, together with (among others) UART and Ethernet I/O options. More information on the Nexys 4 DDR board is available at <https://reference.digilentinc.com/reference/programmable-logic/nexys-4-ddr/start>. The Xilinx 7-series and Ultrascale(+) FPGA families share a similar architecture so that most optimizations done in the design process apply equally to both families.

The host software was developed for PCs with a 64 bit x86 processor architecture (aka x86_64 or amd64) and a Linux-based operating system. The development and testing was performed using Ubuntu 16.04 and openSUSE Tumbleweed.

4.4.7.2 Observed System

Observing the software execution on a SoC of course requires a SoC design in the first place. And even though the Dia Engine is designed and implemented to be flexible and customizable to various observed systems, at some point the implementation becomes specific to the SoC. For the purpose of this thesis we built different SoC designs out of the same set of building blocks. These building blocks are collected in the OpTiMSoC [96] framework, co-developed by the author of this thesis.

All designs we used follow the tiled multi-/many-core design pattern. A mesh NoC connects a variable amount of tiles, most of which are so called compute tiles. Each compute tile contains between one and eight CPU cores, memory, and a network adapter connecting it to the NoC. The NoC is a 32 bit wide, packet-switched and

wormhole routed mesh network, in which the packets are routed according to the XY routing algorithm.

For the CPU cores we made use of the `mor1kx` implementation⁵ of the OpenRISC architecture (`or1k`). It is a multi-core capable 32 bit in-order RISC design with MMU and all “standard” features of microcontroller-class CPU, like separated L1 instruction and data caches and a FPU. Most instructions are executed within one cycle ($IPC = 1$). For the `or1k` architecture a full open source toolchain is available with the GCC compiler for C and C++, a linker, `binutils`, and the GDB run-control debugger. Also available are ports for different C standard libraries (`libc`), most notably `musl`⁶ (which also supports Linux) and `newlib`⁷ (best used for “baremetal” software which does not make use of an operating system).

Even though a port of the Linux operating system kernel to the `mor1kx` CPU core is available we created most applications on top of custom minimal library operating system, similar to programming a microcontroller. We did this to reduce the number of unknowns in the development process: our library operating system is simple enough to understand exactly how the execution of an application reflects on the underlying execution hardware, since for example no scheduler, no virtual memory and no supervisor/user-mode switches interfere with the execution flow. Developing, debugging and evaluating a debugging environment on a system where it is hard to know what the expected software execution should be can easily lead to misinterpretations and wrong results, something we tried to avoid. We claim that this approach does not reduce the validity of our results, but improves them. Initial tests of the Dia Engine together with Linux seem to support this hypothesis, even though we cannot report exhaustive results in this regard.

As the observed system is built from using the OpTiMSoC framework we were able to adjust it as needed for the evaluation of our DiaSys use cases. The exact configuration for each use case is described in Chapter 5.

Out of the components presented in this section we built our observed systems, into which we then integrated the Dia Engine components. In the following sections we take a closer look at them.

4.4.8 Implementation of the Core Event Generator (CEG)

The Core Event Generator (CEG) observes a single CPU core and creates event packets when an observation matches a configured trigger condition. As this event generator is the most commonly used one the implementation details are directly impacting the performance and resource utilization of the Dia Engine. This section gives insight into the implementation of the Core Event Generator.

All implementation work on the CEG is guided by the following functional requirements.

⁵The `mor1kx` CPU core is available online at <https://github.com/openrisc/mor1kx>.

⁶<https://www.musl-libc.org/>

⁷<https://sourceware.org/newlib/>

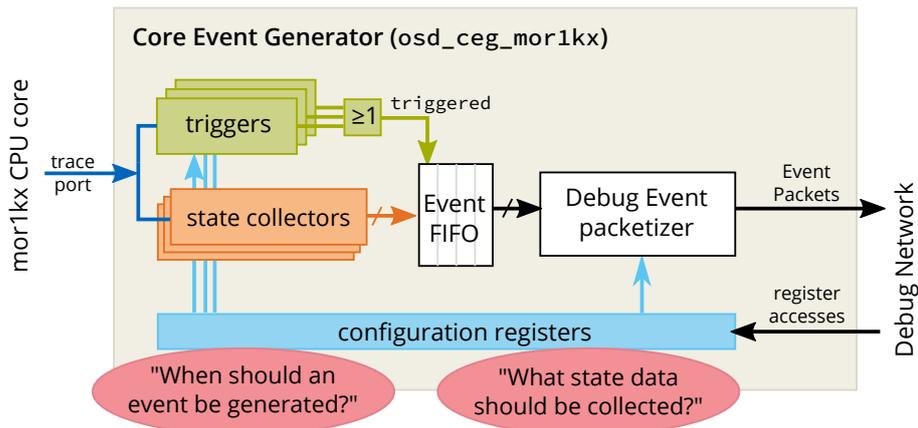


Figure 4.17: Implementation of the Core Event Generator (CEG) module in hardware.

- It must be possible to capture at least two events which are triggered immediately after each other.

As an example, two such events would be generated if two triggers are configured, one at program counter x and one at the next program counter $x + 4$.⁸

- Events may be dropped if the Debug Interconnect cannot accept the generated data in time.
- Each executed instruction may cause only one observation event to be generated, even if two or more trigger conditions hold.

For example, only one event is generated if the CPU executes the program counter $0x1000$ and one trigger is configured for $PC == 0x1000$, and another trigger is configured for $0x0000 \leq PC \leq 0x5000$.

In addition to the functional requirements, which are visible to the user of the Dia Engine, the following non-functional requirements were used in the process of the implementation.

- The implementation of the Core Event Generator should target the mor1kx CPU core implementation. To ease the porting to different CPU core (implementations), all generic parts should be encapsulated in a reusable way.
- The implementation should be optimized for Field Programmable Gate Array (FPGA) synthesis on Xilinx 7-series and newer⁹ FPGAs.
- The implementation should be configurable at synthesis time in various ways to enable the exploration of trade-offs between features and resource consumption.

⁸The example assumes byte addresses and a 32 bit/4 B word width.

⁹The Xilinx 7-series, Ultrascale, and Ultrascale+ FPGA families share a very similar architecture so that most optimizations done in the design process apply equally to both families.

name	width (bit)	description
valid	1	the trace data is valid
pc	32	program counter value
insn	32	executed instruction (opcode)
wben	1	register write back active
wbreg	5	write back register address
wbdata	32	data written to the register

Table 4.2: For the CEG relevant signals of the trace port for the mor1kx CPU core. All data flows from the CPU to the CEG.

A block diagram of the resulting implementation of the Core Event Generator is shown in Figure 4.17. We start the discussion with the interfaces of the module. The trace port carries data from the CPU core to the CEG module (left), and the standard Debug Interconnect Interface (DII) connects the module with the rest of the Dia Engine (right).

The CPU trace port is in its implementation specific to the mor1kx CPU core, but similar data could be obtained from other CPU cores as well. The relevant signals¹⁰ are listed in Table 4.2: in addition to the program counter and the opcode of the executed instruction the trace port includes signals to observe writes to the register file of the CPU.

The inner structure of the CEG (shown in the center of Figure 4.17) can be split into four parts: triggers, state capture, event packetization, and configuration.

All trigger modules work in parallel and continuously observe the data coming from the trace port and check if their configured trigger condition matches. Our implementation provides the following four trigger types.

- Program counter exact match: `observed_pc == conf_pc`
- Program counter range match:
`observed_pc >= conf_pc_lower && observed_pc <= conf_pc_upper`
- Instruction value (opcode) match with an optional mask:
`(observed_instruction & conf_mask) == conf_match`
The mask can be used to filter out the opcode from an instruction value (“assembly statement”), and hence trigger for example at all `l.jal` jump instructions.
- Time interval: a trigger is periodically generated after a configured amount of cycles. This trigger is especially useful for sampling use cases.

Not only the trigger modules, but also the state collector modules observe the trace port. They use, among other things, the observed data to reconstruct parts

¹⁰The interested reader may find the full interface definition in the source file `mor1kx_trace_exec.sv`.

of the internal state of the CPU. The following state information is collected by our implementation and can be included in observation events.

- The current time in the form of a 32 bit wide timestamp.
- The currently executing program counter.
- The program counter of the jump target (the “next” program counter)
- The address of the most recent function call, i.e. the last `l.jal` or `l.jalr` instruction.
- The general purpose register (GPR) Collector module creates a shadow register file by observing the register writes that the CPU performs. The main usage for the GPR collector is to access arguments to a called function, which are stored on the stack.

We continue our journey through the implementation of the Core Event Generator with the event FIFO. This FIFO sits between the state collector and the packetizer. If a trigger fires the FIFO stores all state data together with the index of the firing trigger.

On the consuming side of the FIFO is the packetizer. The packetizer first uses the index of the firing trigger to determine which parts of the collected state data the user wants to be included in the event packet, what DI address the packet should be sent to, and what event identifier should be assigned to it. Then the packetizer continues by reading the relevant state data from the FIFO to construct an event packet, which is then sent out to the Debug Interconnect.¹¹

The operation of the Core Event Generator is highly configurable at runtime. The configuration is performed by writing a set of debug registers. The configuration register module (at the bottom of Figure 4.17) makes this data available to the triggers as well as the packetizer.

The Core Event Generator exposes a number of parameters, which can be used at synthesis time to explore the trade off between features and logic size in various ways.

- The number of parallel triggers is configurable.
- The number of observed CPU general purpose registers is configurable.¹²
- The width of the timestamp (in bit) is configurable.

The most “expensive” resource in chip designs are memory resources. Depending on the FPGA vendor and family different dedicated memory resources are embedded within the FPGA fabric. For example, Xilinx 7-series devices contain memory within Block RAM, Distributed RAM, and LUTRAM [98]. The implementation design of the

¹¹This is a slight simplification: in fact, one event is transmitted as multiple *split packets* if the payload size exceeds a limit imposed by the Debug Interconnect.

¹²The or1k ISA allows implementers to build 32 bit CPU cores with up to 32 general purpose registers [97, Section 4.5], commonly used by a compiler are the first 16 registers.

parameter	chosen value
number of concurrent triggers	4
number of GPRs	12
depth of the event buffer	2

Table 4.3: Parameter values chosen for the CEG.

module heavily influences how well the synthesis tool is able to map the design to such FPGA memory resources. The Implementation Spotlight 1 gives an insight into how this was achieved for the CEG implementation.

4.4.8.1 Parametrization of the CEG

The CEG can be parameterized at synthesis time in various ways to trade off resource consumption with provided functionality. Finding a “sweet spot” in this trade-off requires answering two questions for every parameter value: “How much (logic area) does it cost?” and “How valuable is this parameter setting to the user of the Dia Engine?”

Answering the first question is relatively easy: a synthesis run with the given parameter value will produce the required data. Finding an answer to the second question is harder. The Dia Engine is designed to be a general purpose execution platform, therefore no single use case can be used to determine the value of a feature; instead, a common set of use cases must be analyzed. Even by following this approach the “value” assigned to a parameter value remains a subjective measure—a fact we must acknowledge, but not worry too much about: in an FPGA design, adjusting the parameterization only requires a fresh synthesis run.

Table 4.3 presents the parametrization we chose for the CEG. The following sections discuss how these values were determined.

Number of concurrent triggers The CEG features multiple triggers which concurrently observe the CPU.

We first tried to answer the question “How many concurrent triggers are typically used?” Since we cannot resort to a large body of existing Dia scripts to answer this question, we choose an alternative approach: we analyzed a large collection of scripts from a similar tracing language, SystemTap. This collection of 177 scripts¹³, which is distributed as part of SystemTap, has been developed over a period of 10 years and represents common analysis tasks executed on Linux systems.

We analyzed every script and counted the number of “probes” in it. A “probe” in SystemTap is comparable to a trigger in the CEG, therefore the number of probes in

¹³165 of the 177 scripts were used in the analysis. The directories `stapgames` (games as proof-of-concept) and `tapset/general` (not standalone scripts) were excluded.

Spotlight 1: Optimizing the memory usage of the CEG on an FPGA

A significant portion of the CEG implementation is memory: memory to store the runtime configuration, memory to create shadow data structures like the GPRs and the stack, and memory to decouple the event generation from the (slower) Debug Interconnect.

In such a design it is of critical importance to enable the FPGA toolchain to map the design well into dedicated memory resources on the FPGA device. In the design of the CEG the following rules were followed.

1. **Strictly follow the Vivado code templates.** In the user guide 901 [99] Xilinx describes in detail how Verilog or VHDL code must be written when describing certain types of memories (e.g. single-port RAM). If these templates are closely followed Vivado recognizes the code and infers memory primitives such as Block RAM or Distributed RAM from it. Otherwise all memories are mapped into SLICE registers, which can increase the resource consumption by a factor of ten or more.
2. **Make memories deep, not wide.** The memory primitives provided by Xilinx are all deeper than wide. For example, the 36 kbit Block RAM (BRAM) primitives can be up to 72 bit wide and store in this configuration up to 512 data words. Similar rules apply for Distributed RAM and LUTRAM. It is hence more resource efficient to store a given amount of data as multiple words sequentially, instead of storing it as one large data word. This comes, of course, at the cost of additional latency when accessing the data, and typically requires more design effort, e.g. to create more complex read and write state machines.
3. **Be aware of the width and depth limitations of memory primitives.** All memory primitives have limitations regarding their maximum data width and depth. If one of the limits is exceeded, another memory primitive is instantiated, instantly doubling the resource utilization. On the other side, staying below the limit does not reduce the resource cost either. For example, storing 37 bit wide data in a BRAM incurs the same resource cost as storing 72 bit wide data, while storing 73 bit instantly doubles the resource consumption. The Xilinx user guides 473 and 474 [100, 98] describe the limits in more detail.

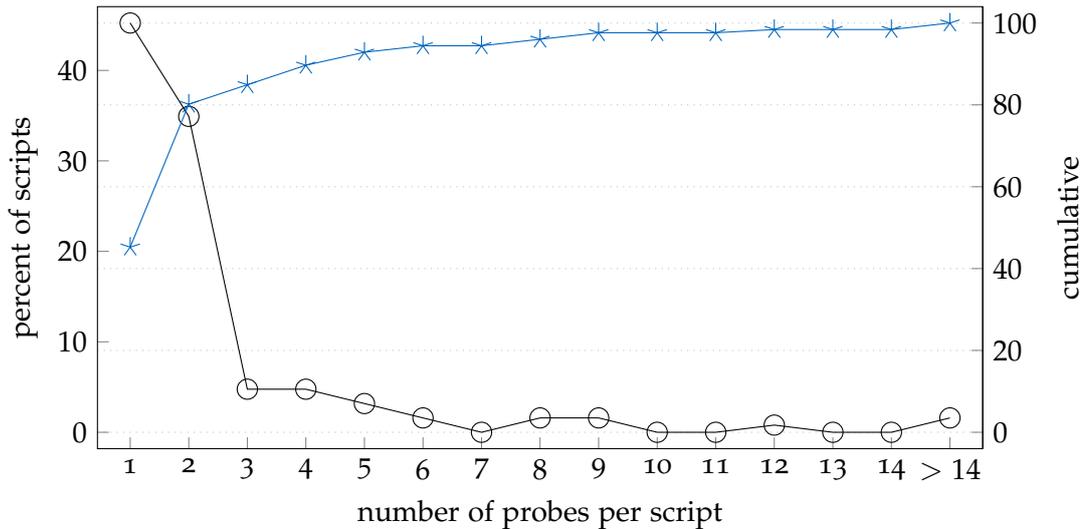


Figure 4.18: Number of probes (similar to concurrent triggers in the Dia Engine) per SystemTap script in the SystemTap example collection.

a SystemTap script gives a good indication for the number of concurrent triggers if a similar script would be implemented in Dia.

