Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2010

MSc THESIS

Quantitative Analysis and Visualization of Memory Access Patterns

Marco Corina

Abstract

As the rate of improvement of processor performance has greatly exceeded the rate of improvement of memory performance, the communication between the (general-purpose) processor and the memory sybsystem became the main obstacle for achieving overall system performance improvements. Conversely, more and more modern application require a considerable amount of computing power. The introduction of heterogeneous reconfigurable systems are increasingly gaining popularity due to their ability to speed up the execution of an application. However, the widespread utilization of such systems through the industry seems inconvenient due to the shortage of tools guiding developers throughout the entire development process. Furthermore, the introduction of heterogenous reconfigurable systems does not still solve the processor-memory dilemma. Hence, there is a compelling need for tools that facilitate the development of applications of heterogeneous platforms and that help the developer gain more insight in the memory access behaviour of an application. In this thesis, a set of sophisticated memory access analysis tools is presented, which provide detailed analysis on the memory access behaviour of an application. This toolset is developed in the context of the Delft Workbench and hArtes. The development of one of the tools in the toolset is the main contribution to this thesis. In this thesis, the toolset will be described and motivated. Emphasis will be put on the tool that is developed during this thesis. Finally, a case study on a real application is conducted, showing the potentialities of the tool.

CE-MS-2010-15

Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Quantitative Analysis and Visualization of Memory Access Patterns

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Marco Corina
born in The Hague, The Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Quantitative Analysis and Visualization of Memory Access Patterns

by Marco Corina

Abstract

As the rate of improvement of processor performance has greatly exceeded the rate of improvement of memory performance, the communication between the (general-purpose) processor and the memory sybsystem became the main obstacle for achieving overall system performance improvements. Conversely, more and more modern application require a considerable amount of computing power. The introduction of heterogeneous reconfigurable systems are increasingly gaining popularity due to their ability to speed up the execution of an application. However, the widespread utilization of such systems through the industry seems inconvenient due to the shortage of tools guiding developers throughout the entire development process. Furthermore, the introduction of heterogenous reconfigurable systems does not still solve the processor-memory dilemma. Hence, there is a compelling need for tools that facilitate the development of applications of heterogeneous platforms and that help the developer gain more insight in the memory access behaviour of an application. In this thesis, a set of sophisticated memory access analysis tools is presented, which provide detailed analysis on the memory access behaviour of an application. This toolset is developed in the context of the Delft Workbench and hArtes. The development of one of the tools in the toolset is the main contribution to this thesis. In this thesis, the toolset will be described and motivated. Emphasis will be put on the tool that is developed during this thesis. Finally, a case study on a real application is conducted, showing the potentialities of the tool.

Laboratory	:	Computer Engineering
Codenumber	:	CE-MS-2010-15

Committee Members :

Advisor:	Koen Bertels, CE, TU Delft
Chairperson:	Koen Bertels, CE, TU Delft
Member:	Eelco Visser, ST, TU Delft
Member:	Georgi Kuzmanov, CE, TU Delft

To my parents

Contents

List of Figures **vii**

List of Tables **ix**

Acknowledgements **xi**

1 Introduction **1**
- 1.1 Problem Definition . . . 2
- 1.2 Thesis outline . . . 3

2 Background **5**
- 2.1 The Need for Heterogenous Reconfigurable Architectures . . . 5
- 2.2 The MOLEN Processor and Programming Paradigm . . . 7
- 2.3 Memory Communication Bottleneck . . . 9
  - 2.3.1 Hardware Exploitation Methods . . . 9
  - 2.3.2 Software Exploitation Methods . . . 11
  - 2.3.3 Concluding Remarks . . . 14
- 2.4 Motivation for Improved Memory Analysis . . . 15
- 2.5 Hardware/Software Co-design . . . 15
  - 2.5.1 Design Space Exploration . . . 16
  - 2.5.2 Hardware/Software Partitioning and Mapping . . . 16
- 2.6 Delft Workbench . . . 17
- 2.7 hArtes . . . 20
- 2.8 Motivation for Dynamic Memory Analysis Tools . . . 21
- 2.9 Conclusion . . . 22

3 Analysis Techniques **23**
- 3.1 Program Analysis . . . 23
- 3.2 Dynamic Binary Instrumentation . . . 24
  - 3.2.1 DynamoRIO . . . 25
  - 3.2.2 Valgrind . . . 25
  - 3.2.3 pin . . . 26
- 3.3 Pin . . . 26
  - 3.3.1 Evaluation of Pin . . . 28
  - 3.3.2 Comparison of Pin against Other DBIs . . . 28
  - 3.3.3 Pro et Contra of Pin . . . 29
- 3.4 Pin Instrumentation API . . . 29
- 3.5 Calling Pin . . . 31
- 3.6 Conclusion . . . 32

4 QUAD: Sophisticated Memory Patterns Analysis Tools 33
4.1 QUAD - Quantitative Usage Analysis of Data 33
4.1.1 QUAD Objective 34
4.1.2 QUAD Design and Implementation 34
4.1.3 How QUAD Works 36
4.1.4 QUAD's Example Outputs 37
4.2 tQUAD 38
4.2.1 tQUAD Objectives 38
4.2.2 tQUAD Design and Implementation 39
4.2.3 How tQUAD Works 41
4.2.4 tQUAD's Example Output 42
4.3 Concluding Remarks - The xQUAD Tool 42

5 The xQUAD Tool 45
5.1 Introduction and overview of xQUAD 45
5.2 Background Information 46
5.2.1 Object File Overview 46
5.2.2 DWARF Debugging Format 49
5.2.3 DWARF Consumer Library 51
5.3 xQUAD Architecture 52
5.3.1 xQUAD Design and Implementation 53
5.3.2 DWARF Debugging Information Module 54
5.3.3 Pin API calls and Analysis Routines 58
5.3.4 How xQUAD Works: Functionalities and Restrictions 60
5.4 xQUAD Analysis Output Examples 61
5.4.1 Memory Map Report File 61
5.5 Detailed Variable report file 63
5.6 Conclusion 63

6 Case Study: Wave Field Synthesis 65
6.1 Introduction to Wave Field Synthesis 65
6.2 Experimental Setup 66
6.3 Memory Access Behaviour of WFS 67
6.4 Ranking of Memory Intensive WFS Kernels 70
6.5 Conclusion 73

7 Conclusion and Future Work 75
7.1 Conclusion 75
7.2 Future Work 76

Bibliography 80

List of Figures

2.1 The MOLEN machine organization 8
2.2 The DWB Architecture 17
2.3 The DWARV Toolset 19
2.4 The hArtes Toolchain 20

3.1 Pin's Architecture 27
3.2 Pin example of instrumentation and analysis routines 30

4.1 Profiling Framework of QUAD within DWB 35
4.2 Architectural overview of QUAD 35
4.3 Implementation overview of QUAD 36
4.4 XML format of producer/consumer binding 38
4.5 Profiling Framework of tQUAD within DWB 39
4.6 Architectural overview of tQUAD 39
4.7 tQUAD instruction instrumentation pseudocode 40
4.8 tQUAD routine instrumentation 41

5.1 Sections and segments of an ELF 48
5.2 Segments of an executable object file mapped in memory 49
5.3 DWARF DIEs sample output 50
5.4 C Program and DWARF description of program 51
5.5 Architectural overview of xQUAD 52
5.6 Example of the MONITOR file passed to the xQUAD tool 54
5.7 xQUAD main function 55
5.8 DWARF Depth-First Tree Traversal 56
5.9 xQUAD memory map example 61
5.10 Example of memory visualization 62

6.1 Principle of WFS 66
6.2 Snapshots of the heap memory usage of WFS 67
6.3 Snapshots of the heap memory usage of WFS 68
6.4 Snapshots of the stack memory usage of the WFS 70
6.5 Snapshots of the stack memory usage of the WFS 71
6.6 Snapshots of the stack memory usage of the WFS 72

List of Tables

4.1 Phases in the execution path of the *hArtes wfs* application. 42

5.1 Example of statistics for the wav_load function from the WFS application 62
5.2 Example of statistics for the wav_load function from the WFS application 63

6.1 Memory usage statistics for the *hArtes wfs* application. 68
6.2 Flat profile for the *hArtes wfs* application. 69
6.3 Rank of WFS kernels 72

Acknowledgements

First of all, I would like to thank my advisor Prof. Koen Bertels, for his support and guidance during this thesis. Further, I would like to thank Arash Ostadzadeh for his invaluable help, it would have been much harder without his assistance. Also, I would like to thank Roel Meeuws for being there when I needed help. Last but not least, I would like to thank my friends and family for their support.

Marco Corina
Delft, The Netherlands
June 18, 2010

Introduction 1

In the past decennia, the quest for greater performance processors continuously grew to very complex architectures and phenomenal speeds[1]. However, the rate of improvement of processors speed exceed the rate of improvement of DRAM (Dynamic Random Access Memory) memories speed. This *processor-memory gap*[21] is continuously becoming a problem in the sense that it is the primary obstacle in improving the overall performance of computing systems. Ideally, a perfect memory system would supply immediately data requested by the processor. However, this is technologically as well as economically an infeasible solution. The advent of cache memories has alleviated the bottleneck caused by the slowness of memory systems, resulting in a memory hierarchy with multiple levels of caches, exposing the principle of locality[2]. Unfortunately, a memory hierarchy including caches does not still solve the *memory-wall* problem, meaning that the speed of an application is still bounded by the speed of the memory system. This is especially true for many data-intensive applications with memory behaviours that are characterized by large working sets and streaming data accesses, which cannot fit into the on-chip caches[3].

Conversely, modern applications require even more processing power than ever. This means that even though processors work at an edge speed, which is practically as fast as it can get with this technology, it is still not enough to cope with the latest developments and desires.

To cope with this problem, there is a need for an alternative architecture other than only the conventional general-purpose processor architecture. Even multi-core architectures do not deliver the expected improvements, as most applications are not parallelized, or they cannot be parallelized efficiently to unleash the power of multi-core systems. Hence applications cannot fully benefit from multi-core architectures. Besides the need for an increased computational power, today, there is also an increasing demand in (mobile) embedded technology. This brings other aspects into the design of an architecture that are to be considered, one of which is lower power consumption. This aspect makes a conventional architecture an unfeasible solution for embedded systems in general.

Designers have therefore increasingly used ASICs (Application Specific Integrated Circuits) along with a general-purpose processor to gain performance speed-up for, e.g., expensive media encoding and decoding algorithms and signal processing. These *heterogeneous systems* permit to deploy multiple types of processing elements within a single target platform, allowing each element to perform the task(s) to which it is best

[1]Furthermore, the reached sub-atomic sizes of modern processors has also put an horizon to Moore's prediction about technological advance, which requires a new way of increasing computing performance.

[2]Briefly, this principle states that accessed memory words will be accessed again quickly (temporal locality) and that memory words adjacent to an accessed word will be accessed soon after the access in question (spatial locality)

[3]Due to the low density and high cost of SRAM, L2 cache memory is usually limited between 4 and 12 MB on state-of-the-art processors, which is not large enough for many multimedia applications.

suited[35]. Elements that constitute these systems can be, for instance, ASICs, Field Programmable Gate Arrays (FPGAs), Graphic Processing Units (GPUs), Digital Signal Processors (DSPs), and the conventional commodity processors.

Each of the above mentioned components has it advantages, like high flexibility (as in the case of general-purpose processors) or high performance (e.g. ASICs). However, reconfigurable logic in the form of FPGAs is becoming increasingly popular as a computing component. Favoured by the technological advance of the latest years, FPGAs have increased tremendously in size and speed. Additionally, today FPGAs have built-in logic that is tailored to the needs of different application domains, including embedded processing cores, high-performance arithmetic units, specialized I/O functionalities, etc. Thanks to this and to the fact that FPGAs can achieve high performances by exploiting both fine-grained as coarse-grained parallelism present in applications, they have been a very centered topic in the research community, favouring the evolution of the concept of *reconfigurable computing*. Due to its potential to greatly accelerate a wide variety of applications[37, 11], this discipline is rapidly increasing. Reconfigurable computing aims at filling the gap between the performance of hardware while maintaining the software flexibility and upgradability. This is achieved by mapping the computational intensive tasks of an application onto the reconfigurable units and maintaining the parts of the application (or tasks) that cannot be performed efficiently on the reconfigurable logic in the general-purpose processor, like data-dependent control and memory accesses. Furthermore, reconfigurable hardware presents advantages like reduced energy and power consumption, which are vital for mobile and embedded systems.

1.1 Problem Definition

While presenting various advantages over custom hardware design, reconfigurable computing also facilitates the growth of heterogeneous computing. However, the widespread utilization of such systems through the industry seems to be inconvenient due to the shortage of tools guiding developers throughout the entire development process. Developing reconfigurable hardware requires knowledge of hardware design methods and tools that most system programmers do not have. A remedy to this problem would be to develop an hybrid programming model that abstract hardware away. Unfortunately, current programming models are still immature and do not permit to fully exploit the capabilities of hybrid general-purpose processors and FPGA designs[23, 4]. Furthermore, the introduction of heterogeneous reconfigurable architectures will still suffer from the *processor-memory gap*[19], even though the evolution of FPGA technology will supposedly include substantial memory resources by which the *memory wall* may be alleviated[46].

Hence, the need for tools that facilitate the development of applications on heterogeneous platforms on one side and tools that help the developer gain more insight in the memory utilization of an application and on how to optimize this on the other side, is of huge importance. Inspecting the behaviour of an application in general, and the actual *pattern of memory accesses* in particular, is an essential aspect of carrying out effective optimizations for the application development of reconfigurable systems. As a result, many research initiatives are emerging that target support tool for application behaviour analysis from different perspectives.

The Delft Workbench (DWB) project (Chapter 2.6) at Delft University of Technology and the hArtes project (Chapter 2.7) target the research and development of heterogeneous reconfigurable systems throughout the entire design cycle, from code profiling to compilation. These toolchains are designed and developed considering the actual hardware/software co-execution on real, heterogeneous, hardware. Hence, both DWB and hArtes should be aware of the memory characteristic behaviour of applications during their profiling stages, which would be accomplished by including some *dynamic profiling tools*. They would provide important information to developers for making critical decision during various hardware/software co-design stages, by considering not only a *computationally intensive task* but also taking into consideration the *memory behaviour* of the task(s).

For this purpose, a set of *memory behaviour analyzers* are developed at Delft University of Technology, with the primary intention to deliver detailed information about the memory usage of applications, and possibly discovering *memory-related bottlenecks*. The results retrieved from these tools can be of invaluable importance during hardware/software partioning steps as well as for gaining information about application code optimizations.

1.2 Thesis outline

This thesis is organized as follows. Chapter 2 discusses the issues introduced in this introduction in more details, and presents some related research around memory optimization. Also it introduces the field of hardware/software co-design, focusing on the research and tools done at Delft University of Technology. Chapter **??** introduces analysis techniques that can be used for the development of tools. In this chapter, more emphasis is put on the specific analysis methods used during this thesis. Chapter 4 presents the work done at Delft University of Technology regarding dynamic memory analysis inside DWB and hArtes. Chapter 5 presents a tool which augments the analysis of the tools described in Chapter 4. The tool in Chapter 5 is the main contribution of this thesis. Chapter 6 validates the tool in Chapter 5 by investigating a real application. Finally, Chapter 7 summarizes the work done in this thesis and discusses the results and future directions in this development.

Background 2

As introduced ealier, the advent of heterogeneous systems incorporating reconfigurable hardware promises great opportunity for application speed-up. However, the ever incresing gap between processor and memory performance makes the memory subsystem the main limiting factor for performance improvement. Even worse, as this gap continues to grow, this *memory-wall* will critically degrade the performance of the systems. This problem becomes even more significant when the target architecture has reconfigurable hardware coupled with a general purpose processor. In fact, the *communication* between reconfigurable unit(s), memory and processor constitute, with a high probability, a *communication bottleneck* of the system, which would automatically lead to computational stalls, degrading the performance of the whole system.

Besides being a limiting factor in terms of performance, the memory usage of systems characterize also the power consumption of these systems. These factors are especially important in embedded systems, as the demand for high-performance, low-power devices continues to grow. Hence, techniques must be found to prevent or limit the incurring stalls.

Additionally, for heterogeneous reconfigurable systems to be widely accepted in the industry, tools are needed that guide developers through the entire process. These tools can be categorized under *hardware/software co-design* methodology tools.

This chapter reviews the concept of heterogenous reconfigurable computing, along with its advantages compared to traditional computing architectures. Next, a review is done on the related research around the *memory communication bottleneck* problem, and various techniques for improving the memory access patterns of programs are introduced, both at software and at hardware level. Subsequently, the hardware/software methodology is presented, briefly explaining two areas in this methodology which are closely related to the design process of heterogeneous reconfigurable computing, i.e. *hardware/-software partitioning* and *design space exploration.* Furthermore, this chapter describes the ongoing research in Delft University of Technology related to reconfigurable computing, i.e. the MOLEN processor, the Delft Workbench and the hArtes, an European project.

2.1 The Need for Heterogenous Reconfigurable Architectures

The ever increasing demand for high performance systems has been tackeled since a few decades. As the complexity of applications increases, achieving overall high performance systems requires to change the archictectural design on which these applications are deployed. This is especially true for multimedia applications.

As mentioned before, a conventional architecture containing a software programmed general-purpose processor does not offer the desired speed-ups, as the clock speed and/or throughput of the processor is not enough to cope with these computation- and data-hungry applications. However, this class of processors offer a very flexible solution to the developer, as executing different operations is allowed by executing different instructions on its *Instruction Set Architecture* (ISA) without changing its hardware. If a certain operation cannot be executed with a specific instruction, the same operation can be executed by a sequence of instructions[1].

On the other hand, Application Specific Integrated Circutis (ASICs) are very efficient in terms of computational performance, size and power consumption, as they are designed specifically for one (or some) dedicated computational task(s). However, once fabricated, these devices cannot be adjusted. If the need occurs, the entire system needs to be redesigned. This expensive process in terms of non-recurring engineering costs and time-to-market represents the main drawback of ASIC designs.

Reconfigurable architectures, in turn, fills the gap between hardware and software solutions, by offering software-like flexibility at a performance rate comparable to hardware solutions. The growth of this field is driven largely by the latest developements in *Field Programmable Gate Array* (FPGA) technology, which permit to implement commodity off-the-shelf FPGA as the main reconfigurable device, reducing time-to-market and costs of reconfigurable systems.

FPGAs excel at implementing applications as highly parallel circuits, thus yielding fast performance. Especially for streaming applications with a lot of data parallelism, where the same repetitive transformations are applied to a large amount of data. However, for applications that do not exhibit enough data parallelism, FPGAs perform poorly. Therefore, for the more sequential operations, a GPP is a better choice as the clock rate of an FPGA is usually one order of magnitude less then the clock rate of a GPP. Hence, the intrinsic characteristics of the application determine the FPGA performance, and thus the choice for the right architecture. Applications determine how much raw parallelism exists, how exploitable it is, and how fast the clock can operate[19].

In order to have a flexible architecture, which would permit to obtain such performances while offering support for a wide range of applications, the combination of general-purpose processors and reconfigurable hardware units into a *heterogenous architecture* achieves this. In such an architecture, the GPP performs various data-dependent control and non-parallelizable operations while the (parallel) computational intensive segments of an application are mapped into the reconfigurable hardware. The MOLEN polymorphic processor[42] is an example of a heterogeneous platform, incorporating both a general-purpose processor and a reconfigurable processing unit. Several architectures are proposed in this field ([20, 18, 26, 8] are examples thereof), however it is not the intention of this thesis to present a complete overview of architectures that augment a programmable processor with one (or more) reconfigurable hardware units. As the MOLEN polymorphic processor is the target processor platform in the work presented in this thesis, its architecture will be described in the next section.

[1]This is of course peculiar to the architecture type. To some extend, an ISA may be more *application specific* than other ISAs (e.g. embedded computing vs. desktop computing). Nevertheless, this class of processors are all considered general-purpose.

2.2 The MOLEN Processor and Programming Paradigm

Reconfigurable hardware coexisting with a core general-purpose processor has been considered by several researcher as a good candidate for speeding up applications[11, 37]. The MOLEN polymorphic processor[42] is such an architecture, which defines how a general-purpose processor (GPP) interacts with one or more (reconfigurable) co-processors. Among the introduced heterogenous architectures, the MOLEN architecture presents various contribution to the shortcomings in the existing reconfigurable computing solutions. These shortcomings are summarized and described below.

Opcode Space Explosion Traditionally, a common approach to reconfigurable computing is to introduce a new instruction for each portion of the application mapped on the FPGA. The consequence is that, to implement a broad number of applications, the opcode space of the application's specific instructions occupies a considerable part of the FPGA space. The designer and the compiler are limited by this unused space.

No modularity Each approach has its own specific definition and implementation targeting a specific reconfigurable hardware. This makes porting an application to a new reconfigurable platform a burdensome job.

Limitation of the number of parameters In a number of approaches, only a limited number of input and output parameters is possible.

No support for parallel execution Many architectures do not take into account the possibility of a parallel execution of data-independent sequential operations, which is an important and powerful feature of FPGAs.

The MOLEN machine organization, depicted in figure 2.1, is composed mainly from the Core Processor, i.e. a general-purpose processor, and the Reconfigurable Processor (RP). Instructions and data are fetched from memory by the Instruction Fetch and the Data Fetch, respectively. The Arbiter performs a partial decoding of the instructions and issue them to the corresponding unit, i.e. to the general-purpose processor or reconfigurable processor. The Memory Unit is responsible for distributing (collecting) data. The reconfigurable processor consists further from a *Custom Configured Unit* (CCU) and a $\rho\mu$-code processor. The CCU consists of a reconfigurable hardware unit (e.g. an FPGA) and a memory unit. An application's code runs on the GPP except the parts of the application that are chosen to be implemented in hardware in order to speed up program execution. Data exhange between the GPP and the RP is performed via the exchange registers (XREGs).

In order to target the $\rho\mu$-code processor, the MOLEN Programming Paradigm[41] is proposed. This is a sequential consistency paradigm for programming reconfigurable processors, possibly including a general-purpose processor. This paradigm permits the execution of hardware operations in parallel, and is intended (currently) for single program execution. To overcome the earlier mentioned opcode explosion shortcoming of other reconfigurable programming paradigms, the MOLEN programming paradigm requires only a one-time ISA (Instruction Set Architecture) extension of, at most, eight

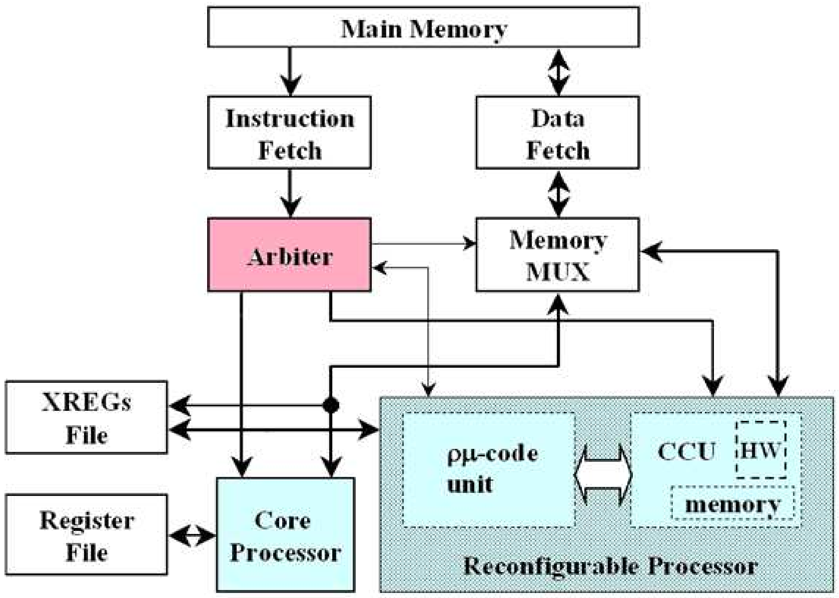


Figure 2.1: The MOLEN machine organization

instruction (denoted as polymorphic ISA, or πISA) to provide a large user reconfigurable operation space. Nevertheless, the smallest set of MOLEN instructions needed to provide a working scenario for an arbitrary number of applications consists of four basic instruction, namely the SET, EXECUTE, MOVTX, and MOVFX.