The outcome of the analysis is shown in Figure 4.18. 90 percent of the SystemTap scripts in the example collection use four or less concurrent probes, and 97 percent of scripts rely on nine or less probes.

With that result we can now turn our attention to the cost side of things: how much logic area is consumed by each additional trigger? To answer this question we synthesized and implemented a full SoC design containing the CEG for a Xilinx 7-series device while varying the number of concurrent triggers between one and twelve.

The analysis shows that between two and eight concurrent triggers, each trigger requires typically 67 Slice Registers. Between nine and twelve triggers no additional registers are used.

The analysis of the combinatorial logic usage, expressed in Slice Look-Up Tables (LUTs), does not give a conclusive result: for some numbers of concurrent triggers the number of required LUTs increases, while for others it decreases by roughly the same amount. We attribute this fluctuation to differences in the non-deterministic synthesis and implementation process.

Given this data we chose to parameterize the CEG with four concurrent triggers.

Number of captured GPRs Each captured GPR increases the size of the CEG on average by 80 registers and 95 LUTs. On 32 bit OpenRISC architectures the following registers are of special interest when observing the program execution [97, p. 336].

- R1 (SP) and R2 (FP), which store the stack and frame pointers,

Instance	Total LUTs	Logic LUTs	RAM (kbit)	DSPs
Core Event Generator (CEG)	1493	1453	0	0
configuration registers	562	522	0	0
triggers	52	52	0	0
state collectors (incl. timestamp)	93	93	0	0
event buffer	651	651	0	0
DI packetization	137	137	0	0

Table 4.4: Hierarchical resource utilization of one instance of the Core Event Generator (CEG) after implementation with Vivado 2018.1 (default settings) for a Xilinx XCVU095 FPGA.

- R₃ to R₈, which store six function parameter words,
- R₉ (LR), the link register (the address to jump back to after a function call)
- R₁₀ (TLS), which can be used to identify the currently executing thread, and
- R₁₁ (RV), the function return value.

The remaining registers R₁₂ to R₃₁ have no special meaning in the Application Binary Interface (ABI) and are used for temporary values. The CEG is therefore parameterized to capture registers R₀ to R₁₁, i.e. 12 GPRs.

Depth of the event buffer The largest contributor to the size of the CEG module is the event buffer. This buffer decouples the event generation from the packetization logic. Its size influences how well the CEG can handle bursts of triggered events.

A look at the data rates at the producing and consuming side explains this. At the producing side of the buffer, up to one event per clock cycle can be generated. At the consuming side the packetizer requires at least four cycles to transmit a minimum-sized packet to the Debug Interconnect (DI). Depending on the configuration of the trigger, e.g. if captured state or a timestamp is included in the packet, the number of cycles required to send out an event can further increase.

An evaluation of the CEG size depending on the depth of the event buffer shows that increasing the event buffer depth by one entry typically adds 419 registers and 419 LUTs to the size of the CEG (using four concurrent triggers and twelve GPRs).

In order to reduce the overhead of the CEG module, while still fulfilling the requirement outlined in Section 4.4.8, we dimensioned the event buffer to be able to store two events.

4.4.8.2 Resource Utilization of the CEG

The resource consumption of a single CEG module (attached to a single processor core) using the parametrization shown in Table 4.3 is presented in Table 4.4. Note

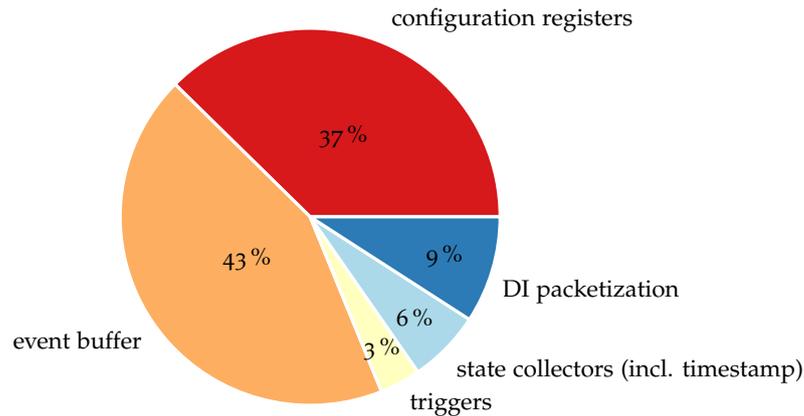


Figure 4.19: Resource utilization in LUTs for one instance of the Core Event Generator. See Table 4.4 for detailed data values.

that the LUT figures used in the dimensioning of the CEG are post-synthesis, while the numbers presented in this table are post-implementation, i.e. taken out of a fully implemented design.

Also notable is the resource consumption of the event buffer and the config registers, both components which mainly consist of memory. Even though we have designed them in a way that allows Vivado to place the memory cells into block RAM (BRAM), Vivado chose not to do so. This decision depends on the utilization of the device and is subject to a global optimization performed during the implementation step (in Vivado). Due to the width of the event buffer, Vivado chose to implement it in normal LUTs. The configuration registers are more narrow, and have been placed into 40 LUTRAM cells.

4.4.9 Implementation of the Debug Interconnect

The Debug Interconnect is the on-chip implementation of the Debug Network. It is implemented as a NoC with unidirectional ring topology, a single (physical) channel, and buffered routers. The topology was chosen mainly due to its simplicity and the comparably low overhead. The data links are 16 bit wide, each hop between two routers takes one cycle. The routers follow a standard implementation pattern, with no pipelining and round-robin port arbitration.

4.4.10 Implementation of the Diagnosis Processor (DIP)

The Diagnosis Processor (DIP) is a freely programmable general-purpose processing node on the chip. Like any processor design, it sacrifices computational density for flexibility.

The Diagnosis Processor design is extended from a standard processor template like it is used in the observed system. The main components, shown in Figure 4.20, are

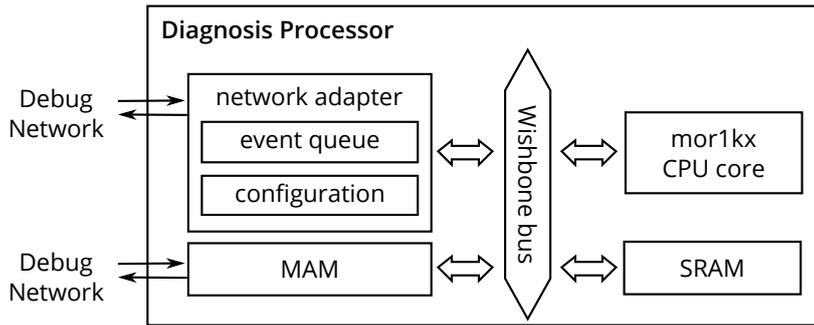


Figure 4.20: Block diagram of the Diagnosis Processor (DIP), a freely programmable processing node.

Instance	Total LUTs	Logic LUTs	RAM (kbit)	DSPs
Diagnosis Processor (DIP)	3439	3418	1584	3
mor1kx CPU core w/ L1 I\$	2723	2723	432	3
network adapter	243	222	0	0
SRAM (128 kB)	130	130	1152	0
MAM (for program loading)	342	342	0	0

Table 4.5: Hierarchical resource utilization of the Diagnosis Processor (DIP) after implementation with Vivado 2018.1 (default settings) for a Xilinx XCVU095 FPGA.

a single mor1kx CPU core and an 128 kB SRAM block acting as program and data memory.

The main extension is the special network adapter, which queues incoming event packets, and sends outgoing event packets to the debug interconnect. To write the program into the SRAM block the Diagnosis Processor contains a standard Memory Access Module (MAM).

The diagnosis processor design is optimized for low resource consumption; the resource consumption numbers on a Xilinx XCVU095 FPGA after the implementation step are given in Table 4.5. The SRAM size is large enough to support all case studies presented in Chapter 5, but still rather small to keep the area cost of the diagnosis processor low.

The diagnosis processor runs user code given as transformation actor in a Dia script. To support the execution of this code we have implemented a minimal runtime environment based on the OpTiMSoC “baremetal” runtime environment (a microcontroller-like minimal runtime system). Without any custom transformation actor code the compiled binary 50 kB large, leaving 78 kB of memory for custom code and data. The reception of one packet with five words takes 3000 cycles, which includes the allocation of memory for the received packet, and copying the data from the buffers in the network adapter to the main memory. The sending of an equally sized packet takes 1340 cycles.

In summary, the Diagnosis Processor has been designed with hardware cost in mind, causing more work on the software side. We show in Chapter 5 that this trade-off is suitable for the case studies we have developed.

If needed in the future, multiple hardware offload options exist. To reduce the amount of time spent on copying data by the software, a DMA engine and hardware buffer management can be introduced. To further reduce the overhead in the software, i.e. the time not spent on executing the transformation actor, a hardware run queue or a hardware scheduler for transformation actors can be added.

4.4.11 Implementation of the Event Counter (CNT)

In cases where the flexibility of a programmable execution unit like the Diagnosis Processor is not needed, special-purpose execution units can be employed. The Event Counter (CNT) is a simple example of such a module, which we have include into our Dia Engine design.

This special-purpose execution unit does what its name suggests: it counts events. It is typically used in profiling scenarios, e.g. to count the number of incoming packets or calls to a function. To retrieve the counter value, a special event is sent to the module, which then answers with the requested data.

The implementation of the CNT module requires 201 LUTs. It operates in streaming mode, i.e. it can count events as fast as the Debug Interconnect can provide them.

4.4.12 Implementation of Dia Engine Host Software

The host software of the Dia Engine is implemented according to the architecture described in Section 4.4.6. In the following we take a glimpse at the implementation itself and highlight some implementation techniques. The implementation work discussed here has been fully incorporated into the Open SoC Debug software reference implementation and is available online at <https://github.com/opensocdebug/osd-sw>.

Overall the software implementation consists of roughly 10,000 lines of C code. It is structured into a library called `libosd`, tools which make use of this library, and bindings to the Python programming language.

4.4.12.1 `libosd`: The Dia Engine Host Software as Library

The code is written following the object-oriented design paradigm. By preferring composition to inheritance dependencies between parts of the code are reduced and the testability is improved.

`libosd` provides functionality which can be grouped into three categories: base building blocks, Debug Module clients, and high-level APIs. The first category of base building blocks includes classes like `osd_hostmod` (implementation of a Debug Module on the host), `osd_hostctrl` (implementation of the host subnet controller), or `osd_gateway` (a device gateway). In the second category Debug Module clients provide a low-level API to access the functionality of a specific Debug Module on-chip.

Examples includes `osd_cl_mam` to access the Memory Access Module, or `osd_cl_scm` to access the Subnet Control Module. Classes of the third category make use of classes in the other two categories to provide high-level functionality. For example, the `osd_memaccess` class uses the MAM and SCM modules through the `osd_cl_mam` and `osd_cl_scm` classes to write and read memories on the chip. Ultimately the user of `libosd` can then make use of the function `osd_memaccess_loadelf()` to transfer a program binary in the ELF format to the system's memory.

4.4.12.2 Tools to Interact with the Dia Engine

The functionality provided by `libosd` can be used in applications to interface with the Dia Engine or extend its functionality. As part of our implementation we provide applications for the most essential tasks performed on a DiaSys-enabled system. These tools are `osd-device-gateway`, `osd-host-controller`, `osd-systrace-log` and `osd-target-run`. `osd-device-gateway`, and `osd-host-controller` do exactly as their name suggests: they implement a device gateway to a target (either an FPGA or a simulation), and implement a host controller. These tools can be implemented with very little code as they only wrap around the corresponding classes. For example, `osd-host-controller` consists only of 55 lines of C code (not counting blank and comment lines). `osd-systrace-log` shows the standard output stream (STDOUT) of an application running on the SoC. And finally `osd-target-run` writes an Executable and Linking Format (ELF) file to a memory in the target system and starts the CPUs.

Writing tools in C is a robust and highly performant way of implementation. In many cases, however, flexibility and implementation productivity are the most important design goals. For those scenarios `libosd` provides Python bindings, something we will explore next.

4.4.12.3 Python Bindings

C is a time-proven programming language which enables the creation of high-quality, performance optimized applications. By writing `libosd` in C we were able to make it a solid foundation which we can build on. For applications on these "higher layers" we were looking for a programming language which provides a higher productivity in writing code, at the cost of less reliability or performance. A good fit in this regard is the Python programming language.

To be able to interface with the Dia Engine through Python we created a Python extension module which wraps `libosd`. This extension is created with Cython¹⁴ and can be used from any Python 3 application by including the `osd` module. To show how simple the creation of an application interfacing with the Dia Engine is in Python we have included Listing 4.3. The listing contains the full source code for a tool to run an application in an ELF file on a target connected over USB 3. The code first starts the host controller listening at the TCP port 9537. It then initiates a device gateway, which connects to the host controller and to the device over USB 3. Finally, the `MemoryAccess`

¹⁴<http://cython.org/>

4 DiaSys Design and Realization

```
1 import osd
2
3 def main():
4     log = osd.Log()
5
6     # start host subnet controller
7     hostctrl = osd.Hostctrl(log, 'tcp://0.0.0.0:9537')
8     hostctrl.start()
9
10    # start device gateway and connect over USB 3 to an FPGA
11    gw = osd.GatewayGlip(log, 'tcp://localhost:9537', 0, 'cypressfx3')
12    gw.connect()
13
14    # find memories in subnet 0 (i.e. the target device)
15    memaccess = osd.MemoryAccess(log, 'tcp://localhost:9537')
16    memaccess.connect()
17    memories = memaccess.find_memories(subnet=0)
18
19    # stop all CPUs in subnet 0
20    memaccess.cpus_stop(0)
21
22    # load ELF file
23    memaccess.loadelf(memories[0], '/some/elf/file.elf', verify=False)
24
25    # start CPUs in subnet 0 to run program
26    memaccess.cpus_start(0)
27
28    # disconnect from the system
29    memaccess.disconnect()
30    gw.disconnect()
31    hostctrl.stop()
32
33 if __name__ == '__main__':
34     main()
```

Listing 4.3: Complete code listing for a Python application loading a ELF file into the memory of a target and starting the CPUs. The application includes the host controller and the device gateway to work without additional components (standalone).

class is used to list all memories available in the system, stop the CPUs in the subnet, initiate the first memory found with the ELF file, and then start the CPUs again.

This example used the high-level classes of `libosd`. Similarly the `Hostmod` class can be used in Python to implement a host Debug Module with a couple lines of code. This functionality is used to connect the Dia Compiler with the Dia Engine, and to prototype and implement special Debug Modules for data analysis tasks, something we explore further in Chapter 5.

4.4.12.4 Quality Assurance of the Software Implementation

Reliability is key for any debugging tool: nobody wants to debug the debugger at the same time as debugging the actual application on the SoC. We therefore went to great lengths to improve the quality of the software implementation through extensive testing. Most test coverage is achieved by unit tests, which target a single class. Unit tests are written using the `check`¹⁵ framework and amount to over 3,000 lines of C code (in addition to the 10,000 lines of code which are being tested). The unit tests target `libosd` and achieve a line code coverage of 84 percent¹⁶.

In addition to testing for functional issues the unit tests are also used to check for non-functional issues, especially in regard to memory usage. Running the tests with Valgrind's `memcheck` tool¹⁷ can uncover especially issues with memory allocations and deallocations on the heap. Running the tests with Address Sanitizer (ASan)¹⁸ primarily finds buffer overflow issues both on the heap and on the stack. Our code is free of any such issues (as reported by the tools).

To maintain the code quality over time all source code changes are automatically tested with all mentioned tools when being checked into the git repository.

4.4.13 Summary: The Dia Engine

The Dia Engine is the “workhorse” in DiaSys: it executes Dia scripts using both on-chip and off-chip components. The power of the Dia Engine comes from its careful design with well-defined abstractions. It starts with the implementation-agnostic definition of the components of the Dia Engine, most importantly the Debug Modules and the Debug Network.