The SET phase, initiated by a SET instruction, provides a general approach for loading an arbitrary number of configurations to the reconfigurable part of the reconfigurable processor. This instruction requires a single parameter, i.e. the beginning address of the configuration microcode. By detection of a SET instruction, the Arbiter reads every sequential memory address until detection of the *end_op* microinstruction, which denotes the termination of the code supposed for hardware execution. The $\rho\mu$-code unit will then ensure that the data fetched from memory is redirected to the reconfigurable unit memory[2]. After completion of the SET phase, the CCU is configured to perform a specific operation. The actual operation is performed in the EXECUTE phase initiated by the EXECUTE instruction. This instruction refers to a single parameter, i.e. the address of the microcode to execute on the CCU, which is configured earlier during the SET phase. The microcode sequence is also terminated by the *end_op* instruction. The MOVTX and MOVFX instruction are used for moving values to and from the general-purpose register and exchange registers (XREGs).

[2]MOLEN only uses BRAM on the FPGA for data and instruction memory. This limits the instruction size to the number of available BRAM

2.3 Memory Communication Bottleneck

While important, increasing the computational speed of an application, by mapping parts of this application in reconfigurable hardware when a certain degree of parallelism is exhibited, is not the only aspect that must be taken into account. To efficiently use the parallel hardware capabilities, the system must provide enough data to the computational units. This puts a performance bound on the input/output capabilities of the system, i.e. computational power will have no benefit unless the required data is fed to the hardware units timely. Applications that use large data sets with few or no data dependence, a peculiarity of multimedia applications, are an example that need enough *memory bandwidth* for exploitation of parallelism. On heterogeneous reconfigurable architecture, this need becomes even more evident, as different processing elements are present on the same platform, which need to communicate with the main memory module. Usually, an increase in processing elements (of whatever nature they are) is directly proportional to the need for increasing I/O capabilities. Thus, the performance of an application mapped into a heterogeneous reconfigurable platform is largely determined by how much exploitable parallelism is available, and by the ability of the system to provide data to keep the parallel hardware operational[19]. The latter is the *processor-memory communication bottleneck* that could result as the main obstacle for performance improving, if not degrading, the overall system performance.

As the exponential factor by which processors have increased their performance is much higher than the exponential factor of perfomance improvement of memory, solutions for solving the *processor-memory bottleneck* have been extensively studied in at least the last three decades. Hardware as well as software solutions are applied, although these solutions only seem to alleviate this phenomenon, keeping it as the main obstacle for achieving speed-up of systems. The ideal solution for this problem would be to discover a memory technology that has a high capacity and whose speed scale with that of a processor. Unfortunatly technology has not come that far yet.

To cope with this gap, many research has been carried out for a better exploitation of the memory subsystem, improving the communication between memory and (general-purpose) processor.

In the next sections, a few methods are briefly introduced which address the amelioration of the processor-memory interface. Along with the introduction of these methods, an explanation is also given why these systems are not fully capable of filling the processor-memory gap.

2.3.1 Hardware Exploitation Methods

An explanation of hardware exploitation techniques will be given in this section. These techniques aim at improving the communication latency between the processor and the memory subsystem. The following examples will give an overview to the most known techniques, and provide by no mean an exhaustive background in this topic.

Caches To limit the latency of data accesses, processors make use of a hierarchy of memories. This hierarchy consists of on-chip and off-chip memories. Because off-chip memories stall the processor for a long time (100-200 processor cycles), creating a hierarchical memory subsystem with (multilevel) caches improves the access latency[21]. Cache parameters such as cache size, line size and associativity, can significantly influence the overall systems performance with respect to both power and execution time. In modern day processors, there is even a deeper hierarchy of caches, with the consequence that cache miss latencies have become extremely large when compared to processor clocks. As a consequence, to achieve good performance, an application must exhibit such a memory reference that is able to exploit caches well, i.e. the reference pattern must exhibit high spatial, temporal and processor locality[24].

Hence the benefit that an application can gain from caches depends on the intrinsic nature of the application. In this sense, the programmer has the possibility to achieve better performance by restructuring the data or code of the application and thus to change the memory reference pattern.

However, even though the space caches in state-of-the-art processors can reach up to 80% of the total logic, these large caches are not effective for applications that present a streaming data access nature and large working sets[34].

Scratch-pad Memory The above described caches are traditionally part of a familiar target architecture consisting of a processor core, data cache(s), and external main memory. However, this target architecture is not always present, especially in case of embedded systems. In fact, as embedded systems are often designed for a specific application, the designer may use a more unconventional architecture that suits the specific application under consideration. One such alternative is *scratch-pad memory*[33]. Scratch-pad is an on-chip memory, and is sometimes also referred to as on-chip SRAM. It referes to data memory on-chip mapped in a different memory space than the off-chip memory, but that uses the same address and data busses as the off-chip memory. The main difference between scrath-pad and data cache is that the former guarantees a single-cycle access time, whereas data caches are subject to *cache misses*, thus increasing the memory latency.

Convey Memory Controllers The Convey HC-1 Computer[8], a hybrid architecture integrating an FPGA-based, reconfigurable coprocessor with an industry standard Intel®64 processor, implements its own high bandwidth memory subsystem that logically shares a cache-coherent global memory space between the host, general-purpose processor, and the coprocessor. To support the bandwidth demands of the coprocessor, 8 Memory Controllers support a total of 16 DDR2 memory channels, which provide an aggregate of over 80GB/sec of bandwidth. To improve efficiency of the memory subsystem, the memory controllers support standard DIMMs as well as Convey designed Scatter-Gather DIMMs (SG-DIMMs). The SG-DIMMs allow access to physical memory by individual words instead of 64-byte cache lines (as the host does). Accessing by 8-byte blocks reduces the ineffiencies encountered when accessing memory by non-unity strides (or randomly) with a cache-based system.

2.3.2 Software Exploitation Methods

Besides techniques that change the hardware for achieving a better usage of the memory subsystems, there also exists software-based techniques that do the same. Even small changes in an application can result in considerable performance gains. Changing an application to optimally use the underlying memory sub-system can be especially useful in data intensive applications where large amount of data is accessed in a streaming fashion. Hence, applying software transformations to an application can result in a better exploitation of data locality, thereby improving its memory specific behaviour, with reasonably contained costs compared to a hardware method. Transformations can be either done manually, performed at a high level of abstraction, i.e. at the source code level, before compile-time, or automatically, meaning that the compiler itself performs the optimizations to the source code at compile-time. Customizing the code to efficiently exploit the memory hierarchy can deliver significant performance gains, both in terms of speed as of power.

Program transformation can be divided into *control transformation* and *data layout transformation.* In control transformations, the access patterns of a program is changed in order to improve cache reuse. This is mostly applied to loops, as these usually represent the most time consuming parts of an application. Data layout transformation on the other hand, reorder the placement of data and/or instructions in the off-chip memory, in order to improve cache reuse without having to change the order of memory references.

Both data layout transformations and control loop transformations have the desirable property of reducing the memory bandwith requirements for a particular application. Following, some widely used control and data transformations are introduced.

Control Transformations The primary emphasis is on loops, since that is generally where the most of the execution time is spent. Traditionally, loop transformations have been the main technique used to improve locality by changing the access pattern as a result of changing the order of execution of loop interations. The effect of loop transformations is local, i.e., a loop transformation affects only the loop nest to which it is applied, and both temporal and spatial locality may improve as a result.

Loop Interchange Loop interchange modifies a nested loop by switching the place of the inner loop with the outer loop. This technique can result in maximizing the use of data in a cache block. The following example shows how this technique is applied.

```
/* Before */
for (j = 0; j < 100; j++)
        for (i = 0; i < 5000; i++)
                x[i][j] = 2*x[i][j];

/* After */
for (i = 0; i < 5000; i++)
        for (j = 0; j < 100; j++)
                x[i][j] = 2*x[i][j];
```

Loop Fusion This technique combines two or more loops into a sigle one. For this transformation to be legal, two conditions must be satisfied. First, the loops must iterate over the same values. Second, the loop bodies must not have dependencies that would be violated if they are executed together[45]. The example below demostrates this.

```
/* Before */
for (i = 0; i < 100; i++)
        x[i] = y[i];
for (i = 0; i < 100; i++)
        z[i] = y[i]+2;

/* After */
for (i = 0; i < 100; i++)
        x[i] = y[i];
        z[i] = y[i]+2;
```

The main benefits of this optimization are data locality improvements and reduced loop overhead. However, it can be the case for some architctures that provide better performance if the loop is splitted in two instead. This is explained next.

Loop Fission Loop fission (or loop distribution) is the opposite of loop fusion, i.e. decomposing a sinle loop into smaller loops. This can result in a better data locality exploitation when the loops iterate over large values. Also, in case of multi-core processors, the two loops can be executed in parallel on each processor.

```
/* Before */
for (i = 0; i < 100; i++)
        x[i] = y1[i];
        z[i] = y2[i];

/* After */
for (i = 0; i < 100; i++)
        x[i] = y1[i];
for (i = 0; i < 100; i++)
        z[i] = y2[i];
```

Loop Tiling Also known as loop blocking, this loop transformation breaks up a loop into a set of nested loops, with each inner loop performing the operations on a subset of the data[45].

```
/* Before */
for (i = 0; i < 100; i++)
        for (j = 0; j < 100; j++)
                x[i] = y[i][j] * z[i];

/* After */
for (i = 0; i < 100; i += 2)
        for (j = 0; j < 100; j += 2)
                for (k = i; k < min(i + 2,n); k++)
                        for (h = j; h < min(j + 2,n); h++)
                                x[k] = y[k][h] * z[k];
```

Here, the original loop is broken into tiles of size two. Each loop is splitted into two loops. The result is is a completely different pattern of accesses accross the y array: instead of waling across one entire row, the transformed code walks through rows and columns according to the tile structure. This transformation is useful when the loops iterates over large values that exceed the size of the cache, allowing to control the behaviour of the caches during the loop execution.

Loop Unrolling This transformation technique *unwinds* the body of loops at the expends of a larger binary size. This transformation results in less loop overhead code (or no overhead code at all, in case of a complete unrolled loop). The example demostrates this.

```
/* Before */
for (i = 0; i < 100; i++)
        x[i] = y[i];


/* After */
for (i = 0; i < 100; i += 5)
        x[i] = y[i];
        x[i] = y[i+1];
        x[i] = y[i+2];
        x[i] = y[i+3];
        x[i] = y[i+4];
```

Function Inlining Function inlining (a very similar technique to loop unrolling) refers to replacing a function call with its body. This transformation has the advantage of reducing the execution time, memory accesses and power consumption of applications by avoiding run-time management of the call stack for the inlined function, at the expense of larger binary size and less readability. To inline a function, the programmer inserts either a keyword to instruct the compiler for function inlining, or by manually writing the body of the function.

Data Transformations The use of data layout transformations is another type of optimization that aims at changing the memory layout of data without affecting the order of instructions. The results obtained by using this transformation technique depends on the specific memory access patterns of the application as well as to how the system uses of the memory subsystem for storing the data.

When dealing with data tranformation a distinction is usually made between row-major, column-major or block-based data layout. In the next paragraphs these techniques are briefly described.

Row-major Data Layout This data layout places the elements of a row in consecutive memory locations. When this layout is used by compilers (for instance, a compiler for the C programming language), attention must be paid on how an array is accessed. Following is an example that exaplains this.

```
/* Before */
for (j = 0; j < 100; j++)
        for (i = 0; i < 100; i++)
                x[i][j] = x[i][j] + 1;
```

Provided that array x is stored in row-major format, the example would show poor spatial locality as the values of the outer loop (i.e., the row values) are not placed in consecutive memory locations. Changing the data layout to column-major format will solve the problem. Note that the same can be accomplished by using the earlier described *loop interchange* transformation.

Column-major Data Layout In programming languages where the default data layout is column-major (e.g. Fortran), the inverse of the described row-major data layout applies. In this case, the elements in a column are place consecutively in memory. Hence, the example showed previously would show a good spatial locality in a colum-major case.

Block Data Layout In case of multiple arrays that have a different accessing pattern, changing to row- or column major formats does not solve the problem. Furthermore, in case of a large matrix stored in memory, assuming, for instance, in a row-major layout, column elements accesses can cause cash conflicts. Instead of operating on entire rows or columns, a block data layout operates on submatrices or *blocks*, and each element in these blocks are placed on contiguous memory locations. The example below, from [21], shows this technique.

```
/* Before */
for (i = 0; i < N; i = i+1)
        for (j = 0; j < N; j = j+1)
        {
                r = 0;
                for (k = 0; k < N; k = k + 1)
                        r = r + y[i][k]*z[k][j];
                x[i] [j] = r;
};
```

In this example, the cache miss frequency depends on N and the size of the cache. In case the cache cannot hold the three N x N matrices, the code can be changed to operate on submatrices of B x B. This is showed below.

```
/* After */
for (jj = 0; jj < N; jj = jj+B)
        for (kk = 0; kk < N; kk = kk+B)
                for (i = 0; i < N; i = i+1)
                        for (j = jj; j < min(jj+B,N); j = j+1)
                        {
                                r = 0;
                                for (k = kk; k < min(kk+B,N); k = k + 1)
                                        r = r + y[i][k]*z[k][j];
                                x [ i ] [ j ] = x [ i ] [ j ] + r;
                        };
```

Morton Data Layout Morton data layout has a similar concept to block data layout. This technique divides the original matrix into four quadrants and lays out these submatrices contiguously in memory. Each of these submatrices is further recursively divided and laid out is the same way. At the end of recursion, elements of the submatrix are stored contiguously. Because of the similarities with the arrangement of blocks in block data layout, Morton data layout can be considered as a variant thereof. The only difference is in the order of the blocks.

2.3.3 Concluding Remarks

Due to the ever increasing demand for high performance systems, techniques for speeding up the execution of applications on both conventional and embedded systems have been researched for decades. This research is still ongoing, as the complexity of applications continue to grow. Especially for data dominated applications, like multimedia and telecommunication algorithms, finding a method for speeding up the executions of these applications is a very centered topic in the research community. The main bottleneck of these applications is the huge amount of transfers requirements from and to (on-chip and off-chip) memory that results performance degradation and in extreme energy consumption.

Furthermore, processing an application on a heterogeneous architecture containing reconfigurable devices results in a different data processing flow compared to traditional architectures. There is also an additional traffic routing compared to the communication traffic found in conventional architectures. Thus, memory subsystem should preserve the inherent advantages of the reconfigurable architectures that are primarily designed for data intensive computations. Therefore, it is important to provide the required *data bandwidth* to these devices, supporting respective parallelization and run-time adjustability of the memory access according to the needs of the application in question. In this scope, it is a prerequisite to *optimize the memory accesses* in order to provide the required data-flow for the most data-hungry computational units.

2.4 Motivation for Improved Memory Analysis

The techniques discussed above, both hardware- and software-based, do improve the communication bottleneck only to a certain degree. As the complexity of applications increases in the near future, their memory access patterns will evidently also be more difficult to predict. Hence, a more in-depth analysis of memory access patterns is required, which would help developers gaining a more detailed insight into how to tackle the well-known *memory-wall* problem. Moreover, with the advent of heterogeneous computing systems containing reconfigurable hardware, the need for utility tools that facilitate the application development process, tuning and optimization become of invaluable importance. Furthermore, these tools should also be able to perform an thorough analysis of the memory access behaviour of data-intensive applications, which, as already discussed, is unmissable. Also, detailed information on the memory access behaviour *inside* a function can deliver important information for optimization purposes on a fine grain scale and for discovering possible unusual behaviour of data objects used in such a function. Performing an analysis only on a function grain would hide the internal memory access behaviour of the function.

2.5 Hardware/Software Co-design

The ongoing complexity of embedded systems, due to the increasing size of integrated circuits, increasing software complexity and decreasing time-to-market requirements and product costs, motivates the need for a new design paradigm. This paradigm should take both the hardware and software aspects of the design cycle into a high-level design methodology, having the goal to shorten the time-to-market while reducing the effort and costs of the design.

With the advent of heterogeneous reconfigurable computing, the development of an application for such a *hybrid system* is typically started with high-level code (like C) that specifies the application. This application is then analysed (profiled) to see which parts of the application (or parts of a task) present a bottleneck. Once discovered the part of the code that results in a (possible) bottleneck, this part(s) are then implemented in hardware. Such a joint design is naturally called *Hardware/Software (HW/SW) Co-design*. The methodology aims at meeting system-level objectives by exploiting the synergism of hardware and software through their concurrent design[12], while considering their dependencies and overall system performance.

The two primary fields in which HW/SW Co-design has focussed are on alleviating the process of *exploring the design space* and on *partitioning* of an application onto a heterogeneous platform. In the next two sections these two topics are introduced.

2.5.1 Design Space Exploration

An important step in any development process is Design Space Exploration (DSE). In the context of heterogeneous reconfigurable systems design, it is essential to perform a DSE. During this phase early decision such as partitioning an application over different computational components, like CPUs, FPGAs, ASICs, or DSPs are made. Exhaustive DSE is necessary in order to identify the various design parameters such as execution time, memory, bandwidth, etc. This process is called Hardware/Software Partitioning, and requires the evaluation of the cost of implementing a task on each components. This process iteratively evaluate a very large design space, therefore a fast cost evaluation is essential. DWB uses code profiling to predict hardware characteristics and identify hot-spots to speed up this process. Estimates for hardware resource consumption, for example, can be used to omit functions that are too large to fit on an FPGA, or too small to exploit any degree of parallelism. DSE uses prediction models for its analysis. Early and fast estimation is essential for any DSE, for both minimizing the overhead of space exploration and for the process of partitioning due to their highly iterative nature.

2.5.2 Hardware/Software Partitioning and Mapping

Hardware/Software partitioning is the process of dividing an application between a processor component (in software) and one or more custom co-processor components (in hardware) to achieve an implementation that best satisfies requirements of performance, size, designer effort, and other metrics. A custom co-processor is a processing circuit that is customized to execute critical application computations far faster than if those computations had been executed on a general-purpose processor. FPGA technology encourages hardware/software partitioning by simplifying the job of implementing custom co-processors, which can be done just by downloading bits onto an FPGA rather than by manufacturing a new integrated circuit or by wiring a printed-circuit board. In fact, new FPGAs even support integration of processors within an FPGA itself, either as separate physical components alongside the FPGA fabric (hard-core processors) or as circuits mapped onto the FPGA fabric just like any other circuit (soft-core processors). High-end computers have also begun integrating microprocessors and FPGAs on boards[8], allowing application designers to make use of both resources when implementing applications.

Hardware/software partitioning is a difficult task, due to the large number of possible partitions. In its simplest form, hardware/software partitioning considers an application as comprising a set of regions and maps each region to either software or hardware such that some cost criteria (e.g., performance) is optimized while some constraints (e.g., size) are satisfied. Partitioning considers two application categories: sequential programs, where an application is a program written in a sequential programming language such as C, C++, or Java and where partitioning maps critical functions and/or loops to co-processors, and parallel programs, where an application is a set of concurrently executing tasks and where partitioning maps some of those tasks to co-processors. Doing this partitioning manually is a hard job, and while this still happens widely, tools to automate this process are intensively researched and developed. The Delft Workbench and the hArtes toolchain are (semi) automatic tools that accomplish this, and will be described later in this chapter.

2.6 Delft Workbench

The Delft Workbench[7] is a semi-automatic tool platform for integrated hardware/-software co-design targeting heterogeneous computing systems containing reconfigurable components. It is based on the Molen Programming Paradigm[41], and it address the entire design cycle (holistic process) from profiling and partitioning to synthesis and compilation of the application. This toolset is designed and developed with a view of actual hardware/software co-execution on a real hardware platform. Hence, the generated designs respect the physically available memory bandwidth and interface specification of the MOLEN polymorphic processor prototype. The design flow of the Delft Workbench is shown in Figure 2.2.

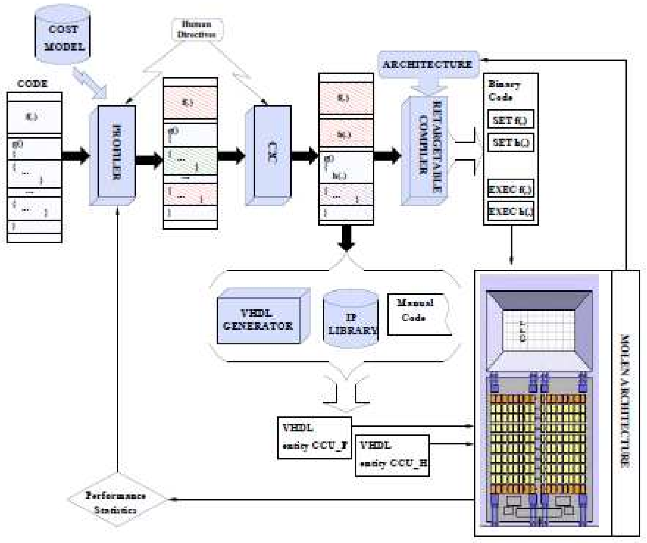


Figure 2.2: The DWB Architecture

The Delft Workbench focuses on four main steps within the entire heterogeneous system design, namely:

- *Code Profiling and Cost Modeling*[25].
- *Graph Transformations and Optimization*[31, 28, 16].
- *Retargetable Compiler*[32].
- *VHDL Generation*[39].

As a starting point, during *Code Profiling and Cost Modelling* in the context of DWB an application written in a high-level language will be profiled so to identify the parts of the application that are good candidates to be mapped on the reconfigurable hardware. This decision is based on estimating the speed-up that would be achieved if a certain part of the application would be implemented in hardware and on estimating the hardware availability itself, given the limited hardware resources. Hence, the goal is to *partition* the application so that some parts of the application will be accelerated in hardware while maintaining the other parts on a conventional processor. 'Part' can refer to an entire function as well as to part of a function. Multiple parts from different functions can be combined together as well and implemented in hardware. Two different profiling options are available in DWB, viz. static and dynamic. The *static* part of the profiling stage in DWB is involved with finding at an early stage hardware cost and performance estimates[25], i.e. it supports the developer in deciding if a certain task will fit into the reconfigurable device and the expected speed-up that will be achieved. Because this profiling step is able to make estimations from high-level languages like C, it can drive early HW/SW partitioning, but it also helps designers in re-factoring code, estimating project feasability, or driving optimization. The *dynamic* profiling path focusses on the run-time behaviour of applications, and, therefore, require a longer profiling time that the static profiling. The dynamic profiling stage uses the *gprof*[17] general profiler to identify hot-spots and frequently executed functions. In section 2.8, a motivation will be done on why a general profiler like *gprof* is not employable for discovering *memory-related bottlenecks*.

During the *Graph Transformation and Optimization*, the candidate parts of the application for hardware implementation are analysed to find out if the proposed code segments can be clustered, when sharing common characteristics. These clusters, if selected, will be implemented as new instructions on the reconfigurable instruction set[3]. This phase is the *graph restructuring* step. Next, an *optimization* step is performed to spot parallelization opportunities. Once the kernels have been clustered and formatted into new instructions into the instruction set, these instructions are further investigated to spot parallelism. As loops represent an important opportunity for parallelism, loop transformation techniques are applied to loops containing the new generated reconfigurable instructions. During this step it is investigated if the entire loop can be implemented in hardware, according to the available hardware resources and latency or if

[3]Unlike general-purpose processors, which have a fixed instruction set, reconfigurable devices have a flexible, extendible and application specific instruction set which can even change at run-time

the loop can be only partially implemented on the reconfigurable hardware. Particular attention must be paid when partitioning a certain application part into hardware, as this can affect the entire cycle time. Cycle time can slow down as the code segment executed on the reconfigurable device grows in size, resulting in an overall slower execution. Afterwards, the design flow forks into two parts: a retargetable compiler and a VHDL generation.

After making the decision of which parts of the code segment to implement in hardware, this code needs to be removed from the original source code and replaced by the appropriate reconfigurable hardware call. The *Retargetable Compiler* then generates the new object code which contains the call to the reconfigurable hardware for the newly identified instructions. It is a task of the compiler to decide on scheduling and mapping of these instructions for the execution on the reconfigurable hardware. This process is denoted as spatio-temporal compilation.