Debug Modules are generic containers for data collection and data processing functionality. They provide a common interface towards the Debug Network, which makes them discoverable across the system and ensures unhindered communication between them.

The Debug Network provides a shared communication platform between all Debug Modules. It specifies a segmented network topology with subnets, and a shared communication protocol. This communication protocol supports both control communication through its `REG` packet type, and streaming or asynchronous traffic through its `EVENT` type. Control communication is modeled as register accesses, a common abstraction in embedded systems.

The Dia Engine spans from the chip across the chip boundary to the host PC. We have designed and implemented both parts, and presented the design choices in Section 4.4.5 and 4.4.6 respectively. The hardware implementation is written in SystemVerilog and runs on FPGAs and in cycle-accurate simulation. It extends a tiled multi-core system

¹⁵<https://libcheck.github.io/check/>

¹⁶Line coverage as calculated and reported by Codecov, see <https://codecov.io/gh/opensocdebug/osd-sw/tree/master/src> for the full report.

¹⁷<http://valgrind.org/docs/manual/mc-manual.html>

¹⁸<https://github.com/google/sanitizers/wiki/AddressSanitizer>

with diagnosis components to observe the CPUs and the memory, and to perform on-chip processing of observations using the Diagnosis Processor.

The implementation on the host PC is composed of individual components which can communicate over TCP. The shared implementation parts are contained within a software library (`libosd`) written in C. Python bindings make it easy to extend the Dia Engine with new Debug Modules, as the bindings are sufficiently abstracted to focus on the functionality of the Debug Module, without the need to worry about the underlying communication infrastructure.

Overall, the Dia Engine has shown to be a flexible and reliable runtime environment, not only for our experiments, but also for use cases beyond our research goals. We have therefore released its specification and implementation under an open source license in the Open SoC Debug project.

4.5 Summary: DiaSys Design and Realization

Based on this general approach we have further refined and implemented DiaSys in three components: the Dia Language, the Dia Compiler, and the Dia Engine. The Dia Language is a domain-specific programming language to describe software diagnosis tasks. A Dia script, a piece of code written in the Dia Language, consists of two parts: first, events are declared using a SQL-like syntax. These events can be either created through software observations on the chip, or carry intermediate data or analysis results between actors in the dataflow program. Transformation actors contain the data analysis program, split up into individual actors written in a C-like language, which is extended with facilities to wait for events and to send out events.

A Dia script is then transformed by the Dia Compiler and executed by the Dia Engine. The event declarations are compiled into configurations for the event generators. The code within the transformation actors is analyzed and either transformed into standard, platform-independent C-code, or into configurations for special-purpose execution units. A semantic analysis step based on tree pattern matching is able to determine the mappability of code to special-purpose execution units (such as event counters); if no such mapping can be found, general-purpose execution units, such as the Diagnosis Processor on-chip, or equivalent execution modules off-chip can be used.

Finally, the compiled Dia script is executed by the Dia Engine. Designed as flexible runtime environment for Dia scripts it spans from the chip to the host PC. By abstracting away implementation-defined details through concepts like Debug Modules and the Debug Network the whole Dia Engine can be seen as “one unit,” independent of where the actual data processing components reside. By necessity, the data collection is performed on-chip where the software runs.

We have designed and implemented all parts of the Dia Engine, which enables us to study various aspects from implementation effort to performance and applicability to certain debugging and testing use cases. We present selected use cases in the next chapter.

5 Case Studies

The previous chapters presented the concept, design, and realization of DiaSys. It is now time to take DiaSys on a “test drive” to explore how it performs in different scenarios.

DiaSys is designed to be a general-purpose approach. Since exhaustively evaluating such an approach in full generality is infeasible, we picked three representative case studies. The main technology we compare DiaSys to are traditional tracing systems. Hence, our case studies resemble tasks for which tracing systems are commonly used today: debugging and testing for time-dependent functional bugs (like race conditions), and the generation of runtime statistics (e.g. profiles).

5.1 The “Chip Design” For Our Case Studies

Two of our three case studies run on an FPGA with our Dia Engine implementation. Just like in a production chip, the parametrization of the Dia Engine must be able to support multiple use cases. We have hence created one “chip design,” which we use in both case studies.

Our “chip design” is shown in Figure 5.1. The observed system follows the tiled multi-core design pattern. Four compute tiles are connected by a 2×2 mesh NoC. The NoC uses 32 bit wide bidirectional links, is wormhole switched, and XY routed. Each compute tile consists of two `mor1kx` CPU cores, a distributed memory block, and a network adapter, all connected by a 32 bit wide Wishbone bus. The `mor1kx` CPU core implements the `or1k` (OpenRISC) instruction set architecture (ISA) and has a standard five-stage in-order pipeline, a FPU, and Level 1 instruction cache (the L1 data cache is disabled). Each distributed memory is a 128 MB large DRAM block. The network adapter is capable of remote memory accesses (remote loads/stores), message passing, and direct memory access (DMA).

The observed system is amended with the Dia Engine implementation. All components of the Dia Engine are connected by the Debug Network, which is implemented on the chip as 16 bit wide unidirectional ring NoC. To observe the software execution each CPU core is attached to one Core Event Generator (CEG), as described in Section 4.4.8. For the on-chip processing of observation data we have integrated two modules: a Diagnosis Processor (DIP), and an Event Counter Module (CNT).

In addition, auxiliary on-chip modules make the system accessible from the host PC. The Software Trace Module (STM) transports the standard output (i.e. the “`printf()` output”) of the observed applications to the host PC. The Memory Access Module (MAM) is used to load the program into the distributed memory within each tile. Fur-

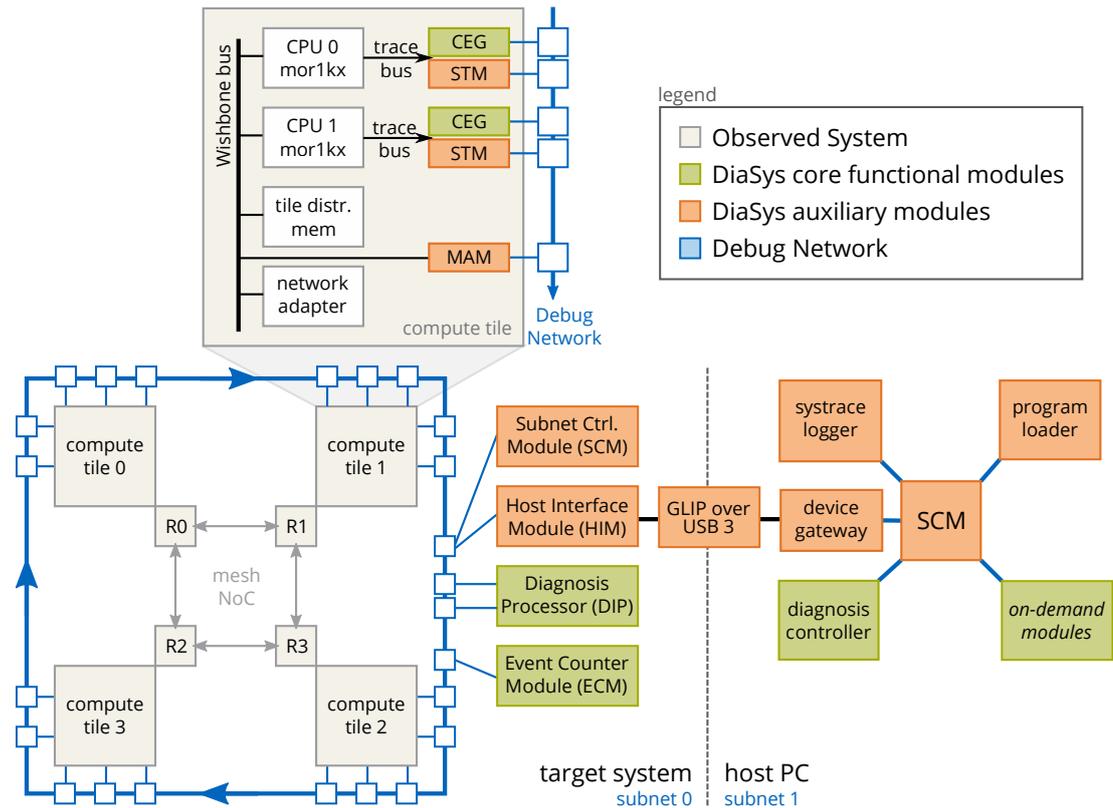


Figure 5.1: The “chip design” used for the case studies on DiaSys. The observed system consists of four dual-core compute tiles with distributed memory, connected by a 2×2 mesh NoC. Shown in green and orange are components of the Dia Engine, and blue are the Debug Network components. Orange components are “auxiliary” components, which are necessary for the operation of the Dia Engine, but do not provide observation or analysis functionality. Key components which are involved in the execution of a Dia script are shown in green.

Other components are the Subnet Control Module (SCM) as described in Section 4.4.3.3, and the Host Interface Module (HIM), which connects the on-chip components of the Dia Engine to the host PC over USB 3.0.

On the host, the SCM is the center of the Debug Network. It establishes the connection to the on-chip components through the device gateway, which forwards the data to the USB interface. All Debug Modules on the host are connected to the SCM. The program loader and the systrace logger are helpers to load the program into the target memories and to log the standard output of the observed applications. The diagnosis controller initiates and controls the execution of a Dia script. Depending on the executed Dia script further “soft” diagnosis processors on the host are instantiated, represented by the “on-demand modules” in Figure 5.1.

All hardware components run on a Xilinx VCU108 evaluation board, containing a Xilinx UltraScale XCVU095 FPGA. The USB 3.0 off-chip interface uses the Cypress FX3

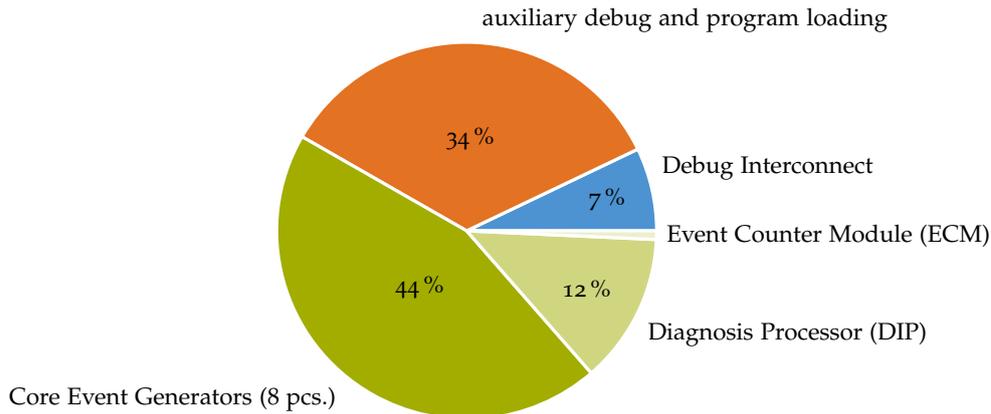


Figure 5.2: Distribution of total LUTs to the components of the Dia Engine. See Table 5.1 for detailed data values.

chip mounted on an adapter board. The host software runs on an Ubuntu Linux 16.04 system with an Intel Core i7-3770 CPU running at 3.4 GHz and 16 GB main memory.

All logic, with the exception of I/O logic, is clocked with a 50 MHz clock. This clock frequency is below the maximum achievable frequency, but was chosen as a safe target to simplify the implementation. Ultimately, the chip frequency is a scaling factor in the evaluation results, but does not (within reasonable bounds) affect the validity of the results.

With the given parameters the Debug Interconnect can carry up to $16 \text{ bit} \cdot 50 \text{ MHz} = 100 \text{ MB/s}$. Each Debug Packet consists of three header flits and between zero and nine payload flits. This leads to a net (payload) data rate of up to $9/12 \cdot 100 \text{ MB/s} = 75 \text{ MB/s}$.

In our case studies we measure the actual off-chip bandwidth, and use this data to discuss possible off-chip interface choices. We have therefore dimensioned the off-chip interface in our “chip design” to be not a bottleneck, as the following calculations show. The Host Interface Module encodes Debug Packets in a way that adds one word per packet to the data stream. With minimum-sized packets of three flits the overhead is $1/3$, resulting in a peak off-chip data rate of 133 MB/s . This is safely below 190 MB/s , the data rate which can be sustained by the USB 3.0 off-chip interface.¹

5.1.1 Evaluation System Resource Usage

The complete implementation of the observed system together with the Dia Engine allows for a closer look at the resource consumption of the evaluation chip design, and its individual components. Table 5.1 gives the resource usage numbers after the routing step of the implementation. All numbers are produced by Xilinx Vivado 2018.1 using the default synthesis and implementation settings.

¹https://www.glip.io/group__backend__cypressfx3-examples-vcu108__loopback.html

Instance	Total LUTs	Logic LUTs	RAM (kbit)	DSPs
observed system <small>(gray)</small>	72 211	69 923	4662	59
compute tile (1 of 4)	12 055	11 937	864	14
CPU core (1 of 2 per tile)	4154	4145	432	7
2 × 2 mesh NoC	4381	4381	0	0
DRAM infrastructure	19 590	17 774	1206	3
Dia Engine	27 047	26 303	1620	3
Debug Interconnect <small>(blue)</small>	1907	1570	0	0
observation and analysis <small>(green)</small>	15 905	15 516	1584	3
Core Event Generator (CEG) (1 of 8)	1493	1453	0	0
Diagnosis Processor (DIP)	3439	3418	1584	3
Event Counter (CNT)	198	198	0	0
auxiliary debug/trace <small>(orange)</small>	9235	9217	36	0
STM (program output) (1 of 8)	923	923	0	0
MAM (program loading) (1 of 4)	340	340	0	0

Table 5.1: Hierarchical resource utilization of our case study design after implementation (routing) with Vivado 2018.1 for a Xilinx XCVU095 FPGA. The “RAM” column shows the sum of all dedicated 18 and 36 kbit RAM blocks (RAMB18 and RAMB36). Additional memory may be instantiated as LUTRAM or shift registers; this memory is included in the “Total LUTs.” The colors refer to Figure 5.1. Only selected modules are shown. Due to cross-hierarchy optimization in the implementation process per-module numbers are not fully accurate.

The observed system (gray in Figure 5.1) consists of a total of 72k 6-input look-up tables (LUTs), 97 percent of which are used for logic. The remaining LUTs are used for memory, shift registers, and other special-purpose functionality. A rather constraint resource on any FPGA are RAM blocks (BRAM); the observed system makes use of a total of 4.5Mbit of such memory. Additionally, 59 DSP blocks are used.

The observed system contains four compute tiles, each with two CPU cores, and the mesh NoC. Each compute tile contains a distributed memory block, which is mapped to DRAM. To interface with this memory Xilinx provides memory interface IP, which hides the complexity of the DDR interface. Since this block contains, among other things, a dedicated soft core microprocessor, it occupies around 27 percent of the observed system (20k LUTs).

The complete Dia Engine requires 27k LUTs, the distribution to its subcomponents is visualized in Figure 5.2. 41 percent of the Dia Engine are used by the Debug Interconnect and the auxiliary modules (sending the program output to the host PC, program loading, and the off-chip interface). The remaining resources are used by the observation and analysis components of the Dia Engine, the main contribution of this work. The eight Core Event Generators dominate the resource usage of 16k LUTs and 1.5Mbit of block RAM. As discussed in Section 4.4.8 their size is mainly due to the amount of memory used to collect data and to buffer event packets in the case of bursts.