Where the first fork concerned the compiler scheduling of the SET and EXECUTE instruction, the second fork, i.e. *VHDL generation* phase, generates hardware description of the kernels. First, the hardware description logic of the kernels is searched inside an *IP-cores* library which, if available, can be directly instantiated. If the IP-cores library cannot provide this implementation, there are two possibilities: *automatic code generation* or *manual code generation*. The automatic code generation is performed by the DWARV toolset[47] and is envisioned for fast prototyping and fast development. The toolset consists of two module: the Data Flow Graph (DFG) Builder and the VHDL Generator. The input to the toolset is pragma annotated C code, which specifies the code segments to be implemented in hardware. Figure 2.3 depicts this toolset.

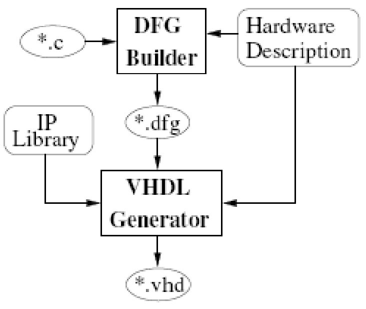


Figure 2.3: The DWARV Toolset

As a first step, the DFG Builder processes the input code. This module performs high-level hardware-independent optimizations on the code and transforms it into an intermediate representation (IR), suitable for hardware mapping. The IR is a hierarchical data-flow graph (HDFG), which is further processed by the VHDL generator. This tool performs low-level hardware-dependent optimizations and generates the final VHDL code. Nevertheless, hand-crafted hardware logic is the preferred method when very high quality results and performance are required.

2.7 hArtes

The hArtes project[6] addresses research and development issues of embedded systems, aiming to lay the foundation of a new *holistic* approach for embedded systems design, providing a tool-chain which accepts applications coded in a multiplicity of high-level algorithm descriptions and produces semi automatically a best fit of such an application into a heterogeneous reconfigurable embedded system. The hArtes project is closely related to DWB, as it targets the same heterogeneous systems. However, it also takes into account digital signal processing hardware and provide its own heterogeneous platform. Figure 2.4 illustrates the overall hArtes toolchain flow.

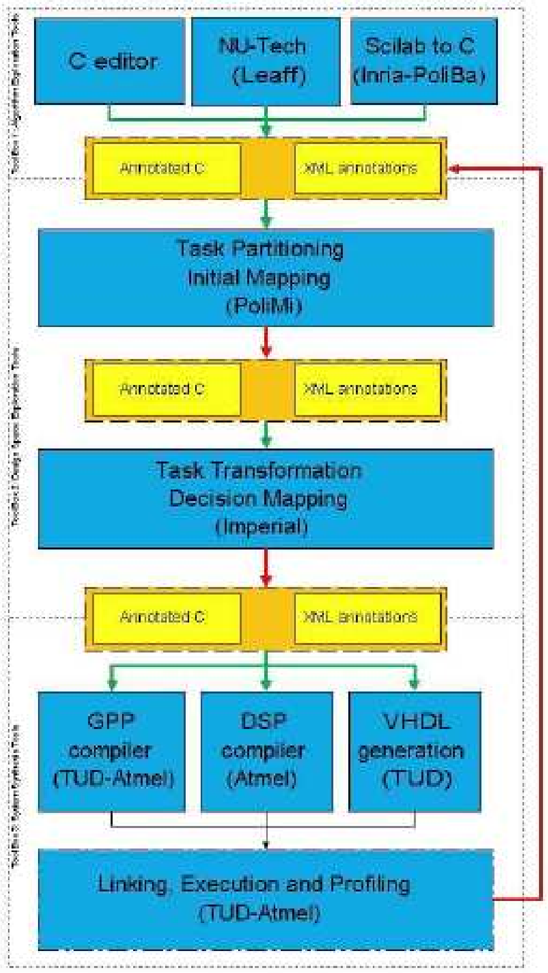


Figure 2.4: The hArtes Toolchain

The hArtes tool-chain is composed of the following three toolboxes:

1. algorithm exploration and translation
2. design space exploration (DSE)

3. system synthesis (SysSyn)

The input of this tool-chain is a high level application algorithm, described in one of several supported formats and languages, such as, graphical description, SciLab or Matlab code, or hand-crafted C. The internal representation of the application algorithms is C code, annotated with pragmas by the tools in the toolchain. The objectives of each hArtes Tool-Box can be summarized as follows:

- The *Algorithm exploration and translation Toolbox* provides tools to assist the designers to instrument and possibly improve the high-level input algorithm, and to translate this algorithm into a unified internal description in C.

- The *Design space exploration Toolbox* provides an optimal HW/SW partitioning of the input algorithm for each reconfigurable heterogeneous system considered. The input of the DSE Toolbox is the C description of the application algorithm, annotated with specification directives from the algorithm exploration and translation toolbox, and models of the reconfigurable heterogeneous system platform.

- The *System synthesis Toolbox* processes the optimized partitioning of the application provided by the previous Toolbox and provides files required to map the application algorithm on the components of the considered reconfigurable heterogeneous system with respect to its partitioning, i.e. program executables, configuration bitstreams, memory images, etc.

The profiling step inside hArtes is similar to DWB, with the only exception that it includes the *gcov*[2] tool, a code coverage tool that provides information such as how often each line of code executes. Section 2.8 will motivate why this stage in hArtes, as it is the case with DWB, needs additional tools to perform an analysis that take into account the problem of *memory-related bottlenecks*.

The hArtes tool-chain, as the Delft WorkBench toolchain, targets the Molen machine organization and the Molen programming paradigm. In fact, the Molen co-processors are not limited to be only reconfigurable implementations, they can actually be various types of augmenting hardware units. For example, in the context of hArtes, a digital signal processor (DSP) and reconfigurable hardware units are considered as Molen co-processors identically.

2.8 Motivation for Dynamic Memory Analysis Tools

Both DWB and hArtes prune their design space by performing a profiling step of the application during their development process. Run-time profiling information for DWB and hArtes is collected using general profiling tools, which analyze the application at functional level and provide application statistics like execution time of functions, helping identifying application hot-spots. However, these general profilers provide only a cumulative execution time, without distinguishing on the time spent for computation and memory access time. Hence, these kind of profilers cannot be employed to discover potential memory related bottlenecks.

At Delft University of Technology, an ongoing research tackles this problem by developing a set of tools that aim at providing detailed information about the memory access behaviour of applications.

The *Quantitative Usage Analysis of Data* (QUAD)[30] tool, is a sophisticated memory access pattern analyzer with the primary goal of *detection actual data communication dependencies* between functions. This tool is supposed to be an *integral part of the profiling step* in the DWB and hArtes development process, along with the mentioned gprof profiler. Besides QUAD, the *tQUAD*[29] tool is developed that aims at providing *detailed timing information* of a single function as well as its *memory bandwidth usage* during an application execution. Both tools will be described in more details in Chapter 4.

The main contribution to this thesis is the development of the xQUAD tool. This tools augments the memory analysis of the previous two tools by delivering even more detailed information. xQUAD provides information about the *general memory usage* of an application, in both flat file format as with the help of a visualization feature. Furthermore, it provides *detailed memory usage statistics in both functions level as at variables level.* This tool will be described thoroughly in Chapter 5.

2.9 Conclusion

In the last decades, the rate of improvements of processors' performances has greatly exceeded the rate of improvement of memory performance. This *gap*, which is likely to increase in the near future, is the main obstacle in improving the overall performance of applications running on both embedded and conventional systems. Conversely, applications show an increasing need in computational power. The advent of *heterogeneous architectures* incorporating *reconfigurable devices* has demonstrated to greatly accelerate the computational aspects of these applications. However, heterogeneous reconfigurable systems as they are, do not solve the *memory-wall* problem. In fact, this problem is even more evident with the introduction of these alternative architectures, as there are more computational processing units that need to communicate with the memory subsystem. If not dealt accordingly, this *communication bottleneck* will degrade most, if not all, of the gained computational performances. Therefore, the need of tools that help the development of the heterogeneous systems and, additionally, give detailed information about the *memory access patterns* of the application are of invaluable importance.

The ongoing research at Delft University of Technology tries to tackle this problem by the developement of sophisticated memory analysis tools which are able to provide detailed memory informations about applications. The QUAD and tQUAD tools will be described in Chapter 4. The xQUAD tool is developed during this thesis project and represent therefore the main contribution to this thesis. Hence it is described in greater detailes in Chapter **??**.

Analysis Techniques 3

As both hardware and software systems grow increasingly complex, tools that help developers analyse and improve programs behaviour are invaluable. These tools implement some kind of *program analysis* to retrieve information about applications. For the purpose of building the memory analysis tools introduced in Chapter 2 (which will be described in Chapter 4 and 5), an *analysis framework* is used, which abstracts the intricacies of the underlying hardware. Various types of *analysis frameworks* exists, which will be introduced in the following chapter. Nevertheless, more emphasis is put on the Pin[22] Dynamic Binary Instrumentation (DBI) framework (see 3.3) that has been chosen for building a set of memory analysis tools.

3.1 Program Analysis

Program analyses can be categorized into two types, according to when the analysis is performed: *static analysis* and *dynamic analysis*. Static analysis techniques do not rely on the execution of the application on real hardware. They rather analyse the source code, or some form of object code, to decide the set of possible execution paths and to obtain an upper bound on the execution in a specified hardware model. This technique is used especially when there is the need to guarantee a termination of the execution of a task, i.e. estimating the Worst Case Execution Time (WCET) of an application[44]. Dynamic techniques instead, analyze a program while it executes. These techniques aim at providing precise information about programs at execution-time. Dynamic analysis tools *instrument* the program for collecting program information, i.e. they insert extra code into the program. This process can be performed at various stages either in the source code, at compile-time, at post-link time, or at run-time.

There are basically two instrumentation approaches, viz.

- *Source instrumentation* involves instrumenting the source code of programs.
- *Binary instrumentation* involves instrumenting executables directly.

The two instrumentation approaches can be further categorized into *static* and *dynamic* techniques. *Source Instrumentation* is a technique that augments directly the source code of the application. The advantage of this approach is that the user can access all language-level information allowed by the programming language of the application. For instance, a developer can augment the source code ensuring that a certain range of functions output determined values to a report/log file. This method brings in a major drawback, namely the burden of work that the programmer has to do. Furthermore, in case only the executable is available, this approach cannot be employed, as the developer has no access to the source code of the application.

As mentioned, binary instrumentation can be divided into static and dynamic. *Static Binary Instrumentation* (SBI) frameworks can be considered the precursors of many *Dynamic Binary Instrumentation* (DBI) frameworks, and many DBI frameworks have models that are similar to SBI frameworks. Analysis based on static instrumentation has many limitations compared to its dynamic variant. As SBIs instrument the binary before run-time, the possibility exists that code and data of an executable can get mixed, and a static tool may not be able to distinguish the two. In this thesis, emphasis is put on *Dynamic Binary Instrumentation*, as this will be the employed technique for the memory analysis tools described in Chapter 4 and 5.

3.2 Dynamic Binary Instrumentation

Dynamic Binary Instrumentation (DBI), in contrary to SBI, occurs at run-time. While the application is running, the analysis code is injected into the executable. The majority of DBI systems employ a *Just-In-Time* (JIT) compiler for inserting instrumentation code into the dynamically generated code from the application. DBI is advantageous as it permits to, among others, discover code at run-time, attach to running processes, running the executable unaltered (i.e. there is no need to re-compile or re-link), and to analyse all the application code. For these reasons, Dynamic Binary Instrumentation frameworks are gaining popularity as means of constructing analysis tools (e.g. profilers), which are categorized under Dynamic Binary Analysis (DBA) tools. However, DBA tools suffer from performance slow-down, as the cost of instrumentation is incurred at run-time. Basically, the overhead of a DBI based tool can be divided into two stages: the overhead for the *instrumentation routines* and the overhead caused by the user-defined *analysis routines.*

Developing a DBI framework from scratch for a *specific* DBA tool is a cumbersome job to achieve. Therefore, over the last decade generic DBI frameworks have been developed to mitigate this problem and to speed up the process of DBA development. This is facilitated by the fact that instrumenting an application is the same for all DBAs, independently of what the precise purpose of the specific DBA tool is. Furthermore, most used DBI (see for instance [22, 27, 9]) have similar internal engines. Basically, a DBI framework intercepts a *block* of code (or code fragment, as it is sometime referred to) and inserts instrumentation code before execution (Just-In-Time). This modified code is stored in a code cache that preserves frequently executed blocks of code for future use. When the execution of the instrumented block is finished, another block is intercepted and instrumented. Instrumentation can occur in different levels of granularity (instruction, basis block, routine, etc.), providing the developer with a broad choice for developing DBA tools, according to their needs. Instrumenting at a certain level of granularity generally implies monitoring everything at that level of granularity, which explains the high slow-down penalty that DBA tools have to pay.

There are two approaches to dynamic instrumentation: *probe-based* and *jit-based.* Probe-based instrumentation works by dynamically replacing instructions in the original program with branches (probes) that point to the instrumentation code. This method presents several drawbacks compared to jit-based instrumentation. First, instrumenting via a probe-based method is *not transparent* because the application's instructions are modified by the branches pointing to the instrumentation code. Further, on architectures with variable instructions size, replacing an instruction with a longer branch instruction may cause to overwrite the subsequent instructions. Finally, branching to instrumentation code may be implemented in multiple levels. This means that a branch may in turn branch to another instruction, etc., resulting in a significant performance overhead. This drawbacks make these systems not suitable for pervasive fine-grained instrumentation (where every executed instruction is instrumented). In contrast, jit-based instrumentation is mode suitable for fine-grained instrumentation, as it works by dynamically compiling the binary and can insert instrumentation code anywhere in the binary.

DBI frameworks basically use two fundamental ways for representing and instrumenting code. One method is *disassemble-and-resynthesize*, which converts machine code to an intermediate representation (IR), adds instrumentation to this IR, and then converts back the IR into machine code. The original instructions from the client application are discarded and the final code is generated exclusively from the IR. Valgrind[27] uses this methods.

Another method is *copy-and-annotate* (used by, e.g., Pin[22] and DynamoRIO[9]), where each instruction is copied as 'it is' and annotated with a description of its effects. Exceptions are made for control flow instructions that need to be changed. Tools use the annotations to guide their instrumentation.

Following, a brief description into some widely used and maintained DBI framework is given, according to recent literature.

3.2.1 DynamoRIO

DynamoRIO [9] is a Dynamic Binary Optimization and Instrumentation framework operating on unmodified native binaries and implemented for both IA-32 Windows and Linux. DynamoRIO is capable of running large desktop applications. The goal of DynamoRIO is to observe and potentially manipulate every single application instruction prior to its execution. To achieve this, DynamoRIO caches translations of frequently executed *basic blocks* code so they can be directly executed in the future. It works on almost unmodified x86 code - only control-flow instructions are modified.

3.2.2 Valgrind

Valgrind [27] is a DBI framework for building heavyweight DBA tools. In fact, Valgrind is designed with particular attention to heavyweight tools, with the unique support for *shadow values* tools. These kind of tools involve large amounts of analysis data that is accessed and updated in irregular interval patterns. Example is the shadowing of every register and memory value with a meta-value. Therefore, lightweight tools built with Valgrind run comparatively slower than when these tools are built with other DBI frameworks. However, Valgrind's developers aim at the fact that Valgrind can be used for more interesting, robust, heavyweight tools that are difficult or impossible to build with other DBI frameworks such as Pin and DynamoRIO [27].

Valgrind is available under the GNU General Public License (GPL), and runs on x86 and AMD64 with Linux, PPC 32/64 with Linux/AIX.

DBA tools are constructed as Valgrind plug-ins, forming a basic view of

```
Valgrind core + tool plug−in = Valgrind tool
```

The execution of a Valgrind DBA tool is as follows: at start-up, the Valgrind's core, the tool plug-in and the client program are loaded into the same address space; the tool then recompiles the machine code of the client one basic block at a time, in a just-in-time, execution-driven fashion; the Valgrind framework disassembles the basic block code into an intermediate representation (IR) which is instrumented by the tool plug-in, and then converted back into machine code. The resulting *translated* code is stored in a code cache to be rerun as necessary.

An important thing to know about Valgrind is that a tool cannot use anything from the C library. Valgrind provides an implementation of a reasonable subset of the C library.

3.2.3 pin

Pin[22] is a DBI that emphasizes ease-of-use, portability, transparency, efficiency, and robustness. Pin uses dynamic compilation for binary instrumentation and code caching for code reusability. Pin is the DBI framework used for the development of the QUAD toolset (see Chapters 4 and 5). Hence it is explained in more detailed in the next section.

3.3 Pin

Pin provides a portable and efficient DBI platform for building a variety of DBA tools (also called Pintools in Pin's jargon) that works with unmodified Linux, Windows and MacOS binaries on multiple architectures, namely IA32, 64bit x86, Itanium®, and ARM[1] architectures. Furthermore it instruments multi-threaded applications. This framework abstracts away from the details of the target architecture (whenever possible), allowing the developer to concentrate on the development of the tools without having to be aware of the intricacies of the underlying system. Pin preserves the original application behaviour by providing instrumentation *transparency*. The application uses the same addresses (both instruction and data) and the same values (both register and memory) as it would in an un-instrumented execution. Furthermore, Pin does not modify the application stack, as some application may intentionally reference to memory addresses beyond the top of the stack.

Pin is designed to be easy-to-use. The user writes instrumentation and analysis routines. Instrumentation routines determine *where* to place calls to analysis routines, e.g. before an instruction; analysis routines define *what* to do when instrumentation is activated. Analysis routines are customizable by the user and are called while the program executes. The arguments to analysis routines can be, among others, the instruction pointer, effective memory address of the instruction, memory or stack value, address of a

[1]The development for ARM stopped years ago, which made the instrumentation framework incompatible with the latest version of GCC

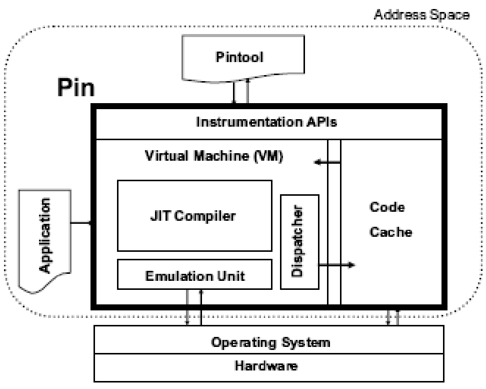


Figure 3.1: Pin's Architecture

branch instruction, system calls values. Instrumentation is performed by a just-in-time (JIT) compiler.

Instrumentation with Pin can be done at different levels of granularity. The finest level of granularity is instrumentation at instruction level, i.e. instrumenting the application one instruction at a time. Further, it is possible to instrument code at *trace level*[2], at *procedure level*, and at an entire *image level*.

Figure 3.1 illustrates Pin's software architecture. At the highest level, Pin consists of a virtual machine (VM), a code cache, and an instrumentation API invoked by Pintools. The VM consists of a JIT compiler, an emulator, and a dispatcher. The input to the JIT compiler is not bytecode, however, but a native executable. Furthermore, Figure 3.1 shows that Pin, Pintool and the application are all present in the same address space. While they share the same address space, they do not share any libraries. Making these libraries private avoids undesired interaction between these three components.

Pin intercepts the execution of the first instruction of the application and re-compiles the executable generating *basic blocks* code starting at this instruction, and instrumenting the code according to the specified instrumentation type. This *trace* is almost identical to the original one, except that it runs under the control of Pin, which means that Pin ensures that it regains control when a branch exits the sequence. After Pin gains control of the application, the virtual machine coordinates its components to execute the application. The JIT is responsible for compiling and instrumenting the application, while the dispatcher launches the application modified by the JIT compiler. In case that a branch exits an earlier generated basic block, the JIT fetches a new code block, which is instrumented by the Pintool (if required) and the cycle is repeated. To improve performance, the generated code and its instrumentation are saved in a code cache for future execution of the same sequence of instructions to improve performance.

[2]A trace is defined as a straight-line sequence of instructions executed sequentially. Pin guarantees that traces only enter at the top, but may have multiple exits.

3.3.1 Evaluation of Pin

Running an instrumented application usually shows a considerable slowdown. This depends on the nature of the instrumented application, as on the overhead caused by the analysis routines in the Pintool. It appears that most of the slowdown is caused on execution of the code, rather than code compilation (which includes inserting the instrumentation code). Hence some performance improvements are done during the compilation phase of the application. Optimization techniques are register reallocation, inlining, liveness analysis, and instruction scheduling. This results in instrumented code running very fast, compared to other DBIs.

In [22] an evaluation of Pin's performance is shown. Two cases are taken into consideration: Pin performance with and without instrumentation code. Without instrumentation, the overhead of Pin on some typical benchmarks is dependent on the amount of code reusability. It is shown that for applications that have a relatively short execution time, and thus insufficient opportunity for code reuse, Pin pays a high cost in re-allocating registers compared to other tool that do not re-allocate registers. Nevertheless, register re-allocation is important as it provides Pin and Pintools more virtual registers than the number of physical registers. In case of enough code reusability, it can be seen that most of the time spent in Pin's execution is in the code cache. The effects of register re-allocation and indirect linking become apparent when evaluating the performance on the ARM architecture, where these two optimization are not yet implemented on ARM. This causes the VM to resolve all indirect control transfers, making the execution time spent for register re-allocation dominant on ARM.

Evaluation of Pin with instrumentation is done using a standard basic-block counting tool, which outputs the execution count of every basic block in the application. The evaluation of this tool without any optimization and with optimization techniques like inlining, liveness analysis and scheduling shows an average slowdown improvement from 10.4x to 2.5x for integer and from 3.9x o 1.4x for floating point[3].

3.3.2 Comparison of Pin against Other DBIs

Furthermore, Pin's performance is compared against DynamoRIO and Valgrind using the standard basic-block counting tool. Withouth instrumentation, the performed analysis shows that both Pin and DynamoRIO outperform Valgrind significantly. Overall, DynamoRIO is faster than Pin, as this framework was primarily designed for *optimization* (furthermore, DynamoRIO does not perform register re-allocation). When considering the performance slowdown with instrumentation, is becomes apparent that Pin significantly outperforms both DynamoRIO and Valgrind: on average Valgrind slows the application down by 8.3x, DynamoRIO by 5.1x, and Pin by 2.5x. In contrast to the other tools, Pin automatically inlines the calls to the block entries and performs liveness analysis, demonstrating a main advantage of Pin: it provides efficient instrumentation without shifting the burden to the Pintool writer. Regarding Valgrind, [27] confirms the above mentioned comparisons. However, the Valgrind's developers specify that the basic-block counting tool, a so called *lightweight* tool, is exactly the kind of tool Val-

[3]The higher overhead on the integer is due to the many more indirect branches and returns

grind is not targeted at, and therefore will never be as fast as Pin or DynamoRIO[27]. Furthermore, it is stressed that Valgrind's targets are *heavyweight* tools, which should take advantage of the characteristics of Valgrind for building robust tools relatively easy, providing powerful instrumentation capabilities and allowing reasonable performance.

3.3.3 Pro et Contra of Pin

Pin's call-based model of instrumentation is simpler than other tools where the user inserts instrumentation by adding and deleting statements in an intermediate language. However, it is equally powerful in its ability to observe architectural state and it free the user from the need to understand the idiosyncrasies of an instruction set or learn an intermediate language. The inserted code may overwrite scratch registers or condition codes; Pin automatically saves and restores states around calls so these side effects do not alter the original application behaviour. The Pin model makes it possible to write efficient and architecture-independent instrumentation tools, regardless of whether the instruction set is RISC, CISC, or VLIW.

As mentioned before, Pin implements register allocation, inlining, liveness analysis, and instruction scheduling to optimize jitted code. This fully automated approach distinguishes Pin from other DBIs which requires the user's assistance to boost performance. For example, Valgrind [27] need user intervention in order to perform inlining; similarly, DynamiRIO [9] requires the tool developer to manually inline and save/restore application's registers. Another feature that distinguishes Pin is the ability to attach and detach to a process. Hence it is possible to attach Pin to a process any time, perform the desired analysis and, eventually, detach. The application incurs instrumentation overhead only during the period that Pin is attached. This is especially useful when instrumenting large, long-running applications.