The Diagnosis Processor is 27 percent smaller than a regular processor since it does not contain a FPU, even though the DIP contains, in contrast to a normal core, two more components: while the main CPU cores use off-chip DRAM, the Diagnosis Processor includes 128 kB on-chip SRAM (block RAM). Additionally, the Diagnosis Processor includes a MAM (to load the Dia script onto it), and a dedicated network adapter to send and receive event packets.

The benefit of special-purpose execution units compared to the Diagnosis Processor is obvious by looking at the Event Counter (CNT): it only takes 0.2k LUTs, less than six percent of a Diagnosis Processor.

Overall, the observation and analysis part of the Dia Engine adds 23 % LUTs and 34 % block RAM to the size of the memory-less observed system. It must be noted, however, that this number is somewhat “unfair” towards the Dia Engine, as it does not equally account for the chip area occupied by memory: the Dia Engine components are self-contained and consist of significant amounts of on-chip SRAM, while the observed system makes use of off-chip DRAM.

5.1.2 Data Rates for Tracing Systems

Our case studies compare the off-chip data rate achieved by DiaSys with data rates from traditional tracing systems, like ARM CoreSight. However, we cannot directly compare measurement results, as the traditional tracing systems are closed systems which do not provide ways to obtain the necessary measurement data. We therefore compare our measurements with calculated data rates from traditional tracing systems, assuming typical compression ratios from literature for instruction and data traces.

For instruction traces, we assume a compression to 2 bit/instruction (c.f. Section 2.2.1). Hence, a full instruction trace from one of our CPUs running at 50 MHz with an IPC of 1 results in a data rate of 12.5 MB/s. Tracing all eight CPUs in the system would create an instruction trace of 100 MB/s. For data traces, we assume a compression to 16 bit/access, which includes the address and the 32 bit data value (c.f. [35]).

By combining these two numbers we can give an estimate of an instruction and data trace for an application executing on a single CPU. Assuming that every fifth instruction performs a memory access, the trace would consume 32.5 MB/s.

While these numbers should provide a fair comparison point, several things must be noted.

- None of the mentioned trace data rates include timestamps, or other information to correlate data traces with instruction traces.
- The compression ratios presented are averages. Depending on the executed software the actual rates can be higher or lower.
- Depending on the filtering and triggering capabilities of the tracing system a full (instruction or data) trace might not be needed for a given observation task.

5 *Case Studies*

The “chip design” we have presented in this section is now used in our case studies. In the first case study we debug and later test a functional problem, a race condition. The focus of this case study is on the usability and flexibility of DiaSys. The second and third case study focus on the data rates produced by runtime analysis scenarios. Case Study II looks at a function profile, and Case Study III creates a lock contention profile from a real-world application in the PARSEC benchmark suite.

5.2 Case Study I: Debugging and Testing of Race Conditions

Race conditions are among the most challenging defects in software. To locate and understand a bug caused by a race condition developers need to reason about multiple concurrent threads of execution, and understand possible interleavings between them. Race conditions are not only challenging for a human developer, but also for the diagnosis tool. The visibility of a race condition depends on the execution timing, which rules out the use of all intrusive diagnosis techniques. In such a scenario tracing systems are employed today.

In this first case study we show that DiaSys performs well in such a challenging debugging scenario. We further show how DiaSys can be used to evolve the manual debugging knowledge into a test which can be run repeatedly to ensure that the race condition does not reappear in future versions of the program code.

The case study presented in this section is an extended version of a case study previously published in [15].

5.2.1 Problem Description

A race condition occurs if the outcome of an operation depends not only on the executed instructions, but also on the timing between them. Sometimes race conditions are intended, but in many cases they are not and considered a bug.

For this case study we chose to use a textbook example of a race condition. Even though the example might seem “trivial” and not related to embedded systems, it helps to reduce our discussion to the essentials of a race condition and its debugging approach, without lengthy discussions of the example itself.

The example features three “actors,” a bank, which holds a single bank account, and two ATMs, which can be used to withdraw money from the bank account. The balance on the bank account may never become negative, and the ATMs check that by first obtaining the current balance from the bank, and proceeding with the withdrawal only if sufficient funds are available.

Expressed as an application running on an embedded system the example looks like this. The bank and the two ATMs are each a task, each running on one processor in different tiles. The tasks communicate with each other by exchanging messages over the NoC.

Each exchanged message has a source, a destination, a type, and a data value. The messages `get_balance_req` and `get_balance_resp` are the request and response messages to get the account balance from the bank. The message `modify_balance` modifies the account balance.

In our implementation both ATM tasks repeatedly get the balance, and withdraw money if the funds are sufficient for the withdrawal. The time between the withdrawal attempts and the amount of “withdrawn money” are random.

In a bug-free implementation, the account balance would never reach a negative value. However, exactly that is observed by the developer. He/she now uses DiaSys to locate the defect.

5 Case Studies

```
1 // int message_send(uint32_t dest_tile_id, message_type_t msg_type,
2 //                   uint32_t msg_data);
3 EVENT message_send TRIGGER
4   AT Cpu WHERE core_id = 0
5   WHEN pc = pc_from_elf("binary.elf", "message_send.call")
6   CAPTURE
7     uint32_t elf_funcarg("binary.elf", "message_send", 1) AS "dest_tile_id",
8     uint32_t elf_funcarg("binary.elf", "message_send", 2) AS "msg_type",
9     uint32_t elf_funcarg("binary.elf", "message_send", 3) AS "msg_data"
10
11 // void queue_message_received(uint32_t src_tile_id, message_type_t msg_type,
12 //                              uint32_t msg_data);
13 EVENT message_received TRIGGER
14   AT Cpu WHERE core_id = 0
15   WHEN pc = pc_from_elf("binary.elf", "queue_message_received.call")
16   CAPTURE
17     uint32_t elf_funcarg("binary.elf", "queue_message_received", 1) AS "src_tile_id",
18     uint32_t elf_funcarg("binary.elf", "queue_message_received", 2) AS "msg_type",
19     uint32_t elf_funcarg("binary.elf", "queue_message_received", 3) AS "msg_data"
20
21
22 __on_host__ log_msgs(in message_received, in message_send) {
23   dia_ev_t* ev = dia_ev_wait(message_received | message_send);
24   switch (ev->type) {
25     case message_send:
26       message_send_t* send_ev = (message_send_t*) ev;
27       printf("<- dest=%lu, msg_type=%lu, msg_data=%lu\n",
28             send_ev->dest_tile_id, send_ev->msg_type, send_ev->msg_data);
29       break;
30     case message_received:
31       message_received_t* rcv_ev = (message_received_t*) ev;
32       printf("-> src=%lu, msg_type=%lu, msg_data=%lu\n",
33             rcv_ev->src_tile_id, rcv_ev->msg_type, rcv_ev->msg_data);
34       break;
35   }
36 }
```

Listing 5.1: Dia script to create a log of messages received by and sent to the bank task. A graphical representation of the script's output is shown in Figure 5.3.

5.2.2 Debugging by Hand: Observe Exchanged Messages

Software diagnosis during the debugging process helps the developer to understand how the program behaves, and consequently, why it misbehaves. A common starting point in debugging is to add additional program output like log messages, and check that output manually for correctness.

This is also the first step in our debugging scenario. Under the assumption that the interaction between the ATMs and the bank is somehow faulty, the developer creates a log of all exchanged messages. To create this log, he/she instructs DiaSys to create an observation event whenever the bank tasks sends or receives a message. A single

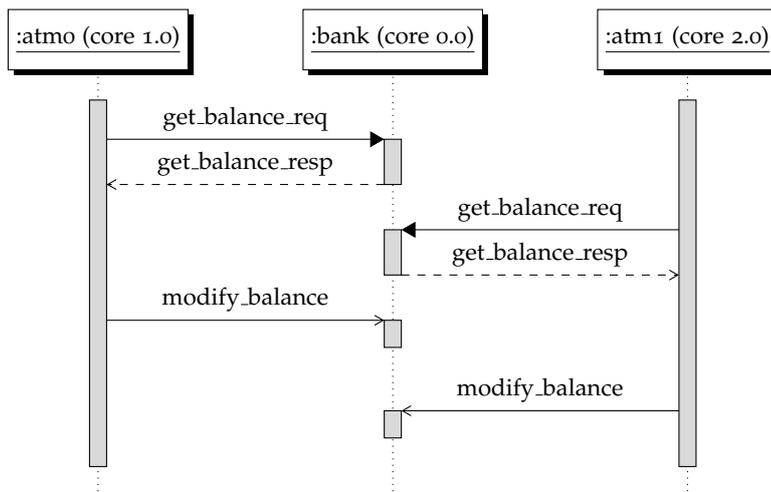


Figure 5.3: Sequence diagram showing the race condition discussed in Case Study I.

transformation actor in the Dia script receives all messages, and displays them in a human-readable form. Listing 5.1 shows the corresponding Dia script.

To make the text log produced by DiaSys even easier to understand we visualized it in the form of a message sequence chart shown in Figure 5.3. Looking at this diagram experienced developers will notice the bug: withdrawal transactions from two ATMs are interleaved. Even though each ATM individually checks the account balance before withdrawing money, the results of the check are invalidated by another ATM changing the balance before the modification happens.

After this analysis the developer has located the bug: the two operations `get_balance` and `modify_balance` are not executed as an atomic transaction. Following the debugging process outlined in Section 2.1.2 (p. 11) the developer can now proceed to correct the defect.

Fixing this bug would usually involve either introducing a lock, or using a compare-and-swap (CAS)-like behavior when modifying the balance variable. To do so the `modify_balance(int amount)` function would be extended to include an old value like `modify_balance(int old_balance, int amount)`. The old balance can then be used by the bank task to detect a race condition and inform the atm tasks to try again.

5.2.3 Discussion: Manual Debugging with DiaSys

The use of a software diagnosis tool for program comprehension is a very common task during the development of software. Due to its non-intrusive nature DiaSys can be used in scenarios where inserting “a `printf()`” into the program code is not an option—a feature it shares with traditional tracing systems. In the following we discuss the example in two aspects. We first compare the data rates of our approach with traditional tracing systems. And we then discuss the usability of DiaSys, especially regarding the Dia Language.

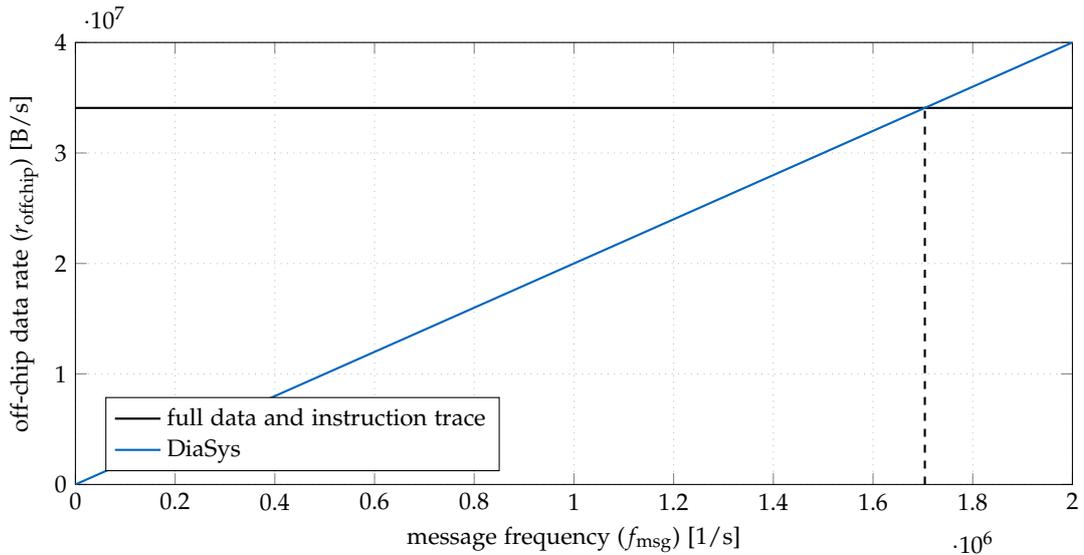


Figure 5.4: Comparison of data rates between a traditional tracing system and DiaSys when detecting race conditions using the manual approach as outlined in the first part of Case Study I. DiaSys requires less off-chip bandwidth than a traditional tracing system up to a message rate of 1.7 million messages per second.

When compared to a traditional tracing system, DiaSys has two advantages: the observation is very selective, and yet provides rich insight. To obtain similar insight from a traditional tracing system, an instruction and a data trace must be obtained. This requirement already excludes many of today's SoCs which offer an instruction trace, but no data trace. If a data trace is available and the off-chip interface is powerful enough to stream it to the host PC, much effort is needed to correlate the two traces on the host side to find the arguments passed to the send and receive functions. By only capturing required data, but including data in the observation events, DiaSys reduces the cost of the off-chip interface, and the processing on the host.

A quantitative analysis of the data rates supports this argument. Each observation event described in Listing 5.1 is 20 B large. The packet header is 6 B, the event ID occupies 2 B, and the captured data 12 B. One packet takes 10 cycles to transmit over our 16bit wide Debug Interconnect. The resulting data rate depends on the observed software, more specifically on the rate of sent and received messages at the bank task.

For comparison, obtaining a compressed instruction and data trace from a single CPU in our system would produce 32.5 MB/s of data (c.f. Section 5.1.2)—independent of the message rate. Hence, up to a rate of one (received or sent) message per 30 clock cycles, or more than 1.7 million messages per second, the data rate produced by DiaSys is below the one produced by typical tracing systems, as Figure 5.4 shows.

The second, necessarily subjective, topic is usability. The Dia script shown in Listing 5.1 is not very long, hence relatively fast to write for a developer. Roughly half of the code is used to specify the events and especially the captured function arguments.

The other half is the transformation actor which converts the received events into text form.

The example code shows that the Dia Language is sufficiently expressive for this task, but it also shows areas in which the language could be improved.

- In event definitions, the specification of function arguments could be made less verbose.
- The transformation actor code could be simplified by removing the need for type casts, complexity that is necessary due to our use of C.
- Finally, it could be considered to have a “default” actor for events which are not processed explicitly by an transformation actor. This actor could print out events in a standardized form, which is sufficient for many manual observation tasks.

In summary, DiaSys provides the necessary insight to help a developer understand the software execution in the presented race condition example. In realistic scenarios DiaSys provides this insight with less off-chip traffic than traditional tracing systems. The Dia Language provides sufficient expressiveness, but could be improved to reduce the amount of necessary “boilerplate” code.

After this discussion we return to the race condition example. Now that the developer has located the defect and fixed it, he or she wants to ensure that it does not happen again. To do this, he/she writes automates the diagnosis in the form of a test.

5.2.4 From Debugging to Testing: A Transaction Checking Test

Once the developer understood the race condition, he/she can use DiaSys to check if all messages are exchanged according to a race-free protocol. In the correct scenario the sequence of getting the balance, comparing it, and modifying it is an atomic transaction. We use this correct scenario as hypothesis and use DiaSys to check it. If the hypothesis does not hold, we have found a race condition.

A Dia script which performs this automated checking is presented in Listing 5.2. It makes use of the same observation events as before (c.f. Listing 5.1). A transformation actor `check_transaction` checks the atomicity of the transaction. If violations are found, a new event `race_found` is triggered, which is ultimately reported back to the host.

5.2.5 Discussion: Testing

The most significant benefit from automating the hypothesis test, as shown in the previous section, is the increase in productivity for the developer. No manual inspection of the event log is necessary, which is beneficial especially if race conditions happen only rarely (as they usually do). Additionally, the test can be executed unattended, for example as part of a continuous integration system, which tests the software repeatedly and reports failures back to the developer.

Furthermore, the additional processing within the `check_transaction` reduces the required off-chip bandwidth: only if a race condition is found, an event is sent off-chip; otherwise, all processing can happen on-chip.

In our evaluation chip design the `check_transaction` actor is mapped to the on-chip Diagnosis Processor (DIP). In the longest code path we measure an execution time of the transformation actor of 2498 cycles after the `dia_ev_wait()` function returns. (The longest code path includes the sending of a `race_found` event.) The reception of an incoming `message_received` packet is performed within an interrupt service routine taking 3000 cycles (c.f. Section 4.4.10), which can be called at any time during the program execution. Since the transformation actor is triggered once per incoming event, we get a total processing time for one incoming event packet of 5498 cycles. Therefore, the Diagnosis Processor running at 50 MHz can execute the `check_transaction` actor $9090\times$ per second. We can conclude that as long as the bank task receives less than 9090 requests per second the atomicity of the `get_balance / modify_balance` transaction can be checked fully on-chip. Only atomicity violations, i.e. race conditions, are reported to the developer on the host.

5.2.6 Summary: Debugging and Testing of Race Conditions with DiaSys

Race conditions are among the most challenging defects faced by software developers, as they rarely occur, and may or may not be visible depending on the execution timing. In this case study we have shown the DiaSys is a suitable debugging and testing tool in such a scenario.

In the first part of the case study, we have used DiaSys for manual debugging of the race condition. A Dia script printed out a log of messages, which allowed the developer to manually compare his/her expectations how the software should execute with the actual execution. Thanks to the very selective data collection mechanisms of DiaSys the off-chip data rate produced by DiaSys is significantly lower than in a traditional tracing system.

In a second step we evolved the Dia script to perform an automated checking of the software execution, i.e. we built knowledge how the software should execute into a test. This test can be run on-chip in the Diagnosis Processor, and the off-chip traffic is limited to the reporting of violations. In addition to saving the developer time, since he/she does not need to manually check the message log anymore, the observation can be run for a long time and/or unattended. As such the test can be executed in a continuous integration environment, ensuring that the race condition, after it has been fixed, does not reappear in future versions of the code.

5.2 Case Study I: Debugging and Testing of Race Conditions

```
1 EVENT message_received TRIGGER
2   AT Cpu WHERE core_id = 0
3   WHEN pc = pc_from_elf("binary.elf", "queue_message_received.call")
4   CAPTURE
5     uint32_t elf_funcarg("binary.elf", "queue_message_received", 1) AS "src_tile_id",
6     uint32_t elf_funcarg("binary.elf", "queue_message_received", 2) AS "msg_type"
7
8 EVENT race_found CONTAINS uint32_t id1, uint32_t id2
9
10
11 check_transaction(in message_received, out race_found) {
12   static bool in_transaction = false;
13   static uint32_t transaction_owner;
14
15   message_received_t* ev = dia_ev_wait(message_received);
16
17   // copied from the application source code for symbolic names of msg_type
18   typedef enum {
19     GET_BALANCE_REQ,
20     GET_BALANCE_RESP,
21     MODIFY_BALANCE
22   } application_message_types;
23
24   // only consider these two messages
25   if (ev->msg_type != GET_BALANCE_REQ && ev->msg_type != MODIFY_BALANCE) {
26     return;
27   }
28
29   // check for race condition
30   if (in_transaction && ev->src_tile_id != transaction_owner) {
31     race_found_t* out_ev = dia_ev_new(race_found);
32     out_ev->id1 = transaction_owner;
33     out_ev->id2 = ev->src_tile_id;
34     dia_ev_send(out_ev);
35   }
36
37   // mark transaction start
38   if (ev->msg_type == GET_BALANCE_REQ) {
39     transaction_owner = ev->src_tile_id;
40     in_transaction = true;
41   }
42   // mark transaction end
43   if (ev->msg_type == MODIFY_BALANCE) {
44     in_transaction = false;
45   }
46 }
47
48 __on_host__ report_results(in race_found) {
49   race_found_t* ev = dia_ev_wait(race_found);
50   printf("Found a violation of the transaction protocol. "
51         "Tile %lu interleaved with tile %lu.\n", ev->id1, ev->id2);
52 }
```

Listing 5.2: The Dia script to automate the checking of the transaction.

5.3 Case Study II: Function Profiling

Profiles, especially function profiles, are a commonly used analysis technique when evaluating the performance of an application. While many developers have a good understanding of the complexity of the source code they are currently working on, it is often less obvious which parts of a program account for the most execution time. A profile provides this information at function granularity in the form of a sorted list, containing the number of calls to the function, the average execution time, and the total execution time (possibly in relation to the calling function, or the whole program). In this case study we present two approaches to create a function profile with DiaSys: one exact approach, and one approach which uses sampling.

5.3.1 Creating a Function Profile with Exact Accounting

Our first profiling approach is “exact” as it counts every function call. Listing 5.3 shows the used Dia script. We configure the Core Event Generator to trigger the creation of a new event whenever a function is called. Calls to a function are identified by their opcode; for the or1k ISA the opcodes `j.jal` and `j.jalr` are used.² As payload we capture the timestamp (`timestamp`), and the program counter of the jump target (`npc`), which is equal to the entry point of the called function.

Two other events are defined to transfer the resulting profile from the transformation actor to the host: the `request_profile` event is sent from the host to the `create_profile` transformation actor to trigger the sending of the profile. The response is sent as a series of `profile_entry` events, each of which contains a single entry (i.e. a single function) in the profile.

In the following we discuss two scenarios: in the first one, the `create_profile` actor is executed off-chip, while in the second scenario it is executed on-chip.

5.3.1.1 Off-chip Profile Generation with Exact Accounting

If the `create_profile` transformation actor is executed off-chip, all `func_call` events must be sent through the off-chip interface to the host PC. Each `func_call` event is 16 B large, hence the resulting off-chip data rate r_{offchip} can be calculated as a function of the function call rate f_{call} .

$$r_{\text{offchip}} = 16 \text{ B} \cdot f_{\text{call}}$$

Figure 5.6 shows this relation (blue). Up to $f_{\text{call}} = 819 \cdot 10^3$ function calls per second, the exact accounting approach of DiaSys is more bandwidth-efficient than traditional tracing, even without any on-chip processing. With on-chip processing even more off-chip bandwidth can be saved, as shown next.

²The `opcode_num()` function used in Listing 5.3 returns the opcode number for the given symbolic name according to the ISA manual. For example, the `l.jal` symbolic name is resolved to the value `0x01`.

5.3.1.2 On-chip Profile Generation with Exact Accounting

The key limiting factor for on-chip profile generation is the available processing power provided by the Diagnosis Processor. We discuss this first, before moving on to discuss the achievable off-chip bandwidth savings.

Listing 5.3 shows the processing performed in the `create_profile` actor to create a profile. At its core is the `stat_account()` function, which is part of the runtime system provided for the DIP. This function manages a data structure which records the total execution time and the number of calls for each observed function call. It is implemented as an open addressing (fixed-size) hash table with linear probing. The implementation requires a static amount of memory (defined at compile time). Looking up keys, or inserting new data, takes constant time if no hash conflict is present.

The program binary loaded onto the DIP has a static size of 53 kB, leaving up to 75 kB for runtime-allocated data. The largest data structure is the statistics hash table, which requires 12 B per bucket (entry in the hash table). An optimal size of the hash table depends on the number of entries, i.e. the number of unique functions observed during the program execution. In our implementation we have dimensioned the hash table to contain 1024 entries, which is large enough for applications up to roughly 700 unique functions.³ Hence, our statistics hash table adds 12 kB of runtime data, leaving enough headroom for other dynamically allocated data, such as incoming packets and the stack.

To process a single `func_call` event the Diagnosis Processor needs 3550 cycles, which consists of 3000 cycles to receive the event from the network interface, and 550 cycles to update the statistics hash table.⁴

The processing time can also be viewed from another angle: the DIP is able to create a function profile if functions in the observed software take 3550 cycles or more on average. If multiple CPUs are observed the number needs to be multiplied with the number of CPUs.

It therefore depends on the observed software if the processing performance of the DIP is sufficient to create a profile. More specifically, up to $f_{\text{call}} = 14.1 \cdot 10^3$ function calls per second (by any CPU), the on-chip profile generation is possible. In this case, shown as dashed line in Figure 5.6, the off-chip data rate is reduced from 12.5 MB/s for a traditional tracing system to 200 B/s, assuming that every second a profile containing 10 entries is sent from the chip to the host PC.

Sampling-based analysis can avoid the dependency on f_{call} , as shown next.

5.3.2 Creating a Function Profile with Sampling

Instead of accounting for every function call in the observed software, sampling can be used to create a profile. The main benefit of sampling is its controllability: whereas

³A hash table as we implemented it performs well up to a maximum load factor of roughly 70%. The load factor is defined as the number of entries over the number of buckets. If this factor is exceeded, the number of hash collisions increases and hash table lookups do not have near-linear performance anymore.

⁴All runtime numbers are given after the initialization has been completed.

5 Case Studies

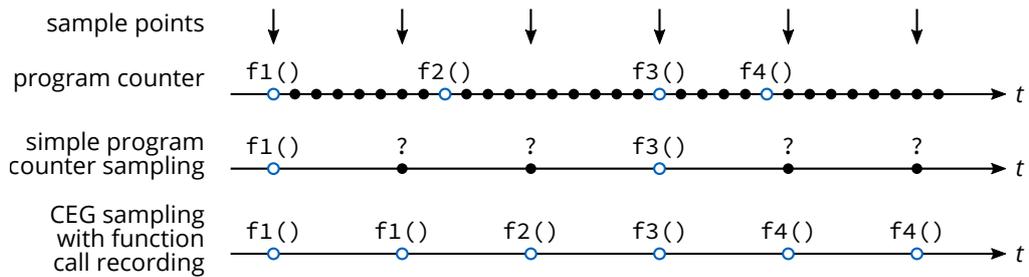


Figure 5.5: Program counter sampling using simple sampling, or the function call sampling feature of the CEG. A black dot represents a program counter value, a blue circle is a function call.

in the exact accounting approach the rate of observation events is a function of the observed software, with sampling it is a design-time constant, the sample rate. For profiling typical sample rates are around 1 kHz and 10 kHz.⁵

To create a function profile using sampling we use two special features of the Core Event Generator.

1. We use the time trigger to issue a new observation event after a configurable period.
2. The CEG continuously remembers the address of the currently executing function. This data value can be captured when the trigger fires and included in the observation event.

Commercially available tracing systems do not provide comparable functionality. Therefore, they need to capture a full instruction trace, sample the program counter, and then use the program binary to map arbitrary program counters to function addresses, as Figure 5.5 illustrates.

The aggregation of observation events into a profile is performed as in Listing 5.3; the only difference is that events are now arriving at a predefined, regular interval. As in the exact accounting case, we can execute the `create_profile` transformation actor on-chip or off-chip.

5.3.2.1 Off-chip Profile Generation with Sampling

If the sampled data is aggregated into a profile on the host PC, all sample events need to be sent off-chip. For a 1 kHz sampling rate the data rate is reduced to 15.6 kB/s, or a factor of 819 compared to a traditional tracing system; for a 10 kHz sampling rate the data rate is reduced by 82×. The absolute data rates are given for a single observed CPU; if a profile from multiple CPUs is to be generated, the required off-chip data rate scales linearly with the number of CPUs. For example, a sampling-based

⁵Typically, rates are used which do not coincide with periods of time-triggered systems, e.g. 999 Hz or 9999 Hz.

profile obtained at a 10 kHz sampling frequency from our eight-core system generates 1.2 MB/s off-chip traffic.

Notably, this off-chip traffic is independent of the frequency of the observed CPUs: a high-performance eight-core server processor generates the same amount of data as our evaluation system. A data rate of 1.2 MB/s can be handled even by a comparably cheap Universal Asynchronous Receiver Transmitter (UART) interface.

5.3.2.2 On-chip Profile Generation with Sampling

With a 10 kHz sampling rate the processing of a single event within the DIP may take up to 5000 cycles, with a 1 kHz sampling rate even up to 50 000 cycles. Since the actual processing only takes 3550 cycles in our current implementation we have room to extend the analysis, e.g. with the calculation of a standard deviation.

As in the exact accounting case, on-chip processing reduces the off-chip data rate from 12.5 MB/s for a traditional tracing system to 200 B/s, assuming that every second a profile containing 10 entries is sent from the chip to the host PC.

In summary, sampling-based profiling is a good way to gain back control over the analysis process, and reduce the off-chip data rate along the way. It allows the developer to design a Dia script which is guaranteed to be executable, since the answer to this question does not depend on the observed software, but only on design parameters. The downside of sampling is a loss in accuracy, especially for functions which have a short runtime and are rather infrequent.

5.3.3 Summary: DiaSys for Function Profiling

Function profiling is one of the most common tasks to assess the performance of an application, and to find a starting point for further investigations. To obtain a good profile, the software execution must be observed for a longer period. In traditional tracing systems, the creation of a profile requires a full instruction trace which is streamed to the host PC [101].

With DiaSys a function profile can be created in two ways: either with exact accounting, or with sampling. The exact method has an important downside: the rate of observation events, and hence the needed processing power to create a profile, depends on the observed software. Given that developers create a profile to understand the software this creates “chicken or egg”-style problem.

Creating a profile using sampling leads a way out of this problem. Since the sampling rate is a developer-controlled parameter he/she can determine a sustainable amount of processing at design time.

In this case study we have shown that the Diagnosis Processor is sufficiently dimensioned to create a sampling-based function profile at 10 kHz sampling rate. Notably this dimensioning is not only sufficient for our evaluation system, but also for any system running at much higher frequencies, e.g. a desktop processor running at 3 GHz. A necessary prerequisite, however, is an event generator as the one present in DiaSys which is able to sample function addresses, as opposed to only program counters.

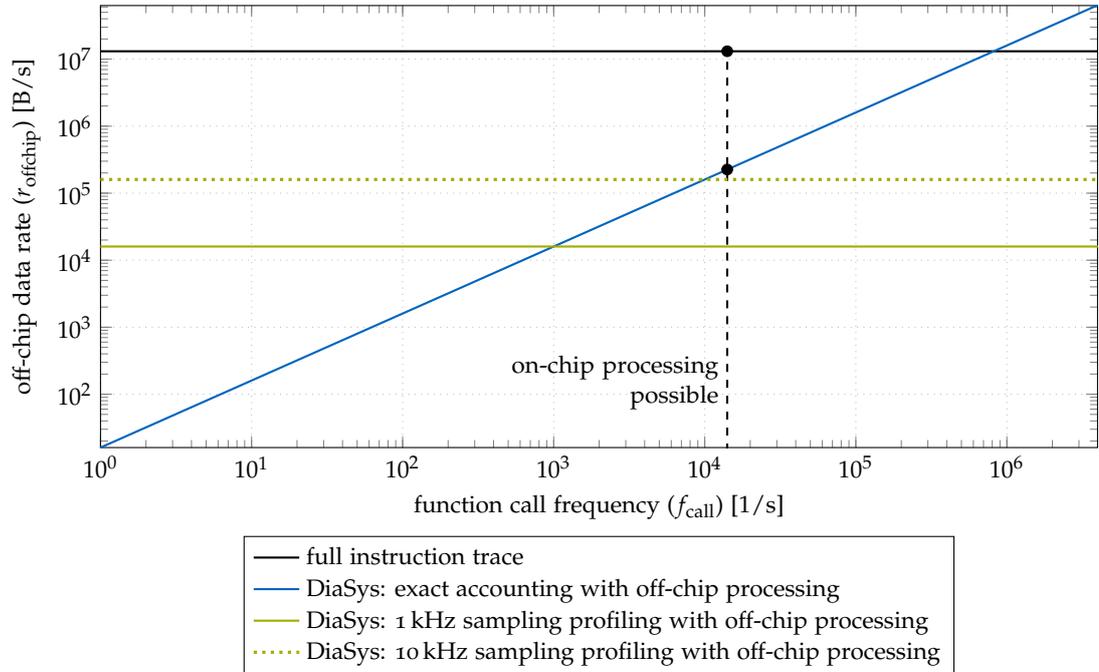


Figure 5.6: Comparison of data rates between a traditional tracing system and DiaSys when creating a function profile. The DiaSys data rates assume no on-chip processing, all processing is performed off-chip.