Pin is further an easy-to-use framework, which permits users to write tools in a more compact way when compared to other tools. For example, a simple tool in Pin for memory tracing requires 30 lines of code, while in Valgrind it requires 100 lines of code [27].

Pin does only provide limited access to symbol and debug information, which means that no function's API for retrieving variable names is available. This requires an alternative choice. For this purpose, DWARF is chosen, and will be explained in Chapter 5.2.2.

3.4 Pin Instrumentation API

As already mentioned, one of the advantages of Pin is its rich API. These API can be divided into *generic APIs* and *architecture-specific APIs*. Generic APIs are architecture independent and abstracts away the underlying instruction set idiosyncrasies and allows context information such as register contents to be passed to the injected code as parameters. Determining memory accesses (e.g. read/write) or control-flow changes (e.g. branch of call instructions) for example can be achieved with the basic APIs.

```
// This function is called when the application exits
VOID Fini(INT32 code, VOID *v)
{
  cerr << "End of instrumentation" << endl;
}

// Analysis Routine
static VOID RecordTrace(VOID *ip, VOID *addr)
{
  cerr << "IP and Effective Address of Memory Instruction are" << endl;
  cerr << "IP: " << ip << endl;
  cerr << "EA: " << addr << endl;
}

// Instrumentation routine
VOID Instruction(INS ins, VOID *v)
{
  if(INS_IsMemoryRead(ins))
    {
    INS_InsertPredicatedCall(ins,
        IPOINT_BEFORE,
        (AFUNPTR)Analysis,
        IARG_INST_PTR, //IP passed to analysis routine
        IARG_MEMORYREAD_EA, //Memory address passed to analysis routine
        IARG_UINT32,
        IARG_END);
    }
}

int main( int argc, char *argv[] )
{
  PIN_InitSymbols();
  if( PIN_Init(argc,argv) )  return Usage();

  INS_AddInstrumentFunction(Instruction,0);
  PIN_AddFiniFunction(Fini,0);
  // Start the program. Never returns
  PIN_StartProgram();
  return 0;
}
```

Figure 3.2: Pin example of instrumentation and analysis routines

Architecture-specific API are available in two kind: API for IA-32 and Intel®64[4] ISA and API for the IA-64 ISA (Itanium Processor Family).

Pin APIs are *call-based*, i.e. tools are build using *instrumentation routines* and *analysis routines*[5]. Instrumentation routines define where instrumentation is inserted, e.g. before memory instructions. Analysis routines specify what to do when the instrumentation is activated, e.g. increase a counter each time a specific memory address is encountered. Figure 3.2 demonstrates a minimal implementation of the Pin API to show an example of instrumentation and analysis routines.

Pin provides limited access to an application's *symbol table*. The only supported symbol object information with Pin is information about function symbols. Other types

[4]Intel began using this name for 64 bit processors since late 2006. Before, Intel referred to 64-bit processor as EM64T (Extended Memory 64 Technology)

[5]This notion is borrowed from ATOM

of symbols (e.g. data symbols) must be retrieved using an external method other than Pin. Nevertheless, to access function names through Pin, ***PIN_InitSymbols*** must be called.

To initialize the Pin framework, ***PIN_Init*** is called.

Pin permits to instrument an application on different granularities, i.e. instrumenting at *image* level, *trace* level, *routine* level, and at *instruction* level.

Image instrumentation permits to instrument an entire application image. The instrumentation API call is done with the ***IMG_AddInstrumentFunction***. A Pintool can analyse the *sections* inside an image, which in turn contains *routines* that the tool can inspect. Routines, in turn, consist of *instructions*, that are made available to the tool. The example below shows this nested analysis.

```
IMG_Name(SEC_Img(RTN_Sec(rtn))
```

Image instrumentation requires the Pin_InitSymbols call to be called before Pin_Init.

Trace instrumentation permits to instrument an application one trace at a time. Traces are a sequence of instructions, beginning at the target of a taken branch and ending with an unconditional control-flow changing instruction, including calls and returns. Pin makes sure that a trace is only entered at the top, but it may contain multiple exits. In case that a branch occurs in the middle of the trace, Pin reconstructs a new trace that begins at the branch targets, and end, again, with an unconditional branch. Pin breaks the trace into basic blocks, which is a sequence of code with a single entry and a single exit, terminating at a (conditional or unconditional) control-flow changing instruction. Instrumenting at trace level is achieved with a call to the ***TRACE_AddInstrumentFunction*** API call.

Making an instrumentation call to the ***RTN_AddInstrumentFunction*** API permits the user to analyse an application based on its functions (or routines). Pin finds routines by using the *symbol table* information. Pin_InitSymbols() must be called beforehand. RTN can be used at instrumentation and at analysis time. Symbol objects provide information about function symbols in the application. Hence, if information about other symbols is needed (e.g. data symbols), they have to be retrieved without the help of Pin.

Finally, Pin permits analysis based on instruction granularity, achieved with the ***INS_AddInstrumentFunction*** API. This lets the tool inspect one instruction at a time. It can be accessed only at instrumentation time.

Besides analyzing a program behaviour, Pin allows also to make changes to it. This may consists of adding or deleting instructions, changin register or memory values, and changing of the control flow.

3.5 Calling Pin

Running a program under the supervision of Pin is done via a few (terminal-based) commands, as the following example demonstrate.

```
pin −t pintool −− application
```

The commands above call the *pin* instrumentation engine with the *-t* parameter, which is used to call the *pintool*, i.e., the instrumentation tool. Afterwards, the – *application* parameter calls the application to be analysed.

3.6 Conclusion

As the complexity of applications grow, analyzing these applications becomes proportionally complex. To cope with this complexity, developers have start building *analysis frameworks*, which abstract away the hardware complexities, speeding up the development of new and reliable *analysis tools*. The work done in this project revolves around *Dynamic Binary Analysis*, which is accomplished by using a *Dynamic Binary Instrumentation* framework. Among the available frameworks, the Pin[22] framework is used, a portable, efficient, easy-to-use, and transparent DBI framework. In the next two chapters, this framework is used as the underlying platform for building three sophisticated memory access analysis tools. These *Dynamic Binary Analysis* tools provide a detailed and comprehensive analysis about applications' memory access behaviour.

4 QUAD: Sophisticated Memory Patterns Analysis Tools

The main focus of the work proposed in this thesis is on the profiling process inside the Delft Workbench and the hArtes toolchains. Profiling information about the memory usage of applications can deliver valuable insights when performing hardware/software partitioning of these applications. In this context different tools are developed, that provide the developer with a detailed analysis *memory access patterns* of applications.

The first described tool, QUAD, is a *memory access tracing* tool that provides a comprehensive quantitative analysis of memory access patters of an application with the primary goal of detecting *pure data dependencies* at function level.

The tQUAD tool is designed as a complementary profiler in the Delft Workbench dynamic profiling framework along with QUAD to deliver *detailed temporal memory bandwidth* usage information for each kernel in an application.

Finally, during this thesis research, the xQUAD tool is developed which augments the analysis generated by the previous two tools by providing both an high level *memory map representation* of application and detailed memory usage information inside a certain kernel, i.e., on a *variable level.* As this tool is an integral part of this thesis research, it is briefly introduced in this chapter, together with the other tools, for consistency reasons and it is described in more details in Chapter 5.

These tools are built upon the Pin[22] Dynamic Binary Instrumentation (DBI) framework and can be classified as a Dynamic Binary Analysis (DBA) tools, which analyse an application at the machine code level. More information about these techniques were provided in Chapter **??**. Furthermore, the tools described in this chapter are intended to be an integral part of the profiling stage of the Delft Workbench and hArtes toolchains (Chapter 2.6 and 2.7).

Following, the QUAD[30] and the tQUAD[29] tools are presented, describing their implementaion detailes as well as their usage. Also, the xQUAD tool is introduced, which will be described in more detail in Chapter 5.

4.1 QUAD - Quantitative Usage Analysis of Data

The QUAD [30] (Quantitative Usage Analysis of Data) tool is a sophisticated memory access tracing analyser that provides a comprehensive quantitative analysis of memory access patterns. The primary goal of this tool is to detect actual data communication dependencies between kernels.

This section describes QUAD's objectives, implementational details and usage.

4.1.1 QUAD Objective

In the previous chapters of this thesis, the need for tools that are able to provide *detailed memory behaviour information* of application became clear. Especially with the introduction of heterogenous archictectures incorporating a (or more) reconfigurable device(s), the need for these kind of tools is even more evident.

Traditionally, a general profiler like *gprof* [17] is employed for the detection of application hot-spots. However, these tools are not able to distinguish between *computation time* and *memory access time*, and cannot be used to discover potential bottlenecks.

Furthermore, most existing memory access analysis tools focus only on detecting memory bottlenecks, memory faults, bugs and leaks, without providing detailed information regarding the data-dependencies in application's memory usage [43, 10]. One of the early simple tools developed for understanding memory access patterns of Fortran programs is presented in [13]. Another tool similar to QUAD is Embla [15], which allows the user to discover data dependencies in sequential programs, thereby exposing opportunities for parallelization. Embla performs the analysis dynamically, and records dependencies at run-time as they arise. However, QUAD aims at discovering the *actual data dependency*[1], which is different from the conventional data dependency referred to in Embla and other similar tools. Data dependency is estimated in the sense of producer/consumer binding.

Even though QUAD can be employed to spot coarse-grained parallelism opportunities in an application, it practically provides a more general-purpose framework that can be utilized for optimizations of various reconfigurable systems, by estimating effective memory access related parameters, e.g. the amount of unique memory addresses used in data communication between two cooperating functions. QUAD can also be used to estimate how many memory references are executed locally compared to the amount of references that have to go to the main memory.

The work conducted with QUAD revolves mainly around (dynamic) profiling. Therefore, as QUAD is intended to be part of the DWB and hArtes toolchain analysis flow, the tool is integrated accordingly as depicted in figure 4.1. In section 2.6 and 2.7 more information about this can be found.

Following, the design and implementation of QUAD are discussed.

4.1.2 QUAD Design and Implementation

QUAD is built as a Dynamic Binary Analysis (DBA) tools using the Pin Dynamic Instrumentation (DBI) framework, described earlier in this chapter.

QUAD aims at discovering actual data dependence, which is estimated in the sense of producer/consumer binding. More precisely, QUAD reports which function is consuming the data produced by another function. The exact amount of data transfer and the number of Unique Memory Addresses (UMA) used in the transfer process are calculated. Based on the efficient Memory Access Tracing (MAT) module implemented in QUAD, which tracks every single access (read/write) to a memory location, a variety of statistics

[1]By definition, a data dependency is the situation in which a program segment (instruction, block, function, etc.) refers to data produced by a preceding segment. Actual data dependency arises when a function consumes data that is produced by another function earlier.

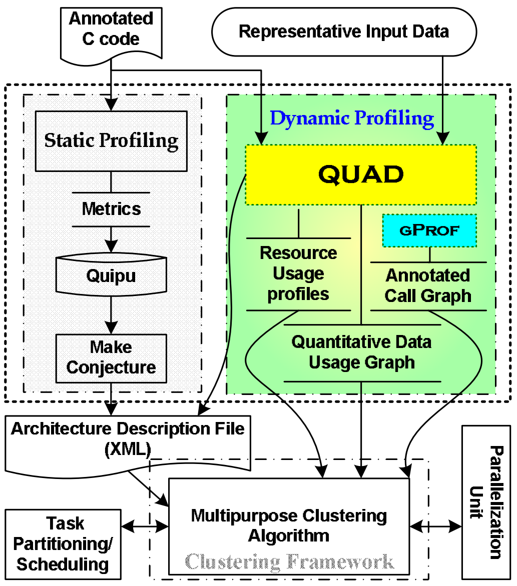


Figure 4.1: Profiling Framework of QUAD within DWB

related to the memory access behavior of an application can be measured, e.g. the ratio of local to global memory accesses in a particular function call.

Figure 4.2 illustrates the architectural overview of QUAD along with the components in Pin. At the highest level, there is a Virtual Machine (VM), a code cache, and an instrumentation API. Compare this with Figure 3.1 to see how QUAD fits along the Pin framework inside the same address space. The main component inside QUAD is the MAT module, which is responsible for building and maintaining dynamic trie data structures to provide relevant memory access information as fast as possible. The trie data structure acts as a shadow memory for each byte accessed within the address space of an application.

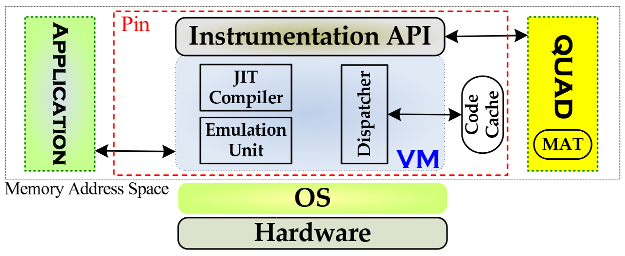


Figure 4.2: Architectural overview of QUAD

QUAD contains instrumentation and analysis routines implemented using the Pin API. This API calls allows, as explained in Section 3.3, to make an analysis based on different level of granularities. QUAD uses two API calls, *INS_AddInstrumentFunction()* for instruction level instrumentation and *RNT_AddInstrumentFunction()* for routine level instrumentation, which set calls to the instrumentation routines *Instruction()* and *UpdateCurrentFunctionName()*, respectively. In turn, these two instrumentation routines calls two main analysis routines, namely *RecordMemRef()* and *EnterFunc()* which are responsible for updating tracing information of memory references and maintaining an internal call graph respectively. Figure 4.3 gives an overview of the implementation of QUAD.

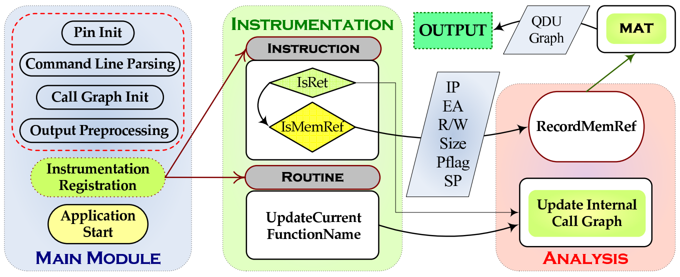


Figure 4.3: Implementation overview of QUAD

As shown in Figure 4.3, the *main module* consists of the initialization of the Pin DBI framework, command line parsing, call graph initialization, instrumentation functions registration and starting the application. Registration of the instrumentation function is done via the Pin API interface, as showed in Figure 4.2. QUAD registers for two types of instrumentation, viz. *Instruction* and *Routine* instrumentation, by calling the INS_AddInstrumentFunction() and RTN_AddInstrumentFunction() API functions, respectively. In turn, these two APIs set up calls to the *Instruction()* and *UpdateCurrent-FunctionName()* instrumentation routines. First, the UpdateCurrentFunctionName() routine is called, which implements an *analysis routine* responsible for maintaining an internal *call stack* representation. Subsequently, the Instruction() routine is called, which is reponsible for inspecting the application at *instruction granularity* for memory references instructions. Everytime a *memory instruction* is detected, the *RecordMem()* analysis routine is called. This analysis routine is responsible for identifying the function that is involved with the currently detected memory instruction, and to pass the required information to the Memory Access Tracing (MAT) module.

The MAT module is involved in detecting and extracting memory reference information during the execution of an application. This module utilizes a *trie* data structure for fast storage and retrieval of memory accesses[30].

4.1.3 How QUAD Works

To start QUAD, a similar manner is used as described in Section 3.5, namely

```
pin -t [QUAD-path] QUAD.so [QUAD-options] – [application-name]
[application-options]
```

on a Linux system and

```
pin -t [QUAD-path] QUAD.dll [QUAD-options] – [application-name]
[application-options]
```

on a Windows system[2].

The command-line options for the QUAD tool are:

- *-filter_uncommon_functions* - This option filters out uncommon function names which are unlikely to be defined by the user. Examples are function names beginning with underscore(s), question mark, etc. This is useful if the user wants to restrict the analysis only on functions that are actually part of the application that is analysed, excluding library function. The default value for this option is set to true. Hence, to see all the functions in the main image file the '-filter_uncommon_functions 0' option will allow this.
- *-include_external_images* - This option enables tracing of functions that are contained in external images file(s). By default, only the functions in the main image file are traced and reported. This option together with the '-filter_uncommon_functions' provides more flexibility to include/exclude required/unwanted functions in the report files. This option also has considerable impact on the reported quantitative bindings data and the corresponding producers/consumers.
- *-ignore_stack_access* - This option make it possible to exclude accesses to the local stack memory, providing a view of the data transferred via non-stack region.
- *-use_monitor_list [file name]* - This option allows users to make a report file based on some predefined functions, excluding all other (uninteresting) functions. The function names to monitor are specified in a text file, with each function declared on a single line.
- *-xmlfile* - This option permits to specify the name of the XML output file. By default this output file is named 'dek_arch.xml'.

4.1.4 QUAD's Example Outputs

The memory reference information gathered by QUAD during the execution of an application are reported in two separate formats. All the *producer/consumer* bindings information is stored in an XML file. This makes it easy for third-party application to import this data for further interpretation and processing. Figure 4.4 gives an example of a XML file generated with QUAD.

In the above figure, a BINDING is established when a function writes to a memory location which is later read by another function. The writing function is detoted with a PRODUCER tag, while the CONSUMER tag denotes the function reading the data. DATA_TRANSFER is the total amount of data being read by the CONSUMER, while the UMA tag shows the number of Unique Memory Addresses in the transferred data.

Furthermore, the binding information explained above will also be stored in .dot format. This allow developers to visualize the XML file, showing the actual bindings between function in a graphical way.

[2]To access symbolic information on Windows, Pin uses 'dbghelp.dll'. This DLL is not distributed with the kit, and must be get separately. Using 'dbghelp.dll' in an instrumented process is not safe and can cause deadlocks in some cases.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE ORGANIZATION SYSTEM "architecture.dtd">
<ORGANIZATION>
    <PROFILE>
        <QUAD>
            <BINDING>
                <PRODUCER>x264_validate_parameters</PRODUCER>
                <CONSUMER>x264_ratecontrol_new</CONSUMER>
                <DATA_TRANSFER>72</DATA_TRANSFER>
                <UMA>72</UMA>
            </BINDING>
            <BINDING>
                <PRODUCER>x264_validate_parameters</PRODUCER>
                <CONSUMER>x264_macroblock_cache_load</CONSUMER>
                <DATA_TRANSFER>28800</DATA_TRANSFER>
                <UMA>464</UMA>
            </BINDING>
        </QUAD>
    </PROFILE>
</ORGANIZATION>
```

Figure 4.4: XML format of producer/consumer binding

4.2 tQUAD

The tQUAD tool provides detailed *timing information* of a single kernel execution as well as its *memory bandwidth usage* during the execution of an application.

4.2.1 tQUAD Objectives

Similar to the reasons explained earlier for QUAD, tQUAD [29] is a tool that aims at facilitating the development of heterogeneous reconfigurable computing systems. In fact, by providing timing information and memory bandwidth usage of a kernel during its execution, tQUAD can address the issue of efficient *scheduling* and *mapping* of tasks onto heterogenous reconfigurable architectures.

tQUAD is, as QUAD, developed in the context of the DWB and hArtes toolchains, and is intended to be part of the *dynamic profiling* phases of these toolchains. Figure 4.5 positions tQUAD inside the DWB process. More information about DWB and hArtes is provided in Chapter 2.6 and 2.7.

As can be seen in Figure 4.5, tQUAD is designed as a complementary profiler in the DWB dynamic profiling framework along with QUAD. While the aim in QUAD is revealing the quantitative information about *data communication between kernels*, the purpose of tQUAD is to extract the *timing information* of a single kernel execution as well as its *memory bandwidth usage* during an application execution. These information can lead to the discovery of *long executing* kernels, and its data transfer. Subsequently, these information have an invaluable importance in design space exploration decisions of task scheduling and mapping process in heterogeneous reconfigurable systems. Furthermore, these information can be used by the application developer for optimizing the application code.

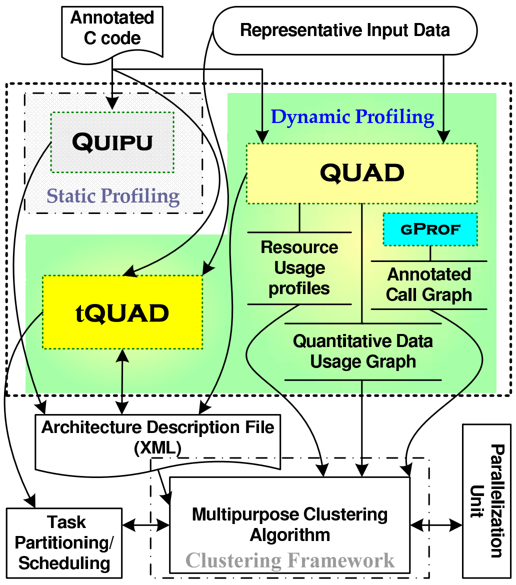

Annotated C code
Representative Input Data
Dynamic Profiling
QUIPU
Static Profiling
QUAD
GPROF
Resource Usage profiles
Annotated Call Graph
tQUAD
Quantitative Data Usage Graph
Architecture Description File (XML)
Parallelization Unit
Task Partitioning/ Scheduling
Multipurpose Clustering Algorithm
Clustering Framework

Figure 4.5: Profiling Framework of tQUAD within DWB

4.2.2 tQUAD Design and Implementation

The architectural overview of tQUAD is similar to that of QUAD, as can be seen in Figure 4.6. Same as QUAD, tQUAD implements calls to instrumentation APIs that permit the tool to hook itself at run-time with the Pin framework.

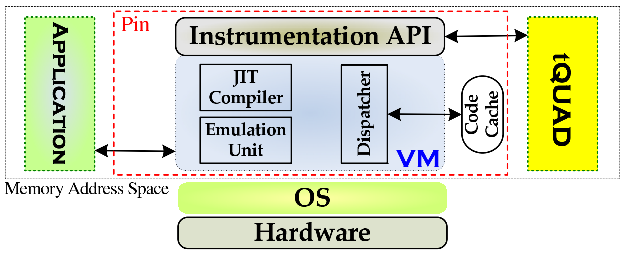


Figure 4.6: Architectural overview of tQUAD

Besides the standard Pin system initialization, i.e. PIN_Init() and PIN_InitSymbols (see 3.3), tQUAD implements two API calls: *INS_AddInstrumentFunction()* for instruction level instrumentation and *RNT_AddInstrumentFunction()* for routine level instrumentation. These two API calls are used to set up calls to the instrumentation routines *Instruction()* and *UpdateCallStack()*, respectively.

Figure 4.7 shows the body of the *Instruction()* instrumentation routine. The *Instruction()* instrumentation routine sets up the call to the analysis routine *IncreaseRead()* every time an instruction that references memory read is executed. There is a similar process in the case of memory write reference. *Instruction()* also monitors instructions for the *return* from a function to maintain the integrity of the internal call stack. When Pin starts the execution of an application, the JIT calls *Instruction()* to insert new instructions into the code cache. If the instruction references memory or signaling the return from a function, tQUAD inserts a call to the corresponding analysis routine before the instruction, passing the required arguments which can be the Instruction Pointer (IP), the number of bytes read or written, and a flag showing whether or not the instruction is a prefetch. The corresponding analysis routines return immediately upon detection of a prefetch state for an instruction. *INS_InsertPredicatedCall()* injects the analysis routine and ensures that the analysis routine is invoked only if the instruction is predicated true. When local stack area memory accesses have to be excluded, the Stack Pointer (*REG_STACK_PTR*) is also passed as an extra argument to the analysis routine for subsequent processing. Furthermore, *Instruction()* is responsible to initiate the time simulation and memory bandwidth snapshot managements.