In contrast to the exact accounting approach (blue), sampling produces constant off-chip traffic, only depending on the sampling rate (green).

Up $14.1 \cdot 10^3$ function calls per second an exact profile can be generated on-chip by the Diagnosis Processor. This boundary is shown as dashed line. The dots denote the maximum data rate savings achievable with on-chip processing, compared to a traditional tracing system, and off-chip processing in DiaSys.

With on-chip processing only the aggregated profile needs to be sent off-chip when the user requests it, the data rate is hence close to zero (not shown).

```

1 EVENT func_call TRIGGER
2   AT Cpu WHERE core_id = 0
3   WHEN opcode = opcode_num("1.jalr") OR opcode = opcode_num("1.jal")
4   CAPTURE uint32_t timestamp, uint32_t npc
5
6 EVENT request_profile
7
8 EVENT profile_entry
9   CONTAINS uint32_t func_pc, uint32_t cnt, uint32_t time
10
11 create_profile(in func_call, in request_profile, out profile_entry) {
12   static uint32_t timestamp_prev = 0;
13   static uint32_t npc_prev = 0;
14   static struct stat_ctx *stat_ctx = NULL;
15
16   dia_ev_t* ev = dia_ev_wait(func_call | request_profile);
17
18   stat_init(&stat_ctx);
19
20   if (ev->type == func_call) {
21     func_call_t *cnt_ev = (func_call_t*)ev;
22
23     if (timestamp_prev != 0) {
24       uint32_t tdiff = cnt_ev->timestamp - timestamp_prev;
25       stat_account(stat_ctx, npc_prev, tdiff);
26     }
27     timestamp_prev = cnt_ev->timestamp;
28     npc_prev = cnt_ev->npc;
29
30   } else if (ev->type == request_profile) {
31     profile_entry_t* out_ev;
32     struct ht_entry *item = stat_first(stat_ctx);
33     while (item != NULL) {
34       out_ev = dia_ev_new(profile_entry);
35       out_ev->func_pc = item->key;
36       out_ev->cnt = item->cnt;
37       out_ev->time = item->sum;
38       dia_ev_send(out_ev);
39
40       item = stat_next(stat_ctx);
41     }
42
43     // send one last event to signal the receiver the end of the transmission
44     // [omitted for space reasons]
45   }
46 }
47
48 __on_host__ show_profile(in profile_entry, out request_profile) {
49   /* Implementation omitted for brevity.
50    1. Send request_profile event
51    2. Wait for all profile_entry events and sort them
52    3. Print data out as table.
53    4. Wait for e.g. 10 seconds, and repeat */
54 }

```

Listing 5.3: The Dia script used in Section 5.3.1 to create a function profile.

5.4 Case Study III: Lock Contention Profiling of a Standard Benchmark Application

In our third case study we generate a lock contention profile. To give meaningful insight into data rates generated when analyzing large real-life applications, we use in this example not a self-created example application, but an application from the PARSEC benchmark suite. We also show how the hierarchical data reduction within a diagnosis application offers trade-offs between adding on-chip processing power and off-chip bandwidth.

The case study presented in this section is an extended version of a case study previously published in [15].

5.4.1 Evaluation Prototype

The hardware implementation prototype of DiaSys presented in Section 4.4.7 is only able to run baremetal applications, i.e. applications which do not require an operating system. To run larger applications, such as standard benchmarks, we therefore created a software prototype of DiaSys. It runs purely in software on a Linux PC and is best suited for an evaluation of event rates inside the diagnosis system. Since no hardware extensions are used, its operation is intrusive, i.e. the timing of the observed application is slightly changed. The prototypical event generators can only trigger on the call and return from a C library function, and the function arguments can be included in the event as data items.

The software prototype consists of two parts, which are shown in Figure 5.7. The first part is a “preload library.” It is a small software library written in C which is able to monitor all calls to C library functions and write them into an event log file. This event log file is then used by a prototype of the diagnosis system implemented in Python. It consists of event generators, which read the event log file. A set of Python functions connected by channel objects represent the transformation actors. (We assume

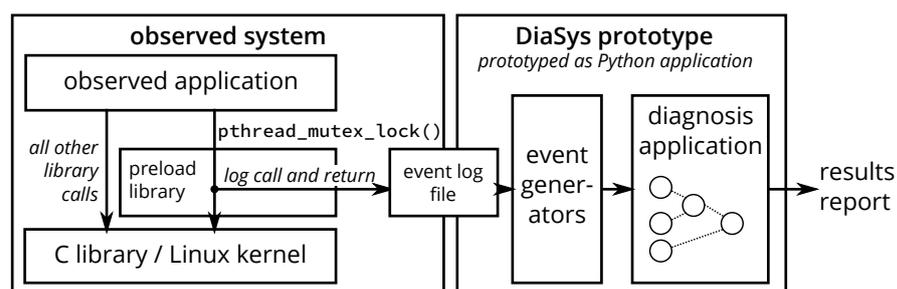


Figure 5.7: The software prototype of the diagnosis system. All calls in the observed application are recorded in an event log file by an preloaded library. The event log file is read by the diagnosis system implemented as Python application. The figure shows the monitoring of all `pthread_mutex_lock()` calls and returns.

5.4 Case Study III: Lock Contention Profiling of a Standard Benchmark Application

```
1 int pthread_mutex_lock(pthread_mutex_t *mutex) {  
2     blocking_wait_until_mutex_is_free(mutex);  
3     lock_mutex(mutex);  
4     return 0 /* success */;  
5 }
```

Listing 5.4: A sketch of the `pthread_mutex_lock()` function. This function must be executed atomically, i.e. without interruption.

a one-to-one mapping of transformation actors to processing nodes in this prototype.) The output of the Dia script is directly printed to a console.

We now use this software prototype to create the lock contention profile.

5.4.2 Problem Description

A *lock contention* occurs in concurrent programs if multiple threads try to acquire a mutex lock at the same time [102, p. 147]. In this case, all but one threads have to wait for the lock to be released before they can continue processing. Therefore, the lock acquisition time is a good metric for program efficiency: the less time it takes, the earlier the thread is done with its work.

In order for a developer to get insight into the lock contention behavior of the program, a contention profile can be created. It lists all acquired locks, together with the summarized and averaged times the acquisition took. Such a profile can be generated in an intrusive way with tools like Intel VTune Amplifier or mutrace⁶, and is traditionally formatted as shown in Listing 5.7.

5.4.3 Measurement Approach

The lock acquisition time can be measured by obtaining the time the mutex lock function took to execute. In applications using pthreads, as it is the case for almost all applications running on Linux, macOS or BSD, the mutex lock function is named `pthread_mutex_lock()`.

As shown in the simplified code sketch in Listing 5.4, the function blocks for an indefinite amount of time until a lock is available. If it is available, it acquires the lock and returns.

To create a lock contention profile, we need to measure the execution times of all `pthread_mutex_lock()` function calls in all threads. We then group this measurement by lock, given by the argument `mutex` of the lock function, to obtain the number of times a lock was acquired, how long all lock acquisitions took in summary, and on average.

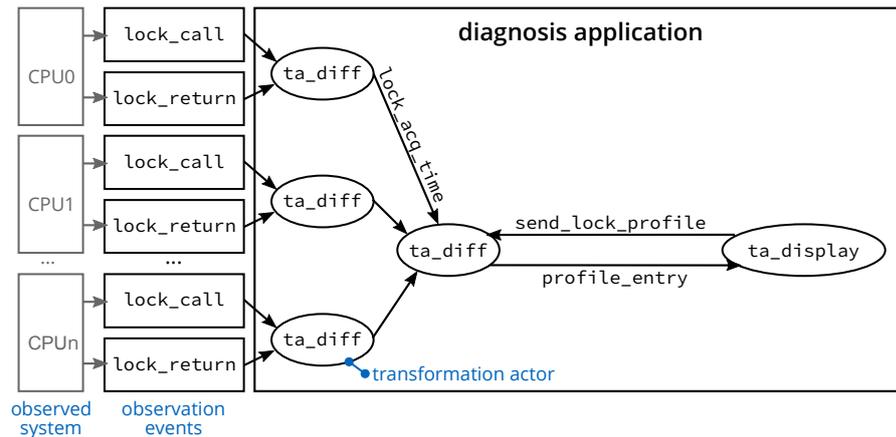


Figure 5.8: A graphical representation of the Dia script to create a lock contention profile.

5.4.4 The Dia Script

To perform the analysis outlined in the previous section, we configure DiaSys as shown in Figure 5.8. First, we configure the event generator on each CPU to generate two observation events which together measure the execution time of the lock acquisition function `pthread_mutex_lock()`.

- One observation event `lock_call` is triggered if the CPU *enters* (calls) the `pthread_mutex_lock` function. The first function argument to `pthread_mutex_lock`, the mutex, is attached to the event as data item, together with a timestamp containing the current time.
- Another observation event `lock_return` is triggered if the CPU *returns* from the `pthread_mutex_lock()` function. For this event, only a timestamp is attached as event data.

To calculate the execution time of the function `pthread_mutex_lock()`, we create a transformation actor `ta_diff` as shown in Listing 5.5.

It waits for both observation events `lock_call` and `lock_return`, calculates the difference between the timestamps, and creates a new event `lock_acq_time` with two data items, the lock acquisition time and a hash of the mutex argument to reduce the data size.

As last step in the processing, all `lock_acq_time` events are aggregated by another transformation actor called `ta_stat`. Again, a pseudo code implementation is given in Listing 5.6.

If an event of type `lock_acq_time` is received, the timestamp is added to a hash data structure which records, grouped by the mutex, the number of calls to the lock function and the total time these calls took. Whenever the (intermediate) results of the profile are wanted, a `send_lock_profile` event is created, which triggers the sending

⁶<http://0pointer.de/blog/projects/mutrace.html>

5.4 Case Study III: Lock Contention Profiling of a Standard Benchmark Application

```
1 ta_diff(in lock_call, in lock_return, out lock_acq_time) {
2   lock_call_t *call = dia_ev_wait(lock_call);
3   lock_return_t *ret = dia_ev_wait(lock_return);
4
5   uint16_t time = ret->ts - call->ts;
6   uint16_t mutex_hash = hash(call->mutex);
7
8   lock_acq_time_t *lock_acq_time = dia_ev_new(lock_acq_time);
9   lock_acq_time->lock = mutex_hash;
10  lock_acq_time->time = time;
11  dia_ev_send(lock_acq_time);
12 }
```

Listing 5.5: The transformation actor calculating the lock acquisition time, written in the Dia Language. In our prototype we implement equivalent functionality in Python.

```
1 ta_stat(in lock_acq_time, in send_lock_profile, out profile_entry) {
2   static struct stat_ctx *stat_ctx = NULL;
3
4   dia_ev_t *ev = dia_ev_wait(lock_acq_time | send_lock_profile);
5
6   stat_init(&stat_ctx);
7
8   // aggregate
9   if (ev->type == lock_acq_time) {
10    lock_acq_time_t acq_time = (lock_acq_time_t*) ev;
11    stat_account(stat_ctx, acq_time->mutex, acq_time->time);
12
13    // send statistics output to host PC
14  } else if (ev->type == send_lock_profile) {
15    profile_entry_t* out_ev;
16    struct ht_entry *item = stat_first(stat_ctx);
17    while (item != NULL) {
18      out_ev = dia_ev_new(profile_entry);
19      out_ev->mutex = item->key;
20      out_ev->cnt = item->cnt;
21      out_ev->time = item->sum;
22      dia_ev_send(out_ev);
23
24      item = stat_next(stat_ctx);
25    }
26
27    // send one last event to signal the receiver the end of the transmission
28    // [omitted for space reasons]
29  }
30 }
```

Listing 5.6: The transformation actor creating the lock profile in the Dia Language.

5 Case Studies

	mutex	# acq.	sum [ns]	avg [ns]
1				
2	(01) 0x7fd9ac018988	47785	8835387	184.90
3	(02) 0x7fd9d1ed2978	47784	226012031	4729.87
4	(03) 0x1c36500	9426	53724035	5699.56
5	(04) 0x1c36660	9423	21904608	2324.59
6	(05) 0x1c36710	4638	12528702	2701.32
7	(06) 0x1c365b0	105	46999	447.61
8	(07) 0x7fd9d2091430	8	1974	246.75
9	(08) 0x7fd9b41948f8	8	2277	284.62
10	(09) 0x7fd9b42b9ad8	8	2560	320.00
11	(10) 0x7fd9d20f8928	8	2215	276.88

Listing 5.7: Output of the lock contention profile diagnosis application observing the PARSEC dedup application.

of the statistics. A transformation actor running on the host can then display the lock contention profile in text form.

5.4.5 Evaluation

In the evaluation of this case study we focus on the event and data rates between the event generators and transformation nodes. In order to provide realistic inputs, we profiled the *dedup* application from the PARSEC 3.0 Benchmark Suite with the large input data sets [103]. As PARSEC does not run on our custom-built prototype MPSoC platform, we used the software prototype described in Section 5.4.1. All transformation actors were implemented in Python code equivalent to the Dia scripts in Listings 5.5 and 5.6.

5.4.5.1 Output of the Diagnosis Application

Before we analyze the diagnosis application itself, we discuss the output it generates, i.e. the lock contention profile shown in Listing 5.7. PARSEC was instructed to use at least four threads; ultimately 16 threads were spawned by the dedup application. (There is no option in PARSEC to specify the exact number of threads used.) The execution of the observed application took 2.68 s.

The output shows the top ten most acquired mutexes, together with the total and averaged lock acquisition time. Notable in this profile are mutexes 2 to 5, which take on average significantly longer to acquire: these locks are called to be “contended.”

A profile helps to understand the program behavior and serves as a starting point to fix possible bugs or inefficiencies. If lock contention is observed (and performance goals of the application are not met), it is common to replace coarse-grained locks with more fine-grained locks, i.e. locks which protect a shorter critical section. However, fixing a bug is not in the scope of this work. Instead, we now turn our discussion to the event and data rates generated when executing the Dia script that generated the profile as shown.

5.4.5.2 Event and Data Rates

We designed DiaSys to reduce the off-chip traffic by moving the data analysis partially into the SoC. To evaluate if the data rates are in fact reduced, we analyze event rates between the transformation actors.

We use the following event sizes:

- A `lock_call` event requires 20 B: 6 B for the packet header, 2 B event type identifier, 4 B, and 8 B for the `mutex` argument.
- A `lock_return` event requires 12 B.
- A `lock_acq_time` event requires 12 B: 6 B for the packet header, 2 B for the event type identifier, 2 B for the lock acquisition time, and 2 B for the hashed `mutex` argument.

Over the whole program run, the event generators attached to the 16 CPUs generate a total of 258 127 `lock_call` and an equal number of `lock_return` observation events, which equals 7.9 MB of data or, over the program runtime, an average data rate of 2.94 MB/s. The `ta_diff` transformation actors reduce the number of events by half, resulting in a data rate of 1.1 MB/s, or a reduction to 37%. Finally, after being aggregated by `ta_stat`, the full result can be transferred off-chip with 204 B.

A traditional tracing system produces much higher data rates, since an instruction trace and a data trace (for the `mutex` function argument) are needed. Since PARSEC runs on Linux on an Intel processor, we only have access to an instruction trace through Intel PT to perform a real-world comparison. However, as a first lower-bound estimation of the data rate generated by a state-of-the-art tracing system, we created a full instruction trace using Intel PT. The same PARSEC dedup application created a trace file of 1.82 GB, which corresponds to 679 MB/s over the program run-time.

In summary, DiaSys is able to reduce the required trace bandwidth compared to an Intel PT instruction trace significantly due to on-chip analysis. When transferring data off-chip after processing in the `ta_diff` processing nodes, the bandwidth is reduced from more than 679 MB/s to 1.1 MB/s, a reduction by 617 \times . After the `ta_stat` transformation actor only the profile itself needs to be transferred as events, which results in not more than a couple of bytes, which need to be transferred whenever an updated profile is desired, e.g. every second.

5.4.5.3 Discussion

Depending on the feature set and timestamp granularity of the various tracing implementations, the bandwidth reduction that DiaSys is able to achieve can vary. However, a general observation holds: the most significant bandwidth savings result from the fact that we very precisely capture only data in the event generators which is relevant to our problem. The subsequent processing step `ta_diff` of calculating the time difference between two events is further able to discard roughly $\frac{2}{3}$ of the data. Due to its

5 *Case Studies*

simplicity this step can be easily performed on-chip. The final step `ta_stat` is again able to give large percentage-wise reductions in data rate, however the absolute savings might not justify an on-chip processing any more. This last step could therefore be executed on the host PC, without changing the Dia script.

5.5 Summary: Case Studies

DiaSys is designed to be a general-purpose observation system for today's and tomorrow's embedded systems. As with any hardware-based system, the chip area needed by the system is an important factor, as it determines the production cost of the system. In Section 5.1.1, we present the implementation results for a eight-core system, which is extended with a fully featured observation system. In this scenario the observation and analysis components amount to 23% of the observed system, a high cost, when seen in relative numbers. This mismatch is mostly caused by the small size of the observed CPU cores, which follow a very simple design pattern and use off-chip DRAM. When seen in absolute numbers, the observation system is smaller than the DRAM infrastructure components.

To analyze the performance of DiaSys we have taken this implementation and ran two case studies on it. The first case study is from the field of functional debugging and testing: DiaSys is used to debug and test for a race condition bug. Our case study showed that this analysis can be performed fully on-chip within the Diagnosis Processor, causing no off-chip traffic except for the transfer of the final results to the developer. The case study also showed how Dia scripts can be evolved from manual debugging helpers to fully automated tests, reducing the learning curve for developer using DiaSys.

The second case study evaluated different options to create a function profile. It concluded that especially with the use of sampling a profile can be created fully on-chip, enabled by the advanced functionality within the Core Event Generator, and the on-chip Diagnosis Processor. Notably, by adding the DiaSys components in the same dimensioning and with the same hardware cost as in our evaluation system a function profile can be created for arbitrary systems, including systems with high-performance CPUs running at multiple GHz.

In the first two case studies we focused on the limits of DiaSys by giving bounds for values which depend on the observed software. In the third case study we used an application from the PARSEC benchmark suite for a closer look into event rates created by a real-world application. The scenario in this case is the creation of a lock contention profile, a task which typically requires a full instruction and a data trace in traditional tracing systems, and hence produces high off-chip data rates.

Overall, we have shown DiaSys to be applicable in many scenarios in which tracing systems are employed today. We have shown that on-chip processing can significantly reduce the off-chip data rate, or conversely, provide more insight into the software execution at the same data rate. Especially the function profiling scenario showed how little additional hardware is needed to create a profile fully on-chip, a task for which traditional tracing systems transfer gigabytes of data between the chip and the host PC.

Table 5.2 gives an overview over all case studies and the achieved reductions in off-chip traffic, which is also given in graphical form in Figure 5.9. The exact reductions in the off-chip traffic depend strongly on the case study and other assumptions. However, it is safe to say that reductions in the order of multiple magnitudes are possible.

	off-chip traffic		
	tracing [B/s]	DiaSys [B/s]	reduction
Case Study I: Debugging a race condition (Section 5.2)			
1.1 message log (manual debugging)	$34 \cdot 10^6$	$20 \cdot 10^3$	$2 \cdot 10^2$
1.2 transaction checking (automated testing)	$34 \cdot 10^6$	$16 \cdot 10^2$	$2 \cdot 10^4$
Case Study II: Function profiling (Section 5.3)			
2.1 accurate function profiling (off-chip processing)	$13 \cdot 10^6$	$225 \cdot 10^3$	$6 \cdot 10^1$
2.2 sampling with 1 kHz (off-chip processing)	$13 \cdot 10^6$	$16 \cdot 10^3$	$8 \cdot 10^2$
2.3 sampling with 10 kHz (off-chip processing)	$13 \cdot 10^6$	$160 \cdot 10^3$	$8 \cdot 10^1$
2.4 full on-chip processing	$13 \cdot 10^6$	200	$6 \cdot 10^4$
Case Study III: Lock contention profiling (Section 5.4)			
3.1 only <code>ta_diff</code> on-chip	$711 \cdot 10^6$	$1 \cdot 10^6$	$6 \cdot 10^2$
3.2 full profile generation on-chip	$711 \cdot 10^6$	204	$3 \cdot 10^6$

Table 5.2: Overview on all case studies, comparig the off-chip traffic of a traditional tracing system with the traffic produced by DiaSys. Most case studies depend strongly on the executed program and other factors, such as the update frequency of a profile. These assumptions are discussed within the case studies.

Furthermore, in case study 1 the message rate is set to 9090 messages/s, the maximum sustainable message rate for on-chip processing. For case study 1.2 it is assumed that 1% of transactions violate the protocol. When a profile is fully generated on-chip, it is sent off-chip once per second.

Figure 5.9 presents the data rate reductions in graphical form.

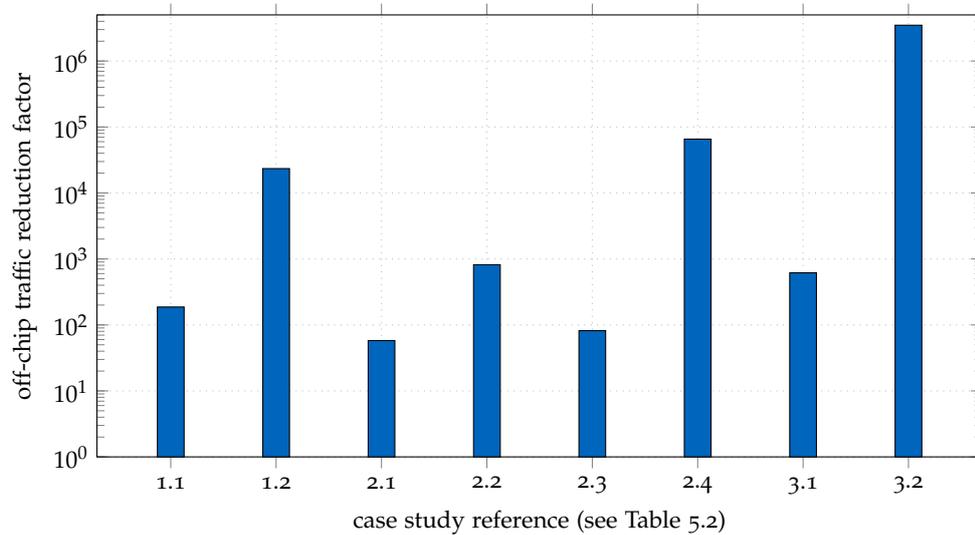


Figure 5.9: Graphical representation of the data rate reductions achieved by DiaSys in the presented case studies, compared to traditional tracing systems. Refer to Table 5.2 for more details on the shown case studies.

6 Conclusion and Outlook

In 1956, Herbert Benington wrote in an article on the software engineering processes used to develop SAGE, a massive defense project during the cold war. He concluded that “Eventually, programming should become a two-way conversation between the imprecise human language and the precise, if unimaginative, machine. The programmer will say, ‘Do this,’ and the machine will answer, ‘OK, but what happens if ...?’” [104]

In 2018, more than two generations of engineers and dozens of generations of “machines” (as they were called back then) later, we are still not there. Despite all the advances in software engineering, it is still the creative human mind that asks questions to the computer, not the computer asking “intelligent” questions to the programmer. Maybe we will never reach this level of sophistication. What is shame, however, is that the tools and methods that allow developers to ask a computer questions about the software execution are still inconvenient to use or wholly unavailable.

If we don’t want software development to happen blindfolded and outpaced by the rapid increase in available processing power, we need to invest more into tools and processes that help developers to observe and understand what they write. At its core this is what motivated the design of DiaSys, which we presented in this work.

Our work addresses non-intrusive runtime observation of software executing on embedded systems. Today, the insight into such systems is limited by the bandwidth of the off-chip interface, which is orders of magnitude below the amount of observation data available on-chip. The key idea of DiaSys is to move the processing of observation data partially onto the chip, thus avoiding the bottleneck, while providing the developer comparable insight into the software execution. The core of method behind DiaSys is the description of an observation and analysis task as dataflow program, which we call “Dia script.” To write such a script we contribute a novel domain-specific language, the Dia Language. It has been designed to enable a distributed execution of the observation data analysis, both on-chip and off-chip. However, this distinction does not need to be made by the developer writing the script: the Dia Compiler translates the high-level description to a target-specific representation, which is executed by the Dia Engine. The Dia Engine consists of both hardware components which are added to an observed chip design, and software components which complement the hardware implementation.

We have designed and implemented all three components of DiaSys, and used them in case studies taken from typical scenarios in debugging and testing: the manual and automated debugging of a race condition, and the creation of a function and a lock contention profile. In these case studies we show that by moving the processing of observation data partially on-chip, DiaSys can significantly reduce the off-chip traffic, and in consequence increase the insight into the software execution.

First we use DiaSys to manually debug a race condition bug, and evolve the manual debugging into an automated test. We then discuss two use cases of DiaSys which are especially challenging for today's tracing systems, since they generate large amounts of trace data. In the function profiling case study, we show that DiaSys can be used to create a function profile for an arbitrarily fast processor with a hardware cost of less than the DRAM infrastructure components. Finally, we employ observation data created from the execution of a PARSEC benchmark application to gain insight into real-world event and data rates.

The case studies have shown that DiaSys can be realized with reasonable hardware cost, and that it is able to reduce the off-chip traffic by several orders of magnitude (at least in the cases we have shown). On the other hand, the case studies also show areas in which future research is needed.

First, we feel that more work is needed in the Dia Language to find an optimal balance between usability, expressiveness, and compilability. Finding a good trade-off in this domain has been a recurring topic over many decades, and the search does not seem to end here.

Second, the evaluation of the Dia Engine implementation, especially the hardware implementation, presents opportunities for further research. The resource utilization could be lowered by sharing replicated structures, e.g. in the Core Event Generators which are present for every observed CPU core. Also, the processing overhead of the Diagnosis Processor could be reduced by including more hardware offloading, such as a hardware scheduler.

Finally, more research into defect patterns and bug detection algorithms is needed to better help developers detect sub-optimal or faulty program behavior. Initial work in this domain has been done by the author of this thesis in collaboration with Infineon, where we explored algorithms to detect wrong hardware configuration settings in [16] and inefficient locking implementations in [17]. Using DiaSys in these and other scenarios will undoubtedly uncover areas in which the current design, the implementation, or the dimensioning of DiaSys need to be extended.

For the first time, DiaSys introduces script-based, non-intrusive diagnosis to embedded developers. It unlocks unprecedented opportunities of sharing diagnosis knowledge in the form of Dia scripts, and automating their execution. While this does not make "the machine" ask questions during debugging, as Benington hoped for, it at least frees developers from asking the machine the same questions again and again.

Bibliography

- [1] IHS. Internet of Things (IoT) connected devices installed base worldwide from 2015 to 2025 (in billions). URL: <https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/>.
- [2] MarketsandMarkets. Embedded Systems Market worth 110.46 Billion USD by 2023 (press release), September 2017. URL: <https://www.marketsandmarkets.com/PressReleases/embedded-system.asp>.
- [3] Radiant Insights. Global embedded system market size worth \$214.39 billion by 2020 (press release), October 2015. URL: <https://www.radiantinsights.com/press-release/global-embedded-system-market>.
- [4] C. Ebert and C. Jones. Embedded Software: Facts, Figures, and Future. *Computer*, 42(4):42–52, April 2009. doi:10.1109/MC.2009.118.
- [5] International Technology Roadmap for Semiconductors, 2013. URL: <http://www.itrs.net/>.
- [6] BITKOM. Studie zur Bedeutung des Sektors Embedded-Systeme in Deutschland. Technical report, Berlin, 2008.
- [7] C. Jones, J. Subramanyam, and O. Bonsignour. *The Economics of Software Quality*. Prentice Hall, Upper Saddle River, NJ, July 2011.
- [8] G. Tassef. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project, 7007(011)*, May 2002. URL: <https://www.nist.gov/document/report02-3pdf>.
- [9] B. Hailpern and P. Santhanam. Software Debugging, Testing, and Verification. *IBM Systems Journal*, 41(1):4–12, January 2002. doi:10.1147/sj.411.0004.
- [10] G. J. Myers, C. Sandler, and T. Badgett. *The Art of Software Testing*. John Wiley & Sons, Hoboken, NJ, USA, 3 edition, November 2011.
- [11] B. Beizer. *Software Testing Techniques*. International Thomson Computer Press, Boston, MA, USA, 2nd edition, 1990.
- [12] P. Wagner, L. Li, T. Wild, A. Mayer, and A. Herkersdorf. Knowledge-Based On-Chip Diagnosis for Multi-Core Systems-on-Chip. In *edaWorkshop 15*, pages 39–45, Dresden, Germany, May 2015.

Bibliography

- [13] P. Wagner, L. Li, T. Wild, A. Mayer, and A. Herkersdorf. What happens on an MPSoC stays on an MPSoC - unfortunately! In *2016 International Symposium on Integrated Circuits (ISIC)*, December 2016. doi:10.1109/ISICIR.2016.7829711.
- [14] P. Wagner, T. Wild, and A. Herkersdorf. DiaSys: On-Chip Trace Analysis for Multi-processor System-on-Chip. In F. Hannig, J. M. P. Cardoso, T. Pi-onteck, D. Fey, W. Schröder-Preikschat, and J. Teich, editors, *Architecture of Computing Systems – ARCS 2016*, number 9637 in Lecture Notes in Computer Science, pages 197–209. Springer International Publishing, April 2016. doi:10.1007/978-3-319-30695-7_15.
- [15] P. Wagner, T. Wild, and A. Herkersdorf. DiaSys: Improving SoC insight through on-chip diagnosis. *Journal of Systems Architecture*, 75:120–132, April 2017. doi:10.1016/j.sysarc.2017.01.005.
- [16] L. Li, P. Wagner, R. Ramaswamy, A. Mayer, T. Wild, and A. Herkersdorf. A Rule-based Methodology for Hardware Configuration Validation in Embedded Systems. In *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems (SCOPES 2016)*, Sankt Goar, Germany, May 2016.
- [17] L. Li, P. Wagner, A. Mayer, T. Wild, and A. Herkersdorf. A Non-Intrusive, Operating System Independent Spinlock Profiler for Embedded Multicore Systems. In *Design, Automation and Test in Europe (DATE 2017)*, Lausanne, Switzerland, March 2017.
- [18] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas. Manifesto for Agile Software Development, 2001. URL: <https://www.agilemanifesto.org/>.
- [19] M. Beedle and K. Schwaber. *Agile Software Development With Scrum*. Prentice Hall, Upper Saddle River, NJ, USA, 1 edition, February 2002.
- [20] A. Zeller. *Why Programs Fail, Second Edition: A Guide to Systematic Debugging*. Morgan Kaufmann, Boston, 2 edition, June 2009.
- [21] C. Kaner. *Exploratory Testing*, November 2006.
- [22] D. Potts, R. Bourquin, L. Andresen, J. Andronick, G. Klein, and G. Heiser. Mathematically Verified Software Kernels: Raising the Bar for High Assurance Implementations. Whitepaper, July 2014.
- [23] A. R. Bernat and B. P. Miller. Anywhere, Any-time Binary Instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools, PASTE '11*, pages 9–16, New York, NY, USA, 2011. ACM. doi:10.1145/2024569.2024572.