```
VOID Instruction(INS ins, VOID *v)
{
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)IncTotalInstCount, IARG_END);
    if (INS_IsRet(ins))  // return from routines is monitored
        INS_InsertPredicatedCall(ins, IPOINT_BEFORE, (AFUNPTR)Return,
            IARG_INST_PTR, IARG_END);
    if (!No_Stack_Flag)  // stack accesses ok
    {
      if (INS_IsMemoryRead(ins) || INS_IsStackRead(ins))
        INS_InsertPredicatedCall(ins,IPOINT_BEFORE,(AFUNPTR)IncreaseRead,
            IARG_MEMORYREAD_SIZE,IARG_UINT32,INS_IsPrefetch(ins),IARG_END);
      if (INS_HasMemoryRead2(ins))
        INS_InsertPredicatedCall(ins,IPOINT_BEFORE,(AFUNPTR)IncreaseRead,
            IARG_MEMORYREAD_SIZE,IARG_UINT32,INS_IsPrefetch(ins),IARG_END);
      if (INS_IsMemoryWrite(ins) || INS_IsStackWrite(ins))
        INS_InsertPredicatedCall(ins,IPOINT_BEFORE,(AFUNPTR)IncreaseWrite,
            IARG_MEMORYWRITE_SIZE,IARG_UINT32,INS_IsPrefetch(ins),IARG_END);
    } // end of Stack is ok!
    else  // ignore stack accesses
    {
      if (INS_IsMemoryRead(ins))
        INS_InsertPredicatedCall(ins,IPOINT_BEFORE,(AFUNPTR)IncreaseReadSP,
            IARG_REG_VALUE,REG_STACK_PTR,IARG_MEMORYREAD_EA,IARG_MEMORYREAD_SIZE,
            IARG_UINT32, INS_IsPrefetch(ins),IARG_END);
      // similar calls for NS_HasMemoryRead2 & INS_IsMemoryWrite(ins)
    } // end of ignore stack
    // check for the snapshot point
    INS_InsertCall(ins,IPOINT_BEFORE,(AFUNPTR)Slice_checkpoint,IARG_END);
}
```

Figure 4.7: tQUAD instruction instrumentation pseudocode

The code for the *UpdateCallStack()* instrumentation routine is presented in Figure 4.8. The *UpdateCallStack()* instrumentation routine sets up the call to the analysis routine *EnterFC()* every time a function is called during program execution. This is necessary to update the internal call stack. Since tQUAD ignores the functions which are not in the main image file of the program, *flag* is used as a signal to indicate the location of the newly-called function. The name of the function, as reported by Pin, is also passed for the internal call stack update process.

```
VOID UpdateCallStack(RTN rtn,VOID *v)
{
     bool flag;
     char *rNtemp;
     string rName;
     flag=(!((IMG_Name(SEC_Img(RTN_Sec(rtn))).find(mainImg)) == string::npos));
     rName=RTN_Name(rtn);
     rNtemp=new char[strlen(rName.c_str())+1];
     strcpy(rNtemp,rName.c_str());
     RTN_Open(rtn);
     // Insert a call at the entry point of a routine to update Call Stack
     RTN_InsertCall(rtn, IPOINT_BEFORE, (AFUNPTR)EnterFC, IARG_PTR, rNtemp,
         IARG_BOOL, flag, IARG_END);
     RTN_Close(rtn);
}
```

Figure 4.8: tQUAD routine instrumentation

4.2.3 How tQUAD Works

To start tQUAD the same method as described in Section 3.5 and as the QUAD tool is used. It is repeated here for the sake of consistency.

```
pin -t [QUAD-path] QUAD.so [QUAD-options] – [application-name]
[application-options]
```

on a Linux system and

```
pin -t [QUAD-path] QUAD.dll [QUAD-options] – [application-name]
[application-options]
```

on a Windows system.

tQUAD supports the following command line options:

- *-filter_uncommon_functions* - This option filters out uncommon function names which are unlikely to be defined by the user. Examples are function names beginning with underscore(s), question mark, etc. This is useful if the user wants to restrict the analysis only on functions that are actually part of the application that is analysed, excluding library function. The default value for this option is set to true. Hence, to see all the functions in the main image file the '-filter_uncommon_functions 0' option will allow this.

- *-slice* - This option allows to perform an analysis based on an interval. This interval is based on the number of instructions, i.e. the user chooses at which interval he/she wants to get a snapshot of the analysis results.

- *-ignore_stack_access* - This option make it possible to exclude accesses to the local stack memory, providing a view of the data transferred via non-stack region.

Table 4.1: Phases in the execution path of the *hArtes wfs* application.

phase	phase span	% phase span	kernel	activity span	average memory bandwidth usage				maximum memory bandwidth usage (R+W)		aggregate MBW
					read access		write access				
					stack incl.	stack excl.	stack incl.	stack excl.	stack incl.	stack excl.	
initialization	53-144	0.007	ffw	92	1.8071	1.2422	0.4807	0.1811	2.4704	1.6376	2.6018
			ldint	1	0.0798	0.0162	0.0516	0.0176	0.1314	0.0338	
wave load	552-14660	1.1103	wav_load	14109	2.0993	1.0358	1.0355	0.9929	3.1566	2.0664	3.1566
wave propagation	540-274868	21.5891	vsmult2d	1570	0.1799	0.0655	0.1182	0.0503	0.3996	0.1548	1.4530
			calculateGainPQ	1600	0.3708	0.0815	0.2633	0.0847	0.7714	<0.2116	
			PrimarySource_deriveTP	235	0.0870	0.0240	0.0547	0.0208	0.2820	0.0980	
WFS main processing	14663-592803	45.4983	fft1d	278781	2.4179	0.3876	0.3501	0.1331	2.8738	<0.6428	<84.1862
			DelayLine_processChunk	115546	2.0859	0.2356	0.2339	0.1180	<3.3316	1.7050	
			bitrev	116755	1.8677	0.2521	0.7457	0.1934	<2.8778	0.4966	
			zeroRealVec	36304	2.1529	0.0145	0.7233	0.3610	<2.9386	<0.4028	
			AudioIo_setFrames	616	21.5553	21.8035	21.0646	21.5860	<53.2686	<52.7330	
			perm	116776	0.3252	0.0280	0.0956	0.0545	<0.6556	<0.1466	
			cadd	41076	0.9882	0.3590	0.6686	0.2753	1.6946	0.6514	
			cmult	41073	1.1456	0.3590	0.7080	0.2753	1.8946	0.6594	
			Filter_process	42583	0.7789	0.3609	0.1332	0.1141	<0.9768	0.5064	
			Filter_process_pre_	1487	1.1113	1.6384	1.6267	1.6290	<3.3862	<3.3302	
			zeroCplxVec	4132	1.7693	0.0141	0.5913	0.3926	2.6874	<0.4710	
			r2c	2716	1.9250	0.1510	0.4474	0.2983	<2.9386	<0.5642	
			c2r	2318	1.9251	0.1774	0.3549	0.1769	2.9138	0.4658	
			AudioIo_getFrames	502	0.8701	0.8296	0.8268	0.8137	1.7482	1.6866	
wave save	592804-1270674	53.3469	wav_store	677871	1.7492	1.0033	0.8064	0.7765	2.7244	1.9044	2.7244

4.2.4 tQUAD's Example Output

As explained in the introduction to this section, tQUAD is able to provide *memory bandwith* information for each kernel during its execution time. Also, it finds the *timing* and *activity span* of kernels. This data is collected in text files which can be later converted into graphs and/or tables. An example output of tQUAD, taken from [29], is presented in Table 4.1.

In Table 4.1 *phase span* indicates the starting and ending time slices for the phase; *% phase span* is the percentage of the phase time interval to the program whole execution time span; *activity span* represents the number of time slices in which the kernel is active (accesses memory); *memory bandwidth usage* is measured in bytes per instruction; *aggregate MBW* represents the summation of all kernels' maximum memory bandwidth usages in the phase including the stack area accesses.

4.3 Concluding Remarks - The xQUAD Tool

The two tools described in this chapter allow to analyse applications memory behaviours. QUAD specifically addresses the detection the actual data communication dependencies between functions, while tQUAD's goal is to find the execution time of kernels along with their memory bandwidth usage. Both tools deliver important information that can be used during critical decision in HW/SW co-design stages, particularly for HW/SW task partitioning, mapping and scheduling of applications onto heterogeneous reconfigurable platforms.

Nevertheless, the described tools do not have capabilities for *recognizing* low-level source code information and, therefore, it is difficult to discretize *where* inside a kernel a certain memory behaviour appears. By providing low-level source information, memory locations can be *coupled* with the *variable name* that occupies this location.

Hence, the xQUAD tool augments the analyses performed by the earlier described tools by delivering even more detailed information about an application's memory usage.

xQUAD is able to provide detailed memory usage on a *function-level*, allowing to spot which memory location is occupied by which variable, how many times this variable is referenced, and the type of operation done on the memory location, i.e. read or write. Furthermore, xQUAD provides information about the complete memory map of an application, i.e. the memory location that are used by functions in terms of time. These can help gaining insights about the total memory usage of applications on a high level. Additionally, the tool can provide memory statistics that summarize, per kernel, the number of used memory addresses divided per *stack*, *heap* and *data* memory region, along with the number of total accesses and the average of accesses per memory location.

As this tool is developed during this thesis project, it is described in more details in the following chapter.

5 The xQUAD Tool

This chapter presents the xQUAD tool, a tool that provides detailed memory usage information of applications. xQUAD can be seen as an extension to the QUAD and tQUAD tools (presented in Chapter 4), as it augments the analysis provided by those tools with finer grained memory informations. Nonetheless, xQUAD can be used independently to provide a global view about the memory usage of an application, as well as detailed information about memory usage of variables. Along with the memory usage analysis, it also provides low-level source information about variables. Pin does not provide any functionality for reading data symbols from object files, hence these data symbols are retrieved directly from the object file, i.e., without the intervention of the Pin framework.

This chapter gives first a global overview into the xQUAD tool. Afterwards, background information about object files in general and about the ELF object file format in particular, is provided. Also, the DWARF debugging information is introduced, along with the library used to read DWARF debugging symbols. Subsequently, the xQUAD architecture an is explained and a description of the usages of the tool is presented.

5.1 Introduction and overview of xQUAD

The xQUAD tool augments the memory analysis generated by QUAD and tQUAD tools. It provides memory access information of an application on a *variable granularity* as well as data about about the *global memory usage*. These data is output in flat text files, which purposes are twofold, as they permit the developer to read these files as well as to visualize them. Visualizing an application's memory behaviour can give a global overview on the memory access patterns of the application, possibly discovering potential memory bottlenecks. Afterwards, the user can always turn back to the flat files which give the *precise* information.

By providing these files in a simple text format, the user can then parse these files extracting the needed information, like the *number of accesses on local memory*, the *frequency of accesses on heap*, the *ratio of usage* of different memory segments, etc. These information, along with the information provided by QUAD and tQUAD, can be used as a guide for revising the application code or, dependent on the developer's need, it can be used as a preliminary step in the HW/SW partitioning and mapping process of the application.

Because one of the purposes of this tool is to give a clear understanding of the memory usage inside functions, on variable's level, it is important for the tool to have some recognition of source-level information. The Pin DBI framework does not provide functions API for retrieving data symbols information. Therefore, source-level information as variable names must be extracted from the debugging section of the object file. One prerequisite is that the application must be compiled with the debugging information flag on, which permits the compiler to augments the object file with a *debugging section*. In case of an ELF executable, compiled with GCC, the standard debugging format is DWARF version 2[39] (see Section 5.2.2), unless otherwise specified.

Following, this chapter will give a background introduction into the format of object files and of the debugging information section inside it. As this tool currently supports only ELF object files, no other object file formats are considered in the subsequent sections.

5.2 Background Information

As the Pin Dynamic Binary Instrumentation framework does not provide an interface to the data symbols information of an object file, an alternative method is implementated for retrieving low-level sorce code information. Provided that the application contains a *debugging information section*, xQUAD extract these information by implementing a module for this purpose. The extraction of the debugging data happens in a *transparent* way, i.e., the developer is not aware of this process.

Next section explains briefly the format of an ELF object file and the position of the debugging section inside it. Subsequently, the DWARF Debugging Information format is presented, along with the interface used to retrieve these debugging information. The next subsections should provide enough information to the reader for the rest of the chapter.

5.2.1 Object File Overview

An object file contains the object code and the data produced by a compiler and assembler. More specifically it contains:

- *Header information* - General information about the file.
- *Object code* - Binary instructions and data generated by a compiler or assembler
- *Relocation* - Information about addresses that have to be fixed up when the linker changes the addresses of the object code.
- *Symbols* - Global symbols kept by linker while running
- *Debugging Information* - Other information about the code needed for debugging purposes. This includes source file and line numbers information, local symbols, and other low-level source code information.

The above information are typical for object files, although there may be an object file containing little or no information beyond the object code. The structure of object files is determined by the target architecture at which these object files are supposed to work. In this thesis we consider the ELF object format[1]. ELF object files can be

[1]As introduced before, in this thesis report we will consider only the ELF object format. Nevertheless, the information reported in this paragraph can be considered applicable, for most parts, to different object formats

of three types, namely *relocatable*, *executable*, and *shared objects*. Relocatable files hold code and data suitable for linking with other object files to create an executable file or a shared object file. These files are denoted by the .o extension. Executable files have all relocations done and all symbols resolved, except perhaps shared library symbols that are to be resolved at run-time. Shared object files contain code and data that are used by the linker to process this data and code with other relocatable and shared objects, creating another object file. Also, the linker may use the data and code of a shared object file to be dynamically linked with an executable and other shared object files, creating a process image. Shared objects are denoted by the .so extension. Even though these different types of object files serve for different purposes, their internal structure is in many ways similar. Therefore, in this thesis focus is put only on executable object files.

From a linker perspective, ELF executables are divided into *sections*, used by the linker for building a program. From a program execution perspective, ELF executables are divided into *segments*, where each segment usually groups multiple sections together[38]. Typically, object files have a ratio of 7-9 segments and 35-40 sections[2]. Typical sections contained in executable objects are the *.text* section containing the program code, the *.data* section containing initialized global and static variables, and the *.bss* section containing uninitialized global and static variables. When the object file is compiled with debugging information on, an additional section *.debug* appears in the object file. This section is, however, not present in any *segment*, as debugging information are not included in the memory space of the application's image. A high level representation of the two views of an ELF file is showed in figure 5.1. To be noted in the figure that the *.debug* section in the linking view is not *mapped* into the exection view.

Upon running an executable, the loader loads these segments into memory creating a process image in the memory. Upon loading and running the executable, the kernel looks at the executable header, locates the .text section within the executable, loads it into the appropriate portions of memory, and marks these pages as read-only. It then locates the .data section in the executable and loads it into the user's address space, this time in read-write memory. Finally, it finds the location and size of the .bss section from the image header, and adds the appropriate pages of memory to the user's address space. Even though the user has not specified the initial values of variables placed in .bss, by convention the kernel will initialize all of this memory to zero. Local variables are not stored in any section in the executable, but are created at run-time.

An ELF header resides at the beginning and holds a road map describing the file's organization. Sections hold the information of an object file for the linking view: instructions, data, symbol table, relocation information, and so on. A program header table, if present, tells the system how to create a process image. Files used to build a process image (when executing a program) must have a program header table; relocatable files do not need one. A section header table contains information describing the file's sections. Thus, the section header table lets one locate all the file's sections. Every section has an entry in the table; each entry gives information such as the section name, the section size, etc., and occupies one contiguous (possibly empty) sequence of bytes within a file.

[2]Information acquired empirically

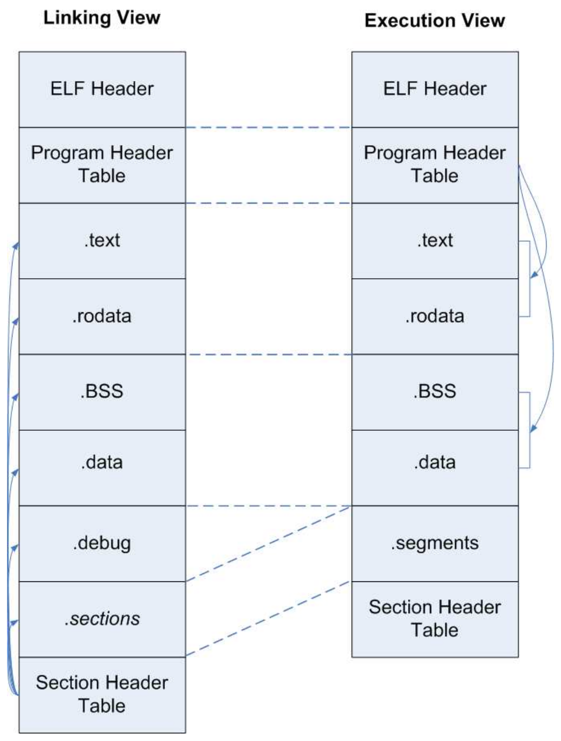


Figure 5.1: Sections and segments of an ELF

Sections in a file may not overlap. No byte in a file resides in more than one section. Files used during linking must have a section header table; other object files may or may not have one. NB. Sections and segments do not have a specified order. Only the ELF header has a fixed position in the file. Hence, the organization of Figure 5.1 may differ. Special sections hold program and control information. Special sections are, for example, .bss, .data, .debug, .fini, .init, .rodata, .text, etc. .bss holds uninitialized data that contribute to the program's memory image. By definition, the system initializes the data with zero when the program begins to run, the section occupies no file space. .data holds initialized data that contribute to the program's memory image. .fini holds executable instructions that contribute to the terminating process of the application. That is, when a program exits normally, the system arranges to execute the code in this section. .init, holds executable instructions that contribute to the process initialization code. That is, when a program starts to run, the system arranges to execute the code in this section before calling the main program entry point (main in C). .rodata holds read-only data that typically contribute to a non-writable segment in the process image. The section .text holds the text, or executable instructions, of a program.

Hence, the loader maps each segment in the object file into memory. The segments essentially become memory areas of an executable[40]. Figure 5.2, taken from [40], depicts the memory layout of a program that is about to begin execution.

Local variables, temporaries, function parameters, spilled registers, etc. are allocated on the stack memory. Furthermore, a heap space is dynamically allocated on demand, as

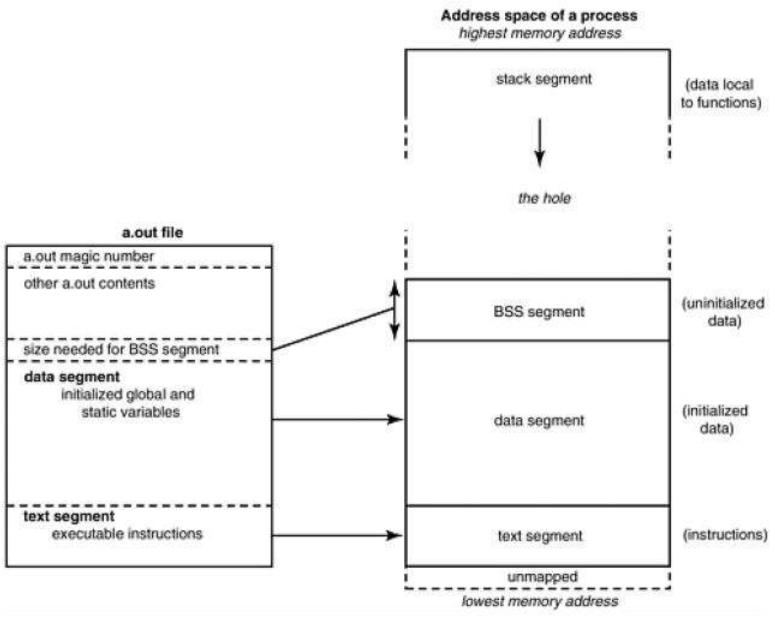


Figure 5.2: Segments of an executable object file mapped in memory

soon as the first call to a memory allocation function, like *malloc*. Note that the lowest part of the virtual memory address space is unmapped; this is still part of the address space of the process, but has not been assigned to any physical address, so any references to it will be illegal. This is typically a few KB of memory from address zero up and is used to catch references through null pointers.

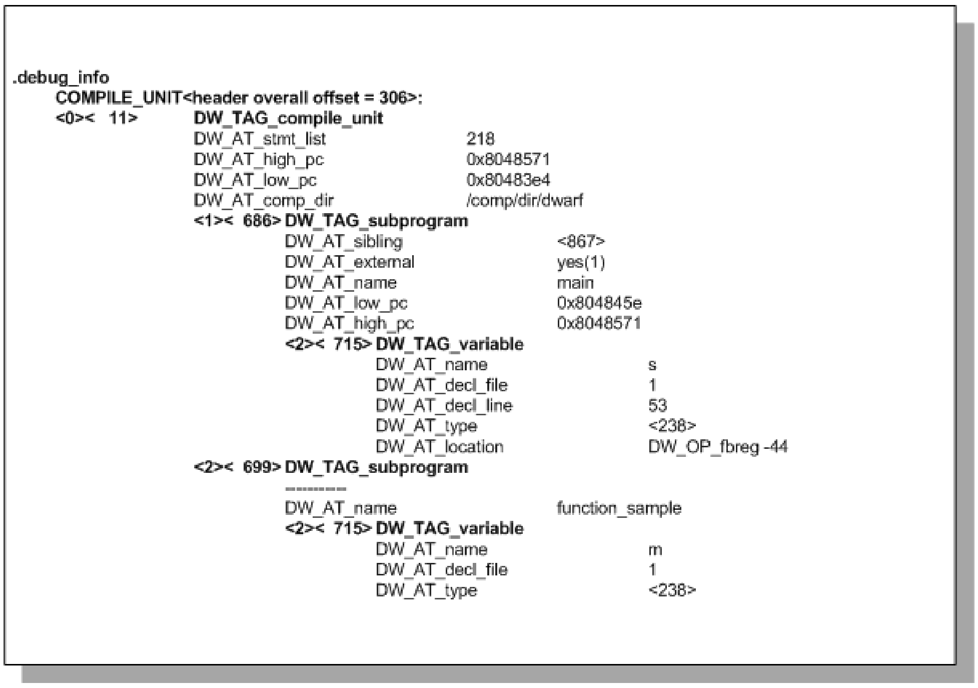
5.2.2 DWARF Debugging Format

After the brief description of an object file contruction, this section introduces general information about the standard debugging format associated to ELF, i.e., DWARF (Debugging With Arbitrary Record Format)[39, 14] version 2[3]. The DWARF debugging information specification is a portable debugging information format. Originally designed for the Intel 32-bit architectures, it has been extended for other platforms including MIPS, ARM, SPARC and PPC, and for 64-bit architectures.

When the ELF executable is compiled with the debugging flag on, besides the sections describing code, data and stack segments, the executable also includes a debugging section. Although DWARF is most commonly associated with ELF, it is independent of the object file format. DWARF provides debugging entries to define low-level source code representation, like, among others, source file and line number information, local symbols, and descriptions of data structures.

[3]DWARF 2 format was specified in 1995, prior to the 1998 ratification of the ISO C++ standard. Thus, version 2.0 does not include specification for some important C++ language constructs such as namespace or mutable class attributes. The DWARF version 3.0 format, which is under review, addresses the version 2.0 language deficiencies and provides a format to sufficiently cover the debugging information required for more recently developed languages such as Java.

The DWARF section inside an object file consists in turn from different subsections, which are all prefixed by the .debug_ keyword. Each .debug section contains three elements, namely DIE sections, DIEs and DIE attributes. A .debug section contains DIE sections, a DIE sections contains DIE, and DIEs have attributes. The most important DWARF section is the .debug_info section, which is the DWARF core data containing the actual debugging information. These debugging information are contained in a data structure called Debugging Information Entry (DIE), which is used to represent each compilation unit, variable, type, procedure, etc. The DIEs (intended to exists in the .debug_info section) are organized in a tree structure: the root of the tree is the debug section, which contains DIE sections (or Compilation Units). The example in Figure 5.3 is an output from the dwarfdump tool. The format is modified to resemble the tree structure of the .debug_info section.



```
.debug_info
    COMPILE_UNIT<header overall offset = 306>:
    <0><  11>       DW_TAG_compile_unit
                    DW_AT_stmt_list            218
                    DW_AT_high_pc              0x8048571
                    DW_AT_low_pc               0x80483e4
                    DW_AT_comp_dir             /comp/dir/dwarf
                    <1><  686> DW_TAG_subprogram
                              DW_AT_sibling              <867>
                              DW_AT_external             yes(1)
                              DW_AT_name                 main
                              DW_AT_low_pc               0x804845e
                              DW_AT_high_pc              0x8048571
                              <2><  715> DW_TAG_variable
                                        DW_AT_name                 s
                                        DW_AT_decl_file            1
                                        DW_AT_decl_line            53
                                        DW_AT_type                 <238>
                                        DW_AT_location             DW_OP_fbreg -44
                    <2><  699> DW_TAG_subprogram
                              ---------
                              DW_AT_name                 function_sample
                              <2><  715> DW_TAG_variable
                                        DW_AT_name                 m
                                        DW_AT_decl_file            1
                                        DW_AT_type                 <238>
```

Figure 5.3: DWARF DIEs sample output

From the output in Figure 5.3, it can be noted that the root of the DWARF structure is, in this case, a .debug_info section. As mentioned before, .debug_info contains one or more DIE sections describing Compilation Unit(s) (CU). Each DIE is described by a *tag*. Hence, a DW_TAG_compile_unit describes a compilation unit DIE. Each DIE has a number of attributes, denoted in DWARF by DW_AT_*attribute*. This is the description of the first DIE, i.e. of the compilation unit DIE. Next, the DW_TAG_subprogram DIE describes the next level in the tree hierarchy. Again, this DIE is augmented with attributes, like DW_AT_name, which describe the function in question (main in this case). The next nested (child) DIE of the DW_TAG_subprogram DIE is the DW_TAG_variable DIE, with its attribute describing the variable in the source code. Then, after having described all the children of the parent DIE (i.e. DW_TAG_subprogram) in question, the next sibling

DIE is processed, which is the example above is again of type DW_TAG_subprogram. Following this depth-first algorithm, the entire tree is described. As most modern programming languages are block structured, DWARF also follows this model. Hence each DIE, except the topmost root DIE representing the CU of the source file, is contained within a parent DIE and may contain children DIEs. DIEs are tree nodes, which may represent types, variables, or functions of a set of attributes, describing the type, name, source line number, location address or references to another DIE (e.g. a reference to a datatype specification). Figure 5.4 shows a sample C program and the related DWARF description of the program.

```
hello.c:
       1: int main()
       2: {
       3:   printf("Hello World!\n");
       4:   return 0;
       5: }
```

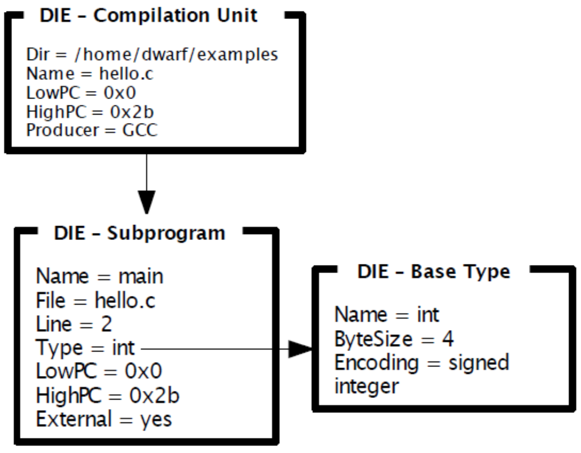


Figure 5.4: C Program and DWARF description of program

5.2.3 DWARF Consumer Library

Following the DWARF structure described before, applications can be build which read and process DWARF information, by using the *libdwarf consumer library interface*[3][4]. This section briefly summarizes the key functions in the library for reading DWARF information.

The first step of every DWARF process is to retrieve the debug descriptor of type Dwarf_Debug from which the DWARF data can be further obtained. The libdwarf library defines a function for this purpose, namely dwarf_init() which returns the debug descriptor. Ending a DWARF operation is accomplished by calling the function dwarf_finish(). The debug descriptor contains all the needed DIE information. As explained in the previous section, every program consists of multiple CUs (even if the application consists of one file, there are other CUs regarding library files).

[4]For creating DWARF debugging sections, a dedicated library is available, see [3] for more information.

The function darf_next_cu_header, called recursively, permits to process each CU in the program. Afterwards, starting from the available CU, functions are called which process each individual DIE in the current CU, by a depth-first tree traversal. Library functions used to accomplish this traversal are dwarf_child() and dwarf_siblingof().

5.3 xQUAD Architecture

The architecture of xQUAD consists of three main parts:

- A module for retrieving DWARF Debugging Information
- Pin's instrumentation functions
- QUAD's analysis functions called from the instrumentation

Figure 5.5 shows the architectural overview of xQUAD and its connection with the Pin framework and the module for retrieving DWARF Debugging Information. Similar to the tools in Chapter 4, xQUAD communicates with Pin's API that permits the tool at run-time to be hooked-up with the Pin framework.

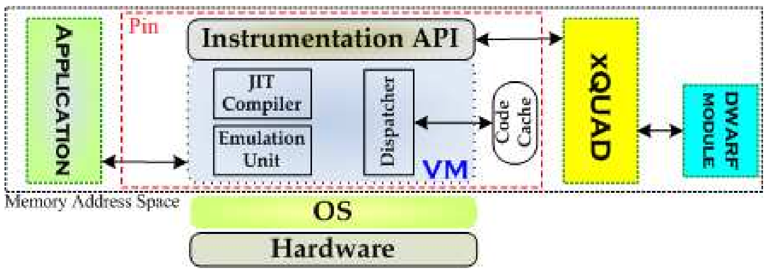


Figure 5.5: Architectural overview of xQUAD

The Pin framework is described in 3.3, but is repeated here the for convenience. At the highest level, there is a Virtual Machine (VM), a code cache, and an instrumentation API. The VM consists of a Just-In-Time (JIT) compiler, an emulator, and a dispatcher. After Pin gains control of the application, the VM coordinates its components to execute this application. The JIT compiles and instruments the application code, which is then launched by the dispatcher. The compiled code is stored in the code cache. Entering (leaving) the VM from (to) the code cache involves saving and restoring the application register state. The emulator interprets instructions that cannot be executed directly. It is used for *system calls* which require special handling from the VM. Since Pin does not reside in the kernel of the operating system, it can only capture user-level code. As Figure 5.5 shows, three binary programs are present when an instrumented program is running: the guest application, Pin, and the xQUAD tool. Pin is the engine that instruments the application. xQUAD contains the instrumentation and analysis routines and it is linked with a library that allows xQUAD to communicate with Pin.

5.3.1 xQUAD Design and Implementation

The design and implementation of the tools described in this thesis have all a similar implementation, as they are all based on the same dynamic analysis framework (see Chapters 3 and 4). However, each tool has its own peculiarity which distincts it from the others. xQUAD, as mentioned before, can be used to perfom two types of analysis:

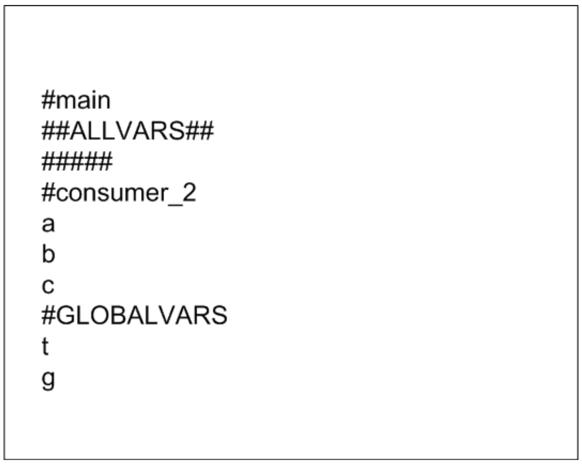
- Detailed variable memory usage information inside a function
- A memory map of the application

Performing an analysis that produce variable memory information for *all* the functions in an application will result soon in a gigantic report file, and most probably unfeasible to read (assuming one can open it, as it can be as big as several tenths of GB). Therefore, the tool must be able to filter out information according to the user needs. Hence, before running the tool, the user is required to define the functions and the variables he/she wants to analyse. These functions and variables are specified in a text file, called the MONITOR file, by following a few simple text formatting rules which are necessary for the tool to correctly read and store the variables of a function to be analysed. Figure 5.6 gives an example MONITOR file describing its format. In this example it is assumed that analysis will be performed on two functions, namely the *main* and *consumer_2* functions, which are prefixed with a #. In case *all variables* in a function needs to be traced, the user should pass this to the tool with the *##ALL-VARS##* keyword. If only *specific variables* need to be analysed, these variables need to be specified individually, below the function name, and each variable on a single line. In case the user wants to analyse the behaviour of *global variables*, passing the *#GLOB-ALVARS* keyword to the tool and specifying these global variables, each on a single line, will accomplish the task. As tracing memory addresses of variables can result in a production of huge files, instructing the tool to analyse the application selectively based on specified variables, will results in a more readable flat file report and a shorter analysis time.

Upon starting, the tool performs some preliminary works as report file initialization, command-line parsing, initialization of data-structures used to hold analysis data and initialization of the Pin framework. As described in Chapter 4, initialization of the Pin framework consists of calling two API functions, i.e. *Pin_initSymbols()*, which initializes the symbol table code, and *Pin_Init()*, for system initialization.

Subsequently, xQUAD parses the MONITOR file and stores the information inside this file into a linked-list data structure, which will be used during execution of the tool to steer the analysis. Figure 5.7 shows a code snippet of the main function of xQUAD. It should be noted that this code snippet is a representative portion of the actual code, although detail of the file and data structure memory allocation checks are omitted for brevity.

As can be seen from Figure 5.7, following various initialization processes, xQUAD makes the call for processing the debugging information from an object file. The following section explains the implementational details of the DWARF Module.



```
#main
##ALLVARS##
#####
#consumer_2
a
b
c
#GLOBALVARS
t
g
```

Figure 5.6: Example of the MONITOR file passed to the xQUAD tool

5.3.2 DWARF Debugging Information Module

The process of retrieving *DWARF Debugging Information* happens, as mentioned earlier, in a transparent way, i.e., the user is not aware of this process and the only requirement is that the object file is compiled with the debugging information. The module for extracting debugging information is based on *dwarfdump*, an open source tool which prints DWARF information from the .debug section of an object file in human readable format, using the *libdwarf* C library[3]. However, dwarfdump is used only as a starting point, as dwarfdump provides very generic output. Hence it has been decided to build a new module from scratch tailored to QUAD's needs, omitting unnecessary DWARF information.

The *libwarf library* contains API functions for reading and writing applications using DWARF 2 information, using the *libdwarf consumer library* and the *libwarf produces library*[3], respectively. xQUAD needs only to read DWARF information, hence in this section only functions from the consumer library are described.

The first call to the *libdwarf* library regards the initializtion of the DWARF process, which returns a handle for accessing debugging information, associated with a file descriptor of the application. This handle is the main object from which all the reading process is done. As explained in 5.2.2, the *.debug_info* section of the application consists of multiple Computational Units (CUs), which all contain debugging information that are to be extracted. Therefore, the debugging information handle is passed to a function for the extraction of each CU.

In the most simple case, where an application consists of only one source file, i.e. one .c/.cpp file, the application will have a single CU of interest, and a number of CUs corresponding to libraries. These libraries CU are excluded from the debugging information parsing process.

In 5.2.2 it is explained that each CU consists of several (siblings) Debugging Information Entries (DIE), which in turn can contain other child DIEs. This forms a tree of DIEs, which is traversed by a *depth-first algorithm.* The DWARF library provides

```
int main (int argc, char *argv[])
{
    char temp[100];
    string var_monitorfilename;

  //open file for detailed analysis and initilization of file
    varTrace.open ("/home/marco/Desktop/QUAD_VarTraceAndTiming.txt");
    varTrace<<"\n\t\t\tVariable Tracing and Timing Report\n\n";
    varTrace<<"MEMORY ADDRESS"<<setw(25)
      <<"VARIABLE NAME"<<setw(15)
      <<"R/W"<<setw(18)
      <<"RTN COUNT"<<setw(23)
      <<"TIME STAMP"<<"\n\n"<<endl;

  //Initilize Pin symbols table and framework
  PIN_InitSymbols();
    if( PIN_Init(argc,argv) ) return Usage();

    //Store slice value
    VarTrace_Snapshot_Interval=atoi(KnobSlice.Value().c_str());
  //Get name of the MONITOR list file to use
    var_monitorfilename=KnobVarMonitorList.Value();

    //Data structures initialization
    CallStack.push("Out_of_the_main_function_scope"); // put the first record in
        our callstack
    // assume 'Out_of_the_main_function_scope' as the first current routine
    current_f_name="Out_of_the_main_function_scope";

    mem_addr_struct = new MEM_ADDR;

    mem_addr_struct->map_addr.insert(pair<string,UINT64>("sample_address", 0));
    map_func_addr_couple[current_f_name] = mem_addr_struct;

  /*Command line parsing code omitted*/

    /* Code for parsing of MONITOR file and for initialization of Memory Map files
         are omitted*/

    //Start Debugging Information processing
    init_addr2var(temp, comp_dir_name);

    //Call to Pin APIs for registering instrumentation functions
    RTN_AddInstrumentFunction(RegisterFunctionCall, 0);
    INS_AddInstrumentFunction(TraceTimingInstruction, 0);

    PIN_AddFiniFunction(Fini, 0);

    cerr << endl << "Starting Variable Tracing and Analysis Application" << endl;

    PIN_StartProgram();

    return 0;
}
```

Figure 5.7: xQUAD main function

functions as *dwarf_child* and *dwarf_siblingof* which accomplish the task of traversing the tree. Figure 5.8 depicts this process. Next, for each CU, a table is constructed holding *variable names* and an *offset* with respect to the *current frame base address*. The current frame base address is retrieved through the call of a Pin API (described later) and the actual address of the variable will be then calculated by adding this address to the via Pin retrieved base pointer address.

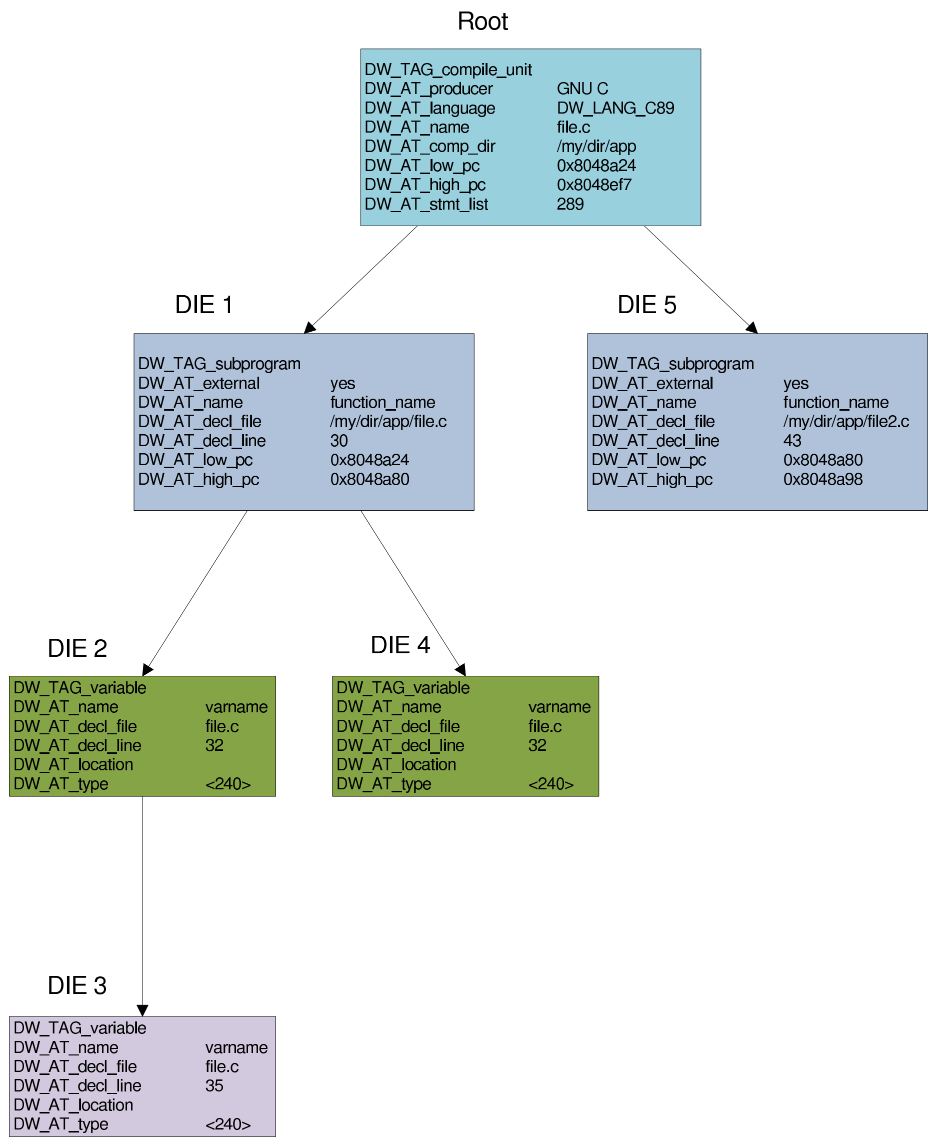


Figure 5.8: DWARF Depth-First Tree Traversal

Subsequently, each DIE, and for each of its children, a function is called for processing the DIE data. As mentioned earlier, each DIE represents types, variables, functions, etc., which is specified by a *tag*. The tool retrieves the tag by calling the library function *dwarf_tag* and filters out DIEs with a:

- *DW_TAG_subprogram* tag, for rerieving debugging data about functions
- *DW_TAG_variable* tag, for, not surprisingly, retrieving debugging data about vari-

ables

Furthermore, in case of a DW_TAG_variable, an additional distinction is made for *local* and *global* variables. The implementation details for processing functions and (global/local) variables are described next.

Functions When processing a subprogram DIE (i.e. a function), the tool compares the name of the function DIE (by calling the DWARF library function *dwarf_diename*) with the function names stored in the data structures parsed from the MONITOR file. Upon processing a function DIE in which we are interested, i.e. predefined in the monitor list, this function name is stored in the data structured containing the information parsed from the monitor file. Additionally, it is checked for the presence of the DW_MIPS_linkage_name attribute, which represents the internal name representation assigned by the compiler. In case of a C++ compilation, function names are *mangled* to simplify the compiler's process[5]. This attribute, if present, is also stored in the data structures as in a later stage of the analysis there is the need to compared function names stored in these data structures with function names that are recognizeable by the Pin framework, which are also *mangled*.

After determining if the name of the function matches a function's name stored in the MONITOR file, the tool needs to filter out the variable names accordingly.

Local Variables A DW_TAG_variable tag denotes a variable, either local or global. When a local variable is encountered, this variable DIE is a child DIE corresponding to a *parent* subprogram DIE. In this case, by using the library function *dwarf_attrlist* a list of attributes corresponding to this variable DIE is retrieved. Then, for each attribute in the attribute in the list, by the function *dwarf_whatattr*, a call to a function is made for processing these attributes of the variable DIE. Each variable DIE (global or local) has a *location attribute* (DW_AT_location). When it regards a local variable, the DW_AT_location location attribute, as mentioned before, will contain an offset value. This value is stored in a data structure holding variable names/offset values pairs. The case for global variables requires a slightly different approach, which will be described next.

Global Variables In case of a global variable, the variable DIE presents some other attributes that are not present in case of a local variable. Additionally, DWARF builds up different variable DIEs in case an application is compiled with GCC or G++. This is taken into account by the tool.

When an application is compiled with G++, the global variable is described by DWARF in *two steps*. First, DIEs are built with an attribute *DW_AT_declaration*, along with other common variable descriptions like variable name, line number, data type, and so on. This attribute is *only present in the case of G++ compilation*, which is used by the tool to discriminate between GCC/G++ compiled binaries. In this case, the DIE does not have a location attribute, which is stored in another DIE, containing the

[5]as C++ provides function overloading, multiple functions may have the same name. It is therefore necessary to provide an unique name for each function. This process is called *name mangling*

DW_AT_specification attribute. This attribute appears only in DIEs in this situation. This *regular* structure simplifies somewhat this process, and the fixed global address from the DW_AT_location attribute can be retrieved and stored, as mentioned before, along with the variable name of the global variable.

In case of an application compiled with GCC, the process of retrieving the global variables addresses is very straightforward compared to the two steps method described above. In this case, the DIE describing the global variable directly contains the DW_AT_location attribute. Hence, the global variable address can be directly stored in the corresponding data structure.

The process described here recursively repeats for each DIE in the current Computational Unit (CU). When the current CU has no DIE left, the process is again repeated for each CU present in the application. Finally, when every CU is processed, the DWARF process is terminated and the file descriptor for the application is closed. At this point all the debugging information is kept in memory, which are used subsequently during the analysis done with the Pin framework.

5.3.3 Pin API calls and Analysis Routines

After completion of the process of reading the debugging information, the tool starts the instrumentation process. As explained in 3.3, Pin permits the user to instrument at different granularities. xQUAD instruments the application at two different granularities, namely at *routine level* and at *instruction level*, by calling the API functions *RTN_AddInstrumentation()* and *INS_AddInstrumentation()*, respectively. RTN_AddInstrumentation calls the *RegisterFunctionCall* every time a new routine is executed. The purpose for this function call is to keep an own version of the call stack for routines' calls. Here the name of the currently executing routine is retrieved by calling the API function *RTN_name* and then by using *RTN_Open*, the routine is opened to be processed. *RTN_InsertCall* inserts a call relative to the earlier opened routine, and calls the analysis function *FunctionCallStack*. The RTN_InsertCall function call is shown below.

```
RTN_InsertCall( rtn , IPOINT_BEFORE,
     (AFUNPTR) FunctionCallStack ,
     IARG_INST_PTR, IARG_PTR,
     name_char ,
     IARG_END)
```

Afterwards the currently executing routine rtn is closed by calling the *RTN_Close*, which must be called before opening any other routines.

The necessity for having an own version of the call stack arises from the fact during analysis it is required to know if a *currently executing function* has been *called* for the first time, meaning it is producing memory accesses also for the first time. The other possibility is that a function *returned* from a callee, meaning that the data accesses currently produced must be accounted for an *previous call*.

In FunctionCallStack the tool builds the actual function call stack. Off course the function calls stored in the call stack are to be matched with the functions defined in the MONITOR file by the user. In this case, the previously stored mangled name is compared with the currently executing function that Pin provides (through RTN_name). In case that we have no mangled name stored it means that the mangled name is not present and thus the comparison can be done simply based on (unmangled) function names. Upon a match, these functions are stored in a call stack, so that a flow of function call will be available.

After having built our own function call stack, the INS_AddInstrumentation API function is called. This function allows instrumentation analysis at instruction level. This instrumentation API calls the instrumented function *TraceTimingInstruction*, which is responsible for calling analysis routines based on the current executing instruction. Upon entering the instrumented routine, there is a call to

```
INS_InsertCall( ins ,
    IPOINT_BEFORE,
    (AFUNPTR) timingCount ,
    IARG_END)
```

The *timingCount* analysis function is responsible for maintaining a *time stamp counter* and a *slice counter*. The time stamp counter holds the number of instructions executed till a certain point in time. This effectively implements a timing functionality to the tool, based on the number of instruction. The slice counter, conversely, is used to hold a timer in case the user wants to analyse an application based on a certain slice interval. For instance the user may want to analyse the application every 10000 instructions.

The tool also checks for *return instructions* (INS_IsRet(ins)). This check is done to be able to store the return address of a function, used for building the memory map. As explained above, retrieving the last address of a function with DWARF does not appear to be reliable, because of the possibility that the last address of the function retrieved with DWARF may be a NOP instruction, which is never executed by Pin, as Pin does not instrument instructions that are never executed. The call to the analysis routine is shown below.