- [24] A. B. T. Hopkins and K. D. McDonald-Maier. Debug support strategy for systems-on-chips with multiple processor cores. *IEEE Transactions on Computers*, 55(2):174 – 184, February 2006. doi:10.1109/TC.2006.22.
- [25] W. Orme. Debug and Trace for Multicore SoCs (ARM white paper), September 2008. URL: <https://www.arm.com/files/pdf/CoresightWhitepaper.pdf>.
- [26] V. Uzelac, A. Milenković, M. Burtscher, and M. Milenković. Real-time unobtrusive program execution trace compression using branch predictor events. In *Proceedings of the 2010 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES '10*, pages 97–106, New York, NY, USA, 2010. ACM. doi:10.1145/1878921.1878938.
- [27] ARM Limited. *ARM CoreSight Architecture Specification v3.0*. February 2017.
- [28] The Nexus 5001 Forum. Standard for a Global Embedded Processor Debug Interface, Version 3.0. Standard 5001-2012, IEEE- Industry Standards and Technology Organization (IEEE-ISTO), Piscataway, NJ, USA, June 2012.
- [29] IPextreme. Infineon Multi-Core Debug Solution: Product Brochure, 2008.
- [30] *Intel Trace Hub Developer's Manual*. Revision 1.0 edition, March 2015.
- [31] ARM Limited. CoreSight Technology System Design Guide, June 2010. URL: http://infocenter.arm.com/help/topic/com.arm.doc.dgi0012d/DGI0012D_coresight_dk_sdg.pdf.
- [32] ARM Limited. CoreSight Components Technical Reference Manual, July 2009. URL: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0314h/DDI0314H_coresight_components_trm.pdf.
- [33] ARM Limited. CoreSight Technical Introduction, August 2013. URL: http://infocenter.arm.com/help/topic/com.arm.doc.epm039795/coresight_technical_introduction_EPM_039795.pdf.
- [34] M. Leach. ETM vs PTM (Linaro coresight mailing list post), Mon Apr 3 20:20:23 UTC 2017. URL: <https://lists.linaro.org/pipermail/coresight/2017-April/000506.html>.
- [35] B. Walshe. ARM CoreSight Design Kits For Cortex A: Performance and Specifications for Cortex A series processors, March 2015. URL: http://www.arm.com/files/pdf/CoreSight_Design_Kits_PDF_CORTEX_A.pdf.
- [36] B. Walshe. ARM CoreSight Design Kits: Performance and Specifications for CoreSight for Cortex-R series processors, March 2015. URL: http://www.arm.com/files/pdf/CoreSight_Design_Kits_PDF_CORTEX_R.pdf.
- [37] R. Dees. *Nexus Revealed: An Introduction to the IEEE-ISTO 5001 Nexus Debug Standard*. incomplete draft, unpublished, 2012.

Bibliography

- [38] The Nexus 5001 Forum. Standard for a Global Embedded Processor Debug Interface, Version 2.0. Standard 5001-2003, IEEE- Industry Standards and Technology Organization (IEEE-ISTO), Piscataway, NJ, USA, December 2003.
- [39] Freescale Semiconductor, Inc. MPC565 Reference Manual, Rev 2.2, November 2005. URL: <https://www.nxp.com/docs/en/data-sheet/MPC565RM.pdf>.
- [40] A. Mayer, H. Siebert, and K. D. McDonald-Maier. Debug Support, Calibration and Emulation for Multiple Processor and Powertrain Control SoCs. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 3, DATE '05*, pages 148–152, Washington, DC, USA, 2005. IEEE Computer Society. doi:10.1109/DATE.2005.109.
- [41] A. Mayer, H. Siebert, and C. Lipsky. Multi-Core Debug Solution IP: SoC Software Debugging and Performance Optimization, May 2007.
- [42] A. Mayer, H. Siebert, and K. D. McDonald-Maier. Boosting Debugging Support for Complex Systems on Chip. *Computer*, 40(4):76 – 81, April 2007. doi:10.1109/MC.2007.118.
- [43] K.-U. Irrgang. *Modellierung von On-Chip-Trace-Architekturen für eingebettete Systeme*. Dissertation, Technischen Universität Dresden, Dresden, Germany, October 2014.
- [44] C. Hochberger and A. Weiss. Acquiring an exhaustive, continuous and real-time trace from SoCs. In *IEEE International Conference on Computer Design, 2008. ICCD 2008*, pages 356 – 362, October 2008. doi:10.1109/ICCD.2008.4751885.
- [45] T. Mathisen. Pentium Secrets: Undocumented features of the Intel Pentium can give you all the information you need to optimize Pentium code. *Byte Magazine*, 19(7):191–193, July 1994.
- [46] IBM. *Power ISA Version 3.0 B*. March 2017.
- [47] A. T. Sudhakar. "IMC Instrumentation Support" on the linuxppc-dev mailing list, 18 Apr 2017 10:03:06 -0700. URL: <https://www.mail-archive.com/linuxppc-dev@lists.ozlabs.org/msg117115.html>.
- [48] J. Braunes and R. G. Spallek. Generating the trace qualification configuration for MCDS from a high level language. In *Automation Test in Europe Conference Exhibition 2009 Design*, pages 1560–1563, April 2009. doi:10.1109/DATE.2009.5090911.
- [49] TASKING. TASKING Embedded Profiler User Guide (v1.0), September 2017. URL: https://www.tasking.com/support/profiler_user_guide.pdf.
- [50] M. S. Johnson. The Design of a High-level, Language-independent Symbolic Debugging System. In *Proceedings of the 1977 Annual Conference*, ACM '77, pages 315–322, New York, NY, USA, 1977. ACM. doi:10.1145/800179.810221.

- [51] P. Bates. Debugging Heterogeneous Distributed Systems Using Event-based Models of Behavior. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, PADD '88*, pages 11–22, New York, NY, USA, 1988. ACM. doi:10.1145/68210.69217.
- [52] P. C. Bates. Debugging Heterogeneous Distributed Systems Using Event-based Models of Behavior. *ACM Transactions Computer Systems*, 13(1):1–31, February 1995. doi:10.1145/200912.200913.
- [53] J. Lumpp, T. Casavant, H. Siegel, and D. Marinescu. Specification and identification of events for debugging and performance monitoring of distributed multi-processor systems. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 476–483, May 1990. doi:10.1109/ICDCS.1990.89317.
- [54] R. A. Olsson, R. H. Crawford, and W. W. Ho. A Dataflow Approach to Event-based Debugging. *Software—Practice & Experience*, 21(2):209–229, February 1991. doi:10.1002/spe.4380210207.
- [55] A. K. Iyengar, T. S. Grzesik, V. J. Ho-Gibson, T. A. Hoover, and J. R. Vasta. An Event-Based, Retargetable Debugger. *Hewlett Packard Journal*, 45:33–33, 1994.
- [56] M. Ducassé. Coca: An Automated Debugger for C. In *Proceedings of the 21st International Conference on Software Engineering, ICSE '99*, pages 504–513, New York, NY, USA, 1999. ACM. doi:10.1145/302405.302682.
- [57] G. Marceau, G. Cooper, S. Krishnamurthi, and S. Reiss. A dataflow language for scriptable debugging. In *19th International Conference on Automated Software Engineering, 2004. Proceedings*, pages 218–227, September 2004. doi:10.1109/ASE.2004.1342739.
- [58] G. Marceau, G. H. Cooper, J. P. Spiro, S. Krishnamurthi, and S. P. Reiss. The design and implementation of a dataflow language for scriptable debugging. *Automated Software Engineering*, 14(1):59–86, March 2007. doi:10.1007/s10515-006-0003-z.
- [59] M. Auguston. FORMAN-Program formal annotation language. In *Proceedings of the Fifth Israel Conference on Computer Systems and Software Engineering*, pages 149–154, Herzlia, Israel, May 1991. doi:10.1109/ICCSSE.1991.151186.
- [60] M. Auguston and P. Fritzson. PARFORMAN—an assertion language for specifying behavior when debugging parallel applications. In *EuroMicro Workshop on Parallel and Distributed Processing (PDP)*, January 1993.
- [61] M. Auguston, C. Jeffery, and S. Underwood. A Monitoring Language for Run Time and Post-Mortem Behavior Analysis and Visualization. In *Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003)*, Ghent, September 2003. arXiv:cs/0310025, doi:cs.SE/0309027.

Bibliography

- [62] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic Instrumentation of Production Systems. In *Proceedings of the General Track: 2004 USENIX Annual Technical Conference, ATEC '04*, Berkeley, CA, USA, 2004. USENIX Association.
- [63] F. C. Eigler, V. Prasad, W. Cohen, H. Nguyen, M. Hunt, J. Keniston, and B. Chen. Architecture of systemtap: A Linux trace/probe tool. 2005. URL: <https://sourceware.org/systemtap/archpaper.pdf>.
- [64] H. Garcia-Molina, F. Germano, and W. H. Kohler. Debugging a Distributed Computing System. *IEEE Transactions on Software Engineering*, SE-10(2):210–219, March 1984. doi:10.1109/TSE.1984.5010224.
- [65] Ú. Erlingsson, M. Peinado, S. Peter, M. Budiu, and G. Mainar-Ruiz. Fay: Extensible Distributed Tracing from Kernels to Clusters. *ACM Transactions on Computer Systems (TOCS)*, 30(4):13, January 2012. doi:10.1145/2382553.2382555.
- [66] J. Mace, R. Roelke, and R. Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 378–393. ACM, April 2015. doi:10.1145/2815400.2815415.
- [67] C. Fidge. Fundamentals of distributed system observation. *IEEE Software*, 13(6):77–83, November 1996. doi:10.1109/52.542297.
- [68] J. Gray. Why do computers stop and what can be done about it? In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12. Los Angeles, CA, USA, 1986.
- [69] P. M. Duvall, S. Matyas, and A. Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Professional, June 2007.
- [70] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.
- [71] G. Kahn. The Semantics of Simple Language for Parallel Programming. In *IFIP Congress*, pages 471–475, 1974.
- [72] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, May 1995. doi:10.1109/5.381846.
- [73] D. M. Rafi, K. R. K. Moses, K. Petersen, and M. V. Mäntylä. Benefits and Limitations of Automated Software Testing: Systematic Literature Review and Practitioner Survey. In *Proceedings of the 7th International Workshop on Automation of Software Test, AST '12*, pages 36–42, Piscataway, NJ, USA, 2012. IEEE Press.
- [74] J. Engblom. A review of reverse debugging. In *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference*, pages 1–6, Vienna, Austria, September 2012.

- [75] R. Ramler and K. Wolfmaier. Economic Perspectives in Test Automation: Balancing Automated and Manual Testing with Opportunity Cost. In *Proceedings of the 2006 International Workshop on Automation of Software Test, AST '06*, pages 85–91, New York, NY, USA, 2006. ACM. doi:10.1145/1138929.1138946.
- [76] ISO/IEC 9075-2:2016: Information technology – Database languages – SQL – Part 2: Foundation (SQL/Foundation). ISO/IEC Standard 9075-2:2016, International Organization for Standardization (ISO), December 2016.
- [77] TIOBE software BV. TIOBE Programming Community Index, July 2018. URL: <https://www.tiobe.com/tiobe-index>.
- [78] S. Cass and N. Diakopoulos. Interactive: The Top Programming Languages 2017, July 2017. URL: <https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2017>.
- [79] M. Koenen. Design and Implementation of a Debug Script Language for Many-core System on Chip. Master Thesis, Chair of Integrated Systems, Technical University of Munich, Munich, Germany, May 2016.
- [80] J. Loeckemann. Diagnosis Script Compiler for an Event-Based Trace System. Master Thesis, Chair of Integrated Systems, Technical University of Munich, Munich, Germany, September 2017.
- [81] ISO/IEC TS 17961:2013 - Information technology – Programming languages, their environments and system software interfaces – C secure coding rules. ISO/IEC Standard 17961:2013, International Organization for Standardization (ISO), November 2013.
- [82] MIRA Limited. *MISRA C:2012: Guidelines for the Use of the C Language in Critical Systems*. Nuneaton, UK, March 2013.
- [83] Carnegie Mellon University. *SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition)*.
- [84] A. Stefik, S. Hanenberg, M. McKenney, A. Andrews, S. K. Yellanki, and S. Siebert. What is the Foundation of Evidence of Human Factors Decisions in Language Design? An Empirical Study on Programming Language Workshops. In *Proceedings of the 22nd International Conference on Program Comprehension, ICPC 2014*, pages 223–231, New York, NY, USA, 2014. ACM. doi:10.1145/2597008.2597154.
- [85] S. Markstrum. Staking Claims: A History of Programming Language Design Claims and Evidence: A Positional Work in Progress. In *Evaluation and Usability of Programming Languages and Tools, PLATEAU '10*, pages 7:1–7:5, New York, NY, USA, 2010. ACM. doi:10.1145/1937117.1937124.
- [86] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, CA, USA, 5. edition, 2002.

Bibliography

- [87] A. V. Aho. *Compilers*. Pearson Addison-Wesley, Boston, MA, USA, 2. edition, 2007.
- [88] ISO/IEC 9899:2011 - Information technology – Programming languages – C. ISO/IEC Standard 9899:2011, International Organization for Standardization (ISO), December 2011.
- [89] C. K. Roy and J. R. Cordy. A Survey on Software Clone Detection Research. Technical Report 2007-541, School of Computing, Queen’s University at Kingston, Kingston, Ontario, Canada, September 2007.
- [90] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *Science of computer programming*, 74(7):470–495, May 2009. doi:10.1016/j.scico.2009.02.007.
- [91] I. D. Baxter, A. Yahin, L. Moura, M. Sant’Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377, November 1998. doi:10.1109/ICSM.1998.738528.
- [92] T. Flouri. *Pattern Matching in Tree Structures*. PhD Thesis, Czech Technical University, Prague, Czech Republic, September 2012.
- [93] XML Path Language (XPath) 3.1. W3C Recommendation, World Wide Web Consortium (W3C), March 2017. URL: <https://www.w3.org/TR/xpath-31/>.
- [94] T. Parr. *The Definitive ANTLR 4 Reference*. O’Reilly, Dallas, Texas, 2nd edition, January 2013.
- [95] P. Wagner. The Open SoC Debug Specification — Open SoC Debug 0.1 documentation. URL: https://opensocdebug.readthedocs.io/en/latest/02_spec/index.html.
- [96] S. Wallentowitz, P. Wagner, M. Tempelmeier, T. Wild, and A. Herkersdorf. Open Tiled Manycore System-on-Chip. *arXiv:1304.5081 [cs]*, April 2013. arXiv:1304.5081.
- [97] openrisc.io and authors. *OpenRISC 1000 Architecture Manual (Architecture Version 1.2, Document Revision 1)*. August 2017.
- [98] Xilinx, Inc. *7 Series FPGAs Configurable Logic Block User Guide (UG474)*. v1.8 edition, September 2016.
- [99] Xilinx, Inc. *Vivado Design Suite User Guide: Synthesis (UG901)*. v2017.4 edition, December 2017.
- [100] Xilinx, Inc. *7 Series FPGAs Memory Resources User Guide (UG473)*. v1.12 edition, September 2016.

- [101] Lauterbach GmbH. Long-Term Trace ETMv3, December 2010.
- [102] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.
- [103] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.
- [104] H. D. Benington. Production of Large Computer Programs. *Annals of the History of Computing*, 5(4):350–361, October 1983. doi:10.1109/MAHC.1983.10102.