```
INS_InsertPredicatedCall(ins ,
                IPOINT_BEFORE,
                (AFUNPTR) Return ,
                IARG_INST_PTR,
                IARG_END)
```

The InsertPredicatedCall is called only if the instruction is predicated true, i.e. if the instruction is actually executed. The analysis routine Return finds the name of the current routine specified by the current instruction pointer, using the RTN_FindNameByAddress Pin's API. Then, for each function defined in the MONITOR file, this analysis routine stores a high program counter value using Pin's *StringFromAddrint* function and pushes this function into the own representation of the call stack.

As mentioned before, xQUAD's main purpose is to analyse the memory usage of variables in an applications. Hence, each instruction operating on memory is instrumented and filtered by using Pin's *INS_IsMemoryRead/Write* and *INS_IsStackRead/Write*. In case an instruction has two memory read operations, the *INS_HasMemoryRead2* is used. If the above are true, the following call to the analysis routine is made in case of a memory read operationg:

```
INS_InsertPredicatedCall(ins,
                    IPOINT_BEFORE,
                    (AFUNPTR) RecordTrace,
                    IARG_INST_PTR,
                    IARG_UINT32,
                    'R',
                    IARG_MEMORYREAD_EA,
                    IARG_UINT32,
                    INS_IsPrefetch(ins),
                    IARG_END)
```

The INS_InsertPredicatedCall assures that the analysis routines is only called upon an actual execution of the instruction. Depending on the type of memory operation, i.e. Read of Write, the character 'R' or 'W' is sent. Also, in case of a write operation on memory the IARG_MEMORYWRITE_EA parameter is sent, representing the effective memory address where the write operation occurs; conversely, IARG_MEMORYREAD_EA represents the effective memory address of the occurring read operation. The parameter INS_IsPrefetch(ins) is true if we are dealing with a prefetch instruction.

The analysis routine RecordTrace stores this information in a structure, in case the INS_IsPrefetch(ins) is false, i.e. the instruction is not a prefetch instruction. xQUAD excludes memory prefetching instructions as these can pollute the memory access pattern.

5.3.4 How xQUAD Works: Functionalities and Restrictions

Running xQUAD is done in a similar way as explained for QUAD and tQUAD in Chapter 4.1 and 4.2, and stated below.

```
pin -t [xQUAD-path] xQUAD.so [xQUAD-options] – [application-name]
[application-options]
```

As mentioned before, xQUAD supports a couple of command-line options, according to the type of analysis the user wants to perform. These command-line options are:

- *-memory_map* - This option instructs the tool to create a memory map of the application. When this is desired, the tool produces one file in which all the memory usage inside a function are reported. This file is then post-processed to produce three file containing the *stack addresses*, *heap addresses*, and the *data addresses* per function. Later, these files can be used for visualization purposes.
- *-variable_monitor_list* - This option enables the users to have a report file based on some predefined functions and varibles, excluding all other (undesired) functions/-variables. The function and variable names to monitor are specified in a text file, following some simple rules which are described in Section 5.3.1.
- *-slice* - This option enable the user to specify the slice number based on which an analysis is made.

One restriction of the xQUAD tool is that it is developed to work on x86 platforms running Linux. However, both Pin and DWARF are known to run on multiple platforms, therefore, the possibility to port xQUAD to other platforms should not be very difficult.

5.4 xQUAD Analysis Output Examples

As explained in the introduction of this chapter, the xQUAD tool generates different flat files that describe the memory usage of an application both in general and in detailed levels. Basically, the outputs of the tool can be divided according to the command-line options described above, i.e., a *memory map* of the entire application and a *detailed memory report file* on function and/or variable level. As these files contain the entire *memory trace of the application*, several information can be parsed out from here, according to the developers need.

In the rest of this section, both analysis outputs are explained by giving examples of the produced files as well as *potential* usage of these files. In Chapter 6, these files are applied to the analysis of a real-life case study.

5.4.1 Memory Map Report File

Instructing the tool to perform a memory map analysis of an application will generate an flat file containing all the addresses used in a function, per function call. An example of a memory map file is showed in Figure 5.9. Furthermore, as can be seen in the figure, the *frequency* of the used memory addresses is also provided, giving a clear idea of the type of the *type of addresses* used in a function and their *intensity* on a certain memory location.

```
wav_load
0x000000000060ed20   56
0x00002b1724b64b00   45
0x00002b17250bd89c   1
0x00002b17250c35a4   1
0x00002b17250cddfc   1
0x00007fff3dbfb408   1
0x00007fff3dbfb7c0   51
0x00007fff3dbfb7c8   195
0x00007fff3dbfb7d8   89
waveprop
0x00007fff3dbfb778   1
vmag2d
0x00007fff3dbfb530   1
vsub2d
0x00007fff3dbfb538   1
wfsClocalizeSources
0x00007fff3dbfb688   1
vmag2d
0x00007fff3dbfb500   1
r2c
0x00007fff3dbfb7cc   2
bitrev
0x00007fff3dbfb75c   3
perm
0x00007fff3dbfb788   1
bitrev
0x00007fff3dbfb758   1
0x00007fff3dbfb75c   3
```

Figure 5.9: xQUAD memory map example

However, such a memory map file can get quite big, and most times it becomes unfeasible to inspect. In fact, as explained before, it is adviced to perform a memory map analysis based on time slice snapshot, generating a more contained memory map file. The example in figure 5.9 is taken from an analysis made with a time slice snapshot of every 10000 instructions. Also, even though the produced file is generated with a *real* application (see Chapter 6), it should be mentioned that the showed memory map is adjusted to give a general idea of the file, and thus some addresses are omitted.

Nevertheless, after having generated a memory map file, the developer may want to filter out addresses based on their *memory region.* This will show in detail the *typeof memory* used by a certain function. Additionally, statistics can be gained about this usage, e.g., the *ratio* of used local memory versus main memory. Table 5.2 demonstrates this by using a function from the Wave Field Synthesis (WFS) application, which is described in more detailed in the case study in Chapter 6.

Table 5.1: Example of statistics for the wav_load function from the WFS application

	nr of addresses	total nr of accesses
Stack addresses	138	20597714
Heap addresses	502127	730790
Data addresses	57	33559334

Besides generating flat, text based files, users may want to *visualize* the generated memory map files so to gain a faster and more abstract idea of the general memory behaviour. An example borrowed from Chapter 6 is showed in Figure 5.4.1. In Chapter 6 more examples are described and exaplained more in detail.

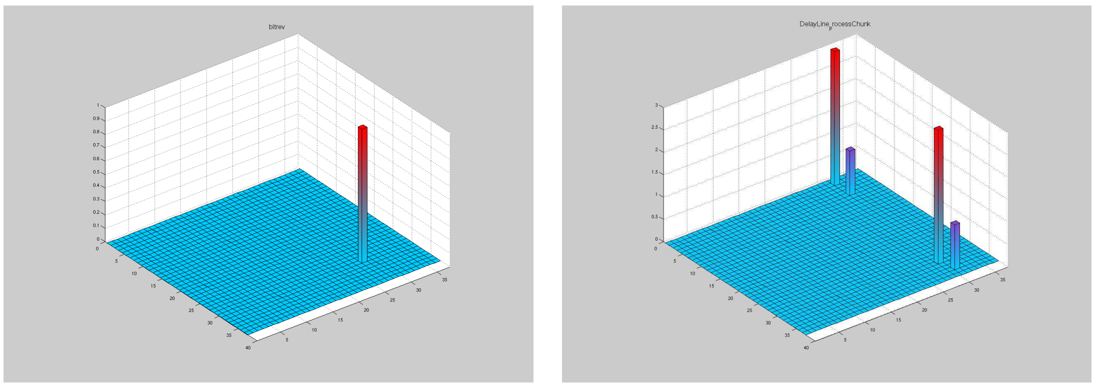


Figure 5.10: Example of memory visualization

The xy-axes in Figure 5.4.1 represents a certain *memory region space* of an application (heap, stack, or data region). The z-axis represents the *number of accesses* on this particular memory location.

5.5 Detailed Variable report file

The other option in xQUAD is to produce a detailed flat file, describing the *details* of the memory usage on *variables level.* The file contains information like

- Effective Address of the variable
- Name of the variable augmented with the function name it belongs to
- Type of operation done on the specific memory location, i.e. read/write
- Number of time a specific variable is referenced
- Timestamp at which a certain variable emmory address access is done

An excerpt from a detailed variable memory usage file is showed in Table **??**. This example is also taken from the WFS application, which is described in the next chapter.

Table 5.2: Example of statistics for the wav_load function from the WFS application

MEMORY ADDRESS	VARIABLE NAME	R/W	COUNT	TIME STAMP
0x7fffff19638	k::fft1d	R	615	1006400000
0x7fffff19620	u::fft1d	R	615	1006410000
0x7fffff19620	u::fft1d	W	615	1006420000

Such a detailed file can be filtered, again according to the specific needs of the developer, so that detailed information on variables and timing in which these variables are active can be retrieved. Of course, the user may want to run the xQUAD tool for a specific variable only, omitting other variables and thus producing a more contained file.

5.6 Conclusion

This chapter described the xQUAD tool, a tool that reports detailed memory usage data of applications. xQUAD augments the analysis provided by the QUAD and tQUAD tools (see Chapter 4) with both a global analysis about applications and a detailed analysis of the memory usage *inside* a kernel. It permits to selectively perform an analysis based on a MONITOR file, that the user is required to provide before executing the tool.

The data generated by xQUAD can be filtered out to get *specific information* about the memory behaviour of an application, according to the user needs. These information can be applied during stages in a HW/SW co-design methodology, like partitioning, mapping and scheduling of an application on a *heterogeneous platform* containing *reconfigurable devices.* Furthermore, the produced information can serve as an hint for optimizing applications.

The architecture and implementation of xQUAD is similar to the tools described in Chapter 4, developed using the Pin[22] DBI framework, described in Chapter **??**. However, the Pin framework does not provide any API for accessing low-level source code information. Hence, to provide xQUAD with such a facility, an external module (i.e., outiside the Pin framework) is implemented, using the DWARF Debugging Information library.

xQUAD is developed and tested on IA32 and Intel 64 platforms, with a Linux operating system. As a future work, this tool can be ported to other operating systems and/or other architectures. In fact, the Pin framework provides support for multiple architectures and binaries (see Chapter 3.3).

The next chapter discusses the usage of xQUAD on a case study on the Wave Field Synthesis[5] audio processing application.

Case Study: Wave Field Synthesis 6

This chapter demonstrates the usage of the tool described in Chapter 5 on the *hArtes Wave Field Synthesis* (WFS) audio processing application. The main goal is to present a detailed memory usage behaviour of this application. The extracted information can be applied during critical decisions in HW/SW Co-design stages, especially for HW/SW partitioning, mapping and scheduling of the application for an heterogeneous reconfigurable platform. Furthermore, the information acquired during this analysis can be used to spot memory related bottlenecks and to revise the application code so that data communication to/from the memory subsystem, is improved.

This case study will show the potentialities of the xQUAD tool by inspecting the behaviour of the memory accesses of the WFS application. In particular, the tool is used to show the global memory usage of the application, and the intensity and behaviour of the memory usage in different memory regions (i.e. stack, heap, data).

Additionally, a *ranking method* is presented which tries to extract the *memory penalty factor* for a kernel from the cumulative execution time retrieved with the *gprof* general profiler. This could give an indication of the memory intensity of a kernel relative to its execution time. However, this metric does not reflect any particular quantitative value of measurement, but it only computes an *index* by taking the *ratio of memory accesses over gprof time*. This index is therfore only applicable for comparison purposes between kernels, which permits to draw up a ranking method.

6.1 Introduction to Wave Field Synthesis

The Wave Field Synthesis[5] concept is a 3D audio rendering technique characterized by the creation of a virtual source and a virtual room. WFS is based on the *Huygens principle*, which, informally, states that each point in a wave front can be considered as a *primary source* for the creation of new, *secondary* waves, which in turn become a primary source for other waves. Hence, an advancing wave can be seen as *constructed* by the summation of all the secondary waves arising from previously *primary source* waves. This principle is reproduced by *loudspeaker arrays* that generate a complete sound field in the listening zone which is identical to an appropriate real sound event. By recording a sound emitted from a *source* (S) with an array of *n microphones* (M), and then by reproducing these recorded microphone signals with an array of equally arranged *n loudspeakers*, the synthesized wave front is created and propagated through a *listening area*. In any place in the listening area, a listener perceives a *realistic reproduction* of the *primary* virtual sound source. In other words, as the listener moves, for instance, along the loudspeaker array, he/she hears the sound in the same direction as it was originally propagated by the virtual sound source, i.e. when the listeners moves towards the location of a virtual source the amplitude of the sound increases in a realistic way[36].

Figure 6.1, adapted from [36], depicts the described situation.

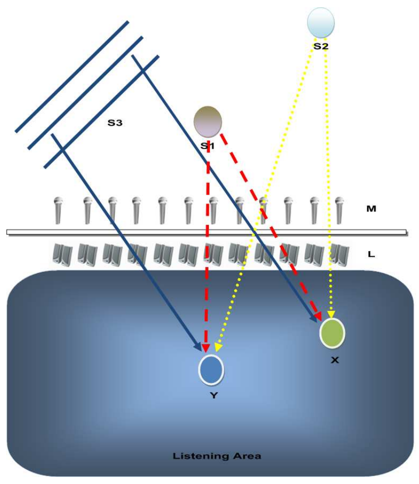


Figure 6.1: Principle of WFS. Source S1 and S2 emit a spherical wave, while source S3 propagate a plane wave. These three sources propagate their signal, which is captured and recorded by an array of microphones M. These signals are then reproduced by an equally arranged array of loudspeakers L, propagated into a listening area. In this area, listener X and Y perceive a realistic sound, e.g. listener X perceives the sound from virtual source S1 differently than listener Y, as the direction of the waves is reproduced accordingly.

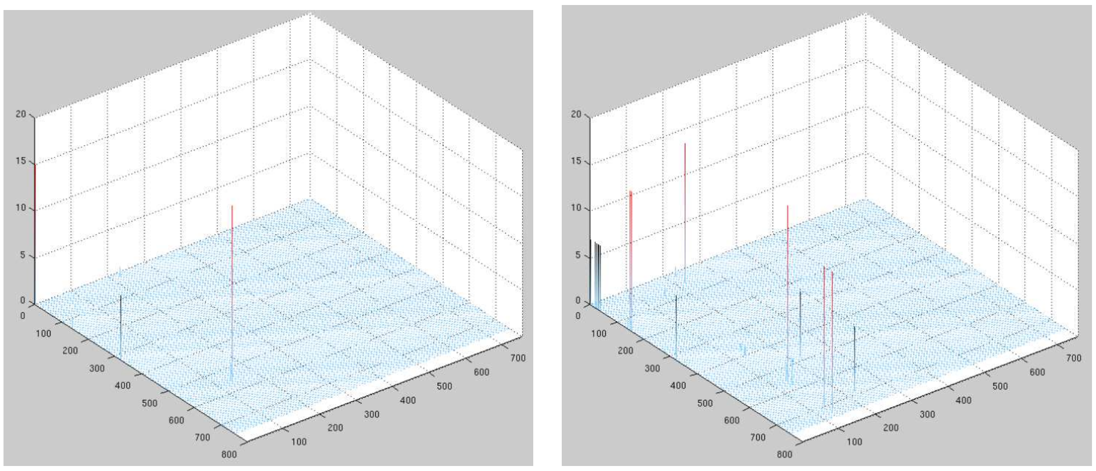
The hArtes WFS application provided by Fraunhofer IDMT[1] implements a self-contained wave field synthesis system.

6.2 Experimental Setup

The experiments were executed on an Intel 64-bit Core 2 Quad CPU Q9550 @ 2.83GHz with a main memory of 8GB, running Linux kernel v2.6.18-164.6.1.e15. The hArtes WFS source code was compiled with gcc v4.1.2. To be able to read debugging information, the source code is compiled with the *-g* flag enabled. Also, to use the *gprof* general profiler, the program is compiled with the *-pg* flag on. xQUAD is executed with a time slice interval ranging from 500 to 10000 instructions per time slice. Furthermore, xQUAD is run with the *-memory_map* and the *variable_monitor_list* command-line options, which used to enable an analysis based on the general memory behaviour of the application and for retrieving detailed information about variable memory tracing, respectively. The *hArtes WFS* runs in off-line mode. This means that the input audio source is read from a file instead of an audio device. In all the experiments, one primary source and thirty-two secondary source (S and L respectively, see Figure 6.1) are assumed, in an acoustical area of 50 square meters.

6.3 Memory Access Behaviour of WFS

Running xQUAD with the command line option *-memory_map* enables the production of the *memory map* file, which describes the memory access behaviour of WFS during its execution. By post-processing this file, we create three different files where each of it contains memory accesses from a particular memory region, viz., *stack*, *heap*, and *data* memory addresses. As a first step, to get a first impression into the global memory usage of WFS, we visualize these distinct memory flat files. Figures 6.3 and 6.3 show snapshots of the usage of the *heap* memory address space taken at different times during the execution of WFS. The memory locations are mapped on the xy-axes, while the intensity of each memory address is revealed by the z-axis. It turns out that from the begin till approximately the half of the execution time there is a sparse usage of heap memory addresses, while during the second half of the execution time this usage becomes more intensive for a certain range of heap addresses.



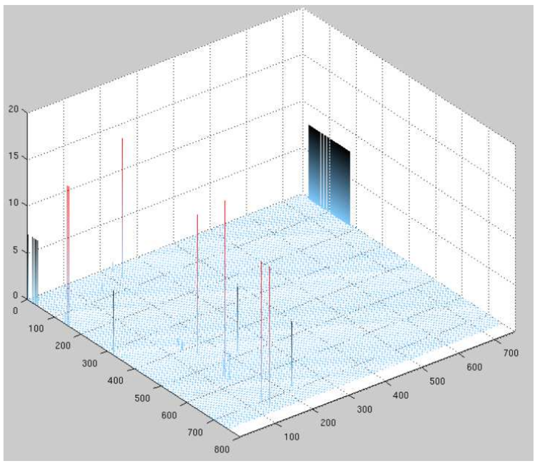
(a) Near the begin of execution time

(b) Near the half of execution time

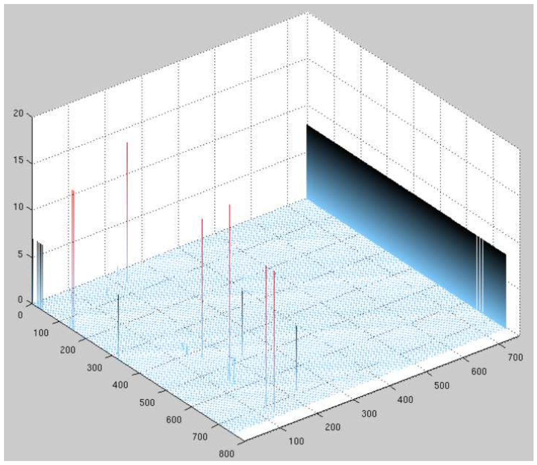
Figure 6.2: Snapshots of the heap memory usage of WFS for the first half of the execution time

The behaviour of intensive heap addresses usage is accounted to the *wav_store* function, which becomes active, approximately at the middle of the execution time of WFS and is the only active function till the end of the execution. This can be explained by the fact that *wav_store* saves the output audio signals produced during its execution from the buffers allocated in memory to an output file in the wav format. To accomplish this, it uses mostly individual heap addresses, which explains the intensive heap usage.

Table 6.1 summarizes all the memory accesses of WFS along with the number of individual memory addresses used. This table confirms the discussed behaviour of *wav_store*, as we can see that the number of individual memory addresses for accessing the main memory is considerably higher compared with the other functions. These results are recorded with a time slice length of 500 instructions, i.e. these results are describing *almost completely* the actual behaviour of WFS w.r.t. its memory usage. Nevertheless,



(a) Near the begin of execution time



(b) Near the half of execution time

Figure 6.3: Snapshots of the heap memory usage of WFS for the second half of the execution time

using a larger time slice, and thus reducing analysis time and disk space usage, can still give an attendable *average* of memory usage[1]. It should be avoided, however, to choose a *too large* time slice, as then possibly too much information will be lost.

Table 6.1: Memory usage statistics for the *hArtes wfs* application.

	stack		**heap**		**data**	
kernel	**number of addresses**	**total number of accesses**	**number of addresses**	**total number of accesses**	**number of addresses**	**total number of accesses**
wav_store	38	3160179	35935	291844	537	131932
fft1d	33	2192660	6	13	9	59
DelayLine_processChunk	41	893129	3	5	9	25
bitrev	26	922918	7	32443	10	64303
zeroRealVec	19	324462	3	281	7	504
AudioIo_setFrames	14	665	32	32	5	12
perm	22	126662	5	17	10	50
cadd	24	86742	5	16155	10	32378
cmult	24	123200	6	16173	10	32251
Filter_process	20	81054	5	19	9	47

The values in this table are recorded with an analysis based on a time slice of 500 instruction cycles.

Interesting to see in Table 6.1 is the behaviour of the *AudioIO_setFrames* function. This function is responsible for copying interleaved audio signal parts into corresponding audio frame in memory. As explained earlier, this study is conducted assuming thirty-two secondary sources (i.e. an array of 32 loudspeakers). Hence, *AudioIO_setFrames* needs 32 distinct addresses for accomplishing its job.

Besides heap, inspecting the local memory of an application can also give interesting insights. The information is also summzrized in Table 6.1. However, the information in

[1]The produced memory map file for WFS with a slice of 500 instructions is almost 50MB. In case of storing *each* instruction the produced file can grow up into the GB range. It is therefore advisable to choose a larger time slice

Table 6.1 can be misleading, as it does not take into account the number of times that a specific function is called. Therefore, WFS is also compiled with profiling information on to make possible the use of the *gprof* general profiler. The results of executing *gprof* on WFS are summarized in Table 6.2.

Here, *wav_store* and *fft1d* are the top two kernels. These two kernels take approximately sixty percent of the whole execution time of the application. Furthermore, Table 6.2 *does* report the number of times a specific function is invoked. At a first sight, in Table 6.1, the *bitrev* function shows a high frequency of stack usage. However, this function is called over 2 million times, which reduces the number of local memory accesses to only a few per call. To gain this information on a high level of abstractness, i.e. without diving directly into numeric values of memory usage, again the visualization feature can be used.

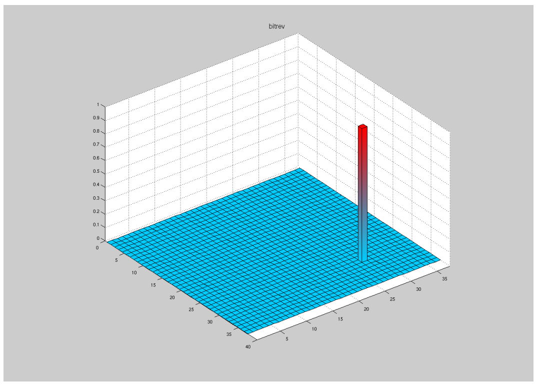
Table 6.2: Flat profile for the *hArtes wfs* application.

kernel	**self %time**	**seconds**	**self calls**	**total ms/call**	**ms/call**
wav_store	31.91	0.28	1	277.25	277.25
fft1d	28.23	0.25	984	0.25	0.25
DelayLine_processChunk	14.23	0.12	493	0.25	0.38
bitrev	8.19	0.07	2015232	0.00	0.00
zeroRealVec	7.44	0.06	15782	0.00	0.00
AudioIo_setFrames	4.01	0.03	493	0.07	0.07
perm	2.07	0.02	984	0.02	0.09
cadd	0.79	0.01	1009664	0.00	0.00
cmult	0.73	0.01	1009664	0.00	0.00
Filter_process	0.71	0.01	493	0.01	0.73
wav_load	0.44	0.00	1	3.80	3.80
Filter_process_pre_	0.35	0.00	493	0.01	0.35
zeroCplxVec	0.28	0.00	495	0.00	0.00
r2c	0.16	0.00	490	0.00	0.00
c2r	0.14	0.00	493	0.00	0.00
AudioIo_getFrames	0.14	0.00	489	0.00	0.00
ffw	0.08	0.00	2	0.35	0.35
vsmult2d	0.02	0.00	7026	0.00	0.00
calculateGainPQ	0.02	0.00	6994	0.00	0.00
PrimarySource_deriveTP	0.02	0.00	236	0.00	0.00
ldint	0.01	0.00	1	0.10	0.10

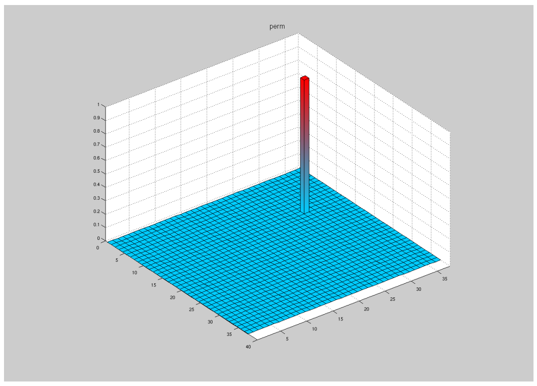
Figure 6.3 shows a typical starting situation of the WFS application, where the most of the stack usage is accounted for a recurring sequence of the functions *bitrev*, *perm*, *fft1d*, and *DelayLine_processChunk*.

Snapshots are also taken during approximately the half of the execution time for the same functions. As Figure 6.3 shows, the memory behaviour of these functions remains the same. This is true for the entire execution of the application, and also for most other functions, especially for those functions responsible for mathematical operations or preliminary work on the audio frames, which mostly conduct the same operations for each function call.

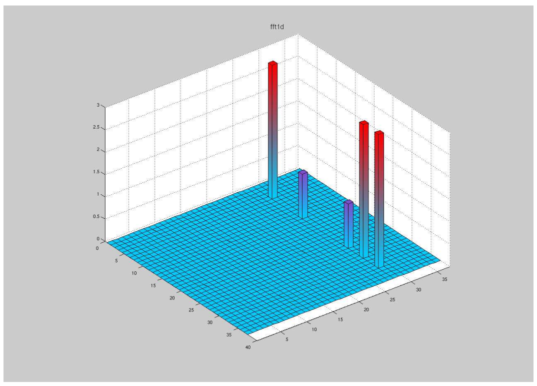
The last execution phases of the discussed functions show a memory access behaviour, depicted in Figure 6.3, which is consistent with the behaviour of these functions in the other two cases. However, as we explained before through the usage of the heap memory



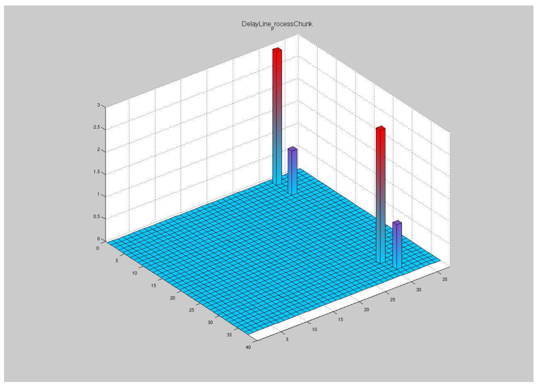
(a) bitrev



(b) perm



(c) fft1d



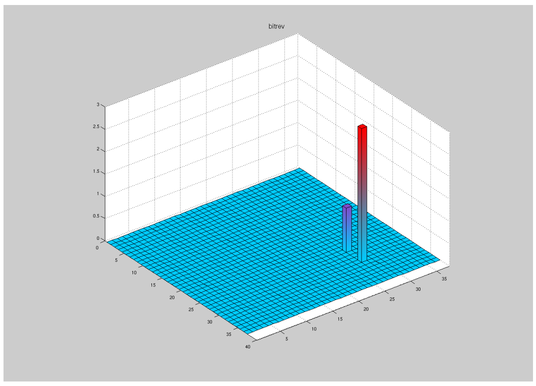
(d) DelayLine_processChunk

Figure 6.4: Snapshots of the stack memory usage of the WFS during begin of the execution time

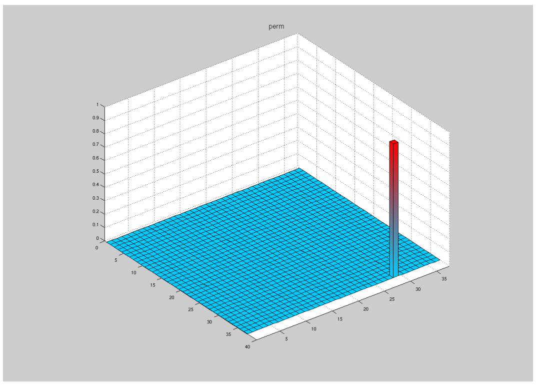
visualization, the last active function producing interesting memory usage in the program is *wav_load*. In Figure 6.3 it becomes apparent that this function exhibits an intensive memory usage behaviour also for local memory.

6.4 Ranking of Memory Intensive WFS Kernels

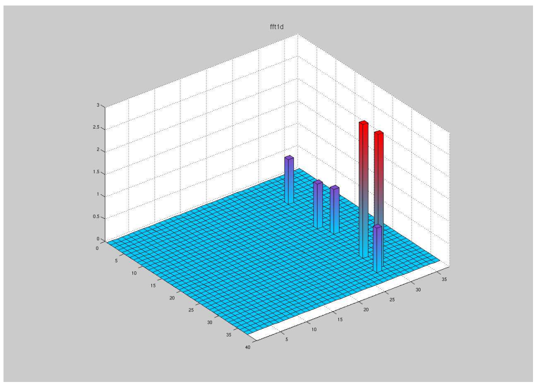
We also tried to calculate the time spent on memory operations w.r.t. the time spent for computations. The gprof profiler provides a cumulative time for each kernel, failing to distinguish between time spent on computations and time spent on memory operations. Selection of potential candidates for hardware implementation based on only the computational intensity cannot be regarded as an accurate metric in the context of reconfigurable computing systems. The reason for this is the high communication data that a system must provide to reconfigurable devices. See Chapter 2 for more information. Finding the time that is spent on a memory access is an arduous task, as this time depends extensively on the intrinsic nature of the underlying platform. This becomes even harder when operating on CISC systems. As no easy way exists for estimating the



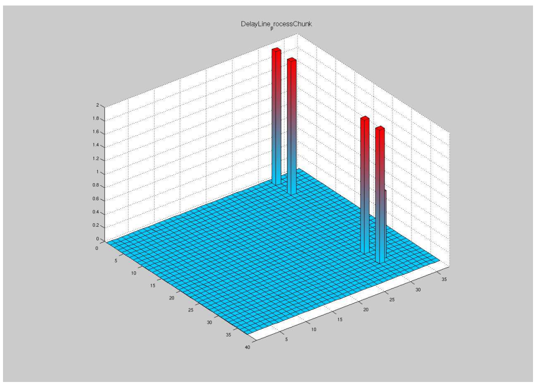
(a) bitrev



(b) perm



(c) fft1d



(d) DelayLine_processChunk

Figure 6.5: Snapshots of the stack memory usage of the WFS during approximately half of the execution time

time taken for a memory access, we developed a *ranking system* to help the estimation of the time spent on memory w.r.t. the time spent on calculations. Under the assumption that

$$\text{number of memory accesses} \sim \text{time}^2$$

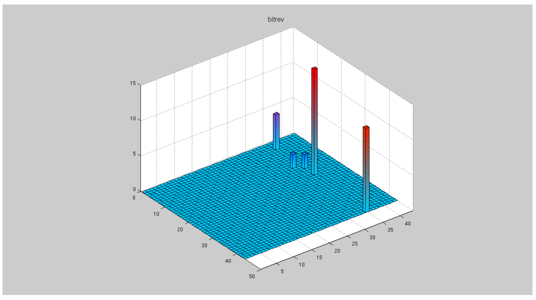
$$\text{total cumulative time} \sim \text{time}$$

we find an *index* that can be used as a metric for discriminating between the memory intensiveness of kernels. This index is calculated by

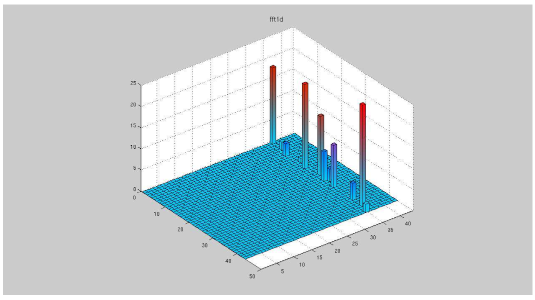
$$\frac{\text{memory accesses}}{\text{total cumulative time}}$$

The total cumulative time is retrieved from the *gprof* profiler in Table 6.2, while the memory access data is listed in Table 6.1. In this case, the stack accesses of the kernels are not taken into account, as most time penalty is expected to be for accessing the heap and data segment of the memory subsystem. In Table 6.3, the*Memory Accesses*

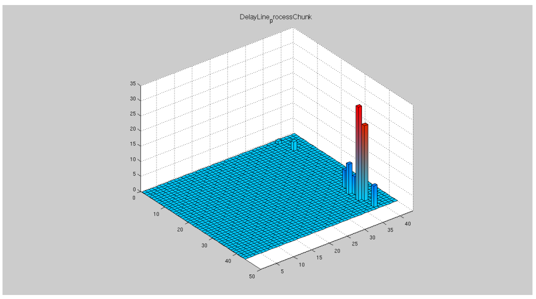
[2]Given two variables y and x, y $\sim$ x denotes the proportionality relation between these two variables.



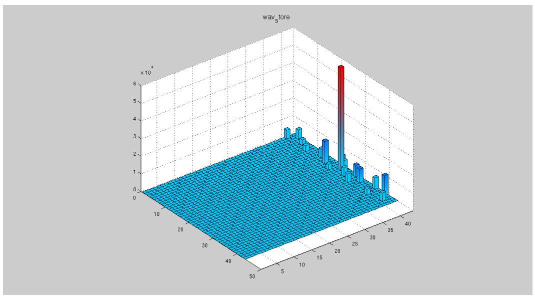
(a) bitrev



(b) fft1d



(c) DelayLine_processChunk



(d) wav_load

Figure 6.6: Snapshots of the stack memory usage of the WFS during the end of the execution time

column reports the sum of the heap and memory accesses, while the *gprof Time* is the cumulative time spent on executing a kernel (see Table 6.2). The *Index* column indicates the memory intensiveness of the kernel relative to the cumulative time spent in the kernel.

Table 6.3: Rank of WFS kernels

Kernel	**Number of Memory Accesses**	**gprof Time in sec.**	**Index**
wav_store	423776	0.28	1513486
bitrev	96746	0.07	1382086
zeroRealVec	785	0.06	13083
AudioIo_setFrames	44	0.03	1467
fft1d	72	0.25	288
DelayLine_processChunk	30	0.12	250

Table 6.3 sorts the kernels from the most intensive kernels to the least intensive, based on the calculated *Index*. As expected, *wav_store* is still accounted as the most expensive kernel in terms of memory usage. Interestingly, *fft1d* and *DelayLine_processChunck*, which were accounted as the second and third most expensive kernels (according to gprof profiling information) drop to the last of this list when we consider our Indexing scheme.

This means that the time spent for memory accesses *relative* to the total, cumulative time spent in the kernels, is not dominating, which makes these kernels good candidates for hardware implementation. Additionally, it was confirmed by Table 6.1 where *fft1d* and *DelayLine_processChunck* show an intensive behaviour on local memory and access rarely the main memory. However, to effectively take advantage of the hardware implementation of these two kernels, their input buffer should be also be implemented on chip.

6.5 Conclusion

This chapter discussed the usage of the xQUAD memory analysis tool applied to the Wave Field Synthesis audio processing application. By producing a memory map of the application, interesting information can be gained on the memory usage behaviour of the application. Visualizing this memory map revealed that during the end of the application's execution time, a high intensity of main memory addresses usage occurs. By parsing the memory map file, statistical information about the memory usage of the application could be gained, which showed that this intensive memory usage was due to a specific function. Furthermore, it is showed how, by mean of visualization, a certain recurrent pattern could be revealed about the usage of local memory addresses.

Finally, we presented an *indexing* system that permitted us to make a *ranking* of kernels based on their memory usage. This revealed that the most *computationally intensive* kernel (wav_load) it also shows the most *memory instensive* behaviour amonng all kernels. However, the second and third most computational intensive kernels doe not exhibit a high memory behaviour, which would them make a good solution for hardware implementation.

In future work, the analysis conducted here can be extended and validated through other analyses, possibly coupling the xQUAD tool together with the QUAD and tQUAD tools described in Chapter 4.

7 Conclusion and Future Work

In this chapter a conclusion is drawn about the work in this thesis. Furthermore, some suggestions for future work are presented.

7.1 Conclusion

The primary obstacle for improving overall computing system performances arises from the communication bottleneck between a (general-purpose) processor and the memory subsystem. This bottleneck is even more evident with the introduction of heterogenous architectures containing reconfigurable devices. To alleviate this problem, a thorough analysis of the memory access patterns of an application is of vital importance.

The aim of this thesis project was the development of a detailed memory access analyzer, xQUAD, which augments the analysis done by QUAD and tQUAD. QUAD, tQUAD and xQUAD are intended to be an integral part of the dynamic profiling stage of the Delft Workbench and the hArtes toolchains. These toolchains currently profile applications using a general profiler, which does not distinguish between computation time and memory access time. Therefore, during this stage it is not possible to reveal possible memory-related bottlenecks of an application. Hence the need for these tools to have some dynamic memory access analyzer tool.

xQUAD is able to perform a memory access analysis at two levels of detail. The finest level performs a memory access behaviour of the variable inside a function. To produce more readable data, xQUAD augments the memory analysis with low-level source information. Furthermore, xQUAD is able to produce a global memory map of the application, where the data access patterns of each function is output to (text) flat report files.

Detailed information on the memory access behaviour *inside* a function can deliver important information for optimization purposes on a fine grain scale and for discovering possible unusual behaviour of data objects used in such a function. Performing an analysis only on a function grain would hide the internal memory access behaviour of the function.

The xQUAD is furthermore tested with the Wave Field Synthesis application. In this case study the memory access patterns of WFS is inspected, both by visualizing the memory access patterns and by collecting statistics on the memory usage. Visualizing the memory behaviour of WFS revealed that near the end of its execution time, WFS shows an intensive main memory usage. Later, this intensive memory usage appeared to be produced by a single specific function. Visualization of the local memory showed a recurrent memory access behaviour.

Finally, an indexing system is presented, which permitted us to rank the kernels inside WFS.

7.2 Future Work

In this section, some directions for future research and improvements are presented.

Due to lack of time, it was not possible to perform more case studies. However, performing a set of case studies on several applications could reveal some common characteristics about their memory access behaviour. Interesting would be to perform case studies among a set of application in the multimedia domain.

Furthermore, the metric used in the case study described in Chapter 6 could be improved. In fact we know the number of accesses to the local memory and the number of those which access the main memory. Retrieving the number of cycles needed for a main memory access could permit us to distinguish the memory access time from the computation time.

Finally, the xQUAD tool could be ported to other architectures and operating systems. Currently, the xQUAD tool is developed and tested on x86 architectures running a Linux operating system. Porting xQUAD to other architectures requires to use a Pin framework that supports that architecture. The xQUAD application is for most part portable to other architecture, as the Pin framework will internally handle most of the conversion necessities. Nevertheless, xQUAD (and the QUADs toolset in general) uses some archicture specific instructions, which would require to be changed accordingly. Porting xQUAD to other operating systems really depends on the supported object file format of the target OS. In case ELF is not supported, a new module for reading debugging information form the object file would be necessary.

Bibliography

[1] *Fraunhofer Institute for Digital Media Technology*, `http://www.idmt.fraunhofer.de/eng/research_topics/wave_field_synthesis.htm`.

[2] *gcov*, `http://gcc.gnu.org/onlinedocs/gcc/Gcov.html`.

[3] David Anderson, *A consumer library interface to dwarf*, (2010).

[4] David Andrews, Douglas Niehaus, and Peter Ashenden, *Programming models for hybrid cpu/fpga chips*, Computer **37** (2004), no. 1, 118–120.

[5] A. J. Berkhout, D. de Vries, and P. Vogel, *Acoustic control by wave field synthesis*, The Journal of the Acoustical Society of America **93** (1993), no. 5, 2764–2778.

[6] K.L.M. Bertels, G.K. Kuzmanov, E. Moscu Panainte, G. N. Gaydadjiev, Y. D. Yankova, V.M. Sima, K Sigdel, R. J. Meeuws, and S. Vassiliadis, *Profiling, compilation, and hdl generation within the hartes project*, FPGAs and Reconfigurable Systems: Adaptive Heterogeneous Systems-on-Chip and European Dimensions (DATE 07 Workshop), April 2007, pp. 53–62.

[7] K.L.M. Bertels, S. Vassiliadis, E. Moscu Panainte, Y. D. Yankova, C. Galuzzi, R. Chaves, and G.K. Kuzmanov, *Developing applications for polymorphic processors: the delft workbench*, (2006), 7.

[8] Tony M. Brewer, *Instruction set innovations for the convey hc-1 computer*, IEEE Micro **30** (2010), 70–79.

[9] Derek Bruening, Timothy Garnett, and Saman Amarasinghe, *An infrastructure for adaptive dynamic optimization*, CGO '03: Proceedings of the international symposium on Code generation and optimization (Washington, DC, USA), IEEE Computer Society, 2003, pp. 265–275.

[10] A.N.M. Imroz Choudhury, Kristin C. Potter, and Steven G. Parker, *Interactive visualization for memory reference traces*, Computer Graphics Forum **27** (2008), no. 3, 815–822.

[11] Katherine Compton and Scott Hauck, *Reconfigurable computing: a survey of systems and software*, ACM Comput. Surv. **34** (2002), no. 2, 171–210.

[12] Giovanni De Micheli and Rajesh K. Gupta, *Hardware/software co-design*, (2002), 30–44.

[13] Brewer Dongarra, O. Brewer, O. Brewer, J. Dongarra, J. Dongarra, D. Sorensen, and D. Sorensen, *Tools to aid in the analysis of memory access patterns for fortran programs*, Parallel Computing 9 **1** (1988), 25–35.

[14] Michael J. Eager, *Introduction to the dwarf debugging format*, (2007).

[15] Karl-Filip Faxén, Konstantin Popov, Sverker Jansson, and Lars Albertsson, *Embla - data dependence profiling for parallel programming*, CISIS '08: Proceedings of the 2008 International Conference on Complex, Intelligent and Software Intensive Systems (Washington, DC, USA), IEEE Computer Society, 2008, pp. 780–785.

[16] C. Galuzzi and K.L.M. Bertels, *A framework for the automatic generation of instruction-set extensions for reconfigurable architectures*, International Workshop on Applied Reconfigurable Computing (ARC), March 2008, pp. 280–286.

[17] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick, *Gprof: A call graph execution profiler*, SIGPLAN Not. **17** (1982), no. 6, 120–126.

[18] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao, *The chimaera reconfigurable functional unit*, FCCM '97: Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines (Washington, DC, USA), IEEE Computer Society, 1997, p. 87.

[19] Scott Hauck and Andre DeHon, *Reconfigurable computing: The theory and practice of fpga-based computation*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[20] John R. Hauser and John Wawrzynek, *Garp: A mips processor with a reconfigurable coprocessor*, 1997, pp. 12–21.

[21] John L. Hennessy and David A. Patterson, *Computer architecture: A quantitative approach (the morgan kaufmann series in computer architecture and design)*, Morgan Kaufmann, May 2007.

[22] Chi keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa, and Reddi Kim Hazelwood, *Pin: Building customized program analysis tools with dynamic instrumentation*, In Programming Language Design and Implementation, ACM Press, 2005, pp. 190–200.

[23] William H. Mangione-Smith, Brad Hutchings, David Andrews, André DeHon, Carl Ebeling, Reiner Hartenstein, Oskar Mencer, John Morris, Krishna Palem, Viktor K. Prasanna, and Henk A. E. Spaanenburg, *Seeking solutions in configurable computing*, Computer **30** (1997), no. 12, 38–43.

[24] Margaret Martonosi, Anoop Gupta, and Thomas Anderson, *Memspy: analyzing memory system bottlenecks in programs*, SIGMETRICS Perform. Eval. Rev. **20** (1992), no. 1, 1–12.

[25] R. J. Meeuws, K Sigdel, Y. D. Yankova, and K.L.M. Bertels, *High level quantitative interconnect estimation for early design space exploration*, Proc. of ICFPT '08, December 2008, pp. 317–320.

[26] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins, *Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix*, 2003, pp. 61–70.

[27] Nicholas Nethercote and Julian Seward, *Valgrind: a framework for heavyweight dynamic binary instrumentation*, PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation (New York, NY, USA), ACM, 2007, pp. 89–100.

[28] S. A. Ostadzadeh, R. J. Meeuws, K Sigdel, and K.L.M. Bertels, *A multipurpose clustering algorithm for task partitioning in multicore reconfigurable systems*, Proc. of CISIS, 2009, pp. 663–668.

[29] S. Arash Ostadzadeh, Marco Corina, Carlo Galuzzi, and Koen Bertels, *tquad - memory bandwidth usage analysis*, 2010.

[30] S. Arash Ostadzadeh, Roel Meeuws, Carlo Galuzzi, and Koen Bertels, *Quad - a memory access pattern analyser*, Proc. of ARC 2010, 2010, pp. 269–281.

[31] S. Arash Ostadzadeh, Roel J. Meeuws, Kamana Sigdel, and Koen Bertels, *A clustering framework for task partitioning based on function-level data usage analysis*, Proc. of FPGA '09, 2009, pp. 279–279.

[32] E. Moscu Panainte, K.L.M. Bertels, and S. Vassiliadis, *The molen compiler for reconfigurable processors*, ACM Transactions in Embedded Computing Systems (TECS) (2007).

[33] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau, *On-chip vs. off-chip memory: the data partitioning problem in embedded processor-based systems*, ACM Trans. Des. Autom. Electron. Syst. **5** (2000), no. 3, 682–704.

[34] Parthasarathy Ranganathan, Sarita Adve, and Norman P. Jouppi, *Performance of image and video processing with general-purpose processors and media isa extensions*, Computer Architecture, International Symposium on **0** (1999), 0124.

[35] Amar Shan, *Heterogeneous processing: a strategy for augmenting moore's law*, Linux J. **2006** (2006), no. 142, 7.

[36] G. Theile and WittekH., *Wave field synthesis: A promising spatial audio rendering concept*, Acoustical Science and Technology (2004), 393–399.

[37] T.J. Todman, G.A. Constantinides, S.J.E. Wilton, O. Mencer, W. Luk, and P.Y.K. Cheung, *Reconfigurable computing: architectures and design methods*, IEE Proceedings - Computers and Digital Techniques **152** (2005), no. 2, 193–207.

[38] Tools Interface Standards Committee, *Executable and linkable format(ELF)*.

[39] Inc. Unix International, *Dwarf debugging information format, vesion 2*, (1993).

[40] Peter van der Linden, *Expert c programming: deep c secrets*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[41] S. Vassiliadis, G. N. Gaydadjiev, K.L.M. Bertels, and E. Moscu Panainte, *The molen programming paradigm*, Proceedings of the Third International Workshop on Systems, Architectures, Modeling, and Simulation, July 2003, pp. 1–10.

[42] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K.L.M. Bertels, G.K. Kuzmanov, and E. Moscu Panainte, *The molen polymorphic processor*, IEEE Transactions on Computers (2004), 1363– 1375.

[43] Guru Venkataramani, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic, *Memtracker: An accelerator for memory debugging and monitoring*, ACM Trans. Archit. Code Optim. **6** (2009), no. 2, 1–33.

[44] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström, *The worst-case execution-time problem—overview of methods and survey of tools*, ACM Trans. Embed. Comput. Syst. **7** (2008), no. 3, 1–53.

[45] Wayne Wolf, *Computers as components: principles of embedded computing system design*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[46] Rong Yan and Seth Copen Goldstein, *Mobile memory: Improving memory locality in very large reconfigurable fabrics*, FCCM, 2002, pp. 195–204.

[47] Y. D. Yankova, G.K. Kuzmanov, K.L.M. Bertels, G. N. Gaydadjiev, Y. Lu, and S. Vassiliadis, *Dwarv: Delftworkbench automated reconfigurable vhdl generator*, In Proceedings of the 17th International Conference on Field Programmable Logic and Applications (FPL07), August 2007, pp. 697–701.