University of Twente.

Master thesis

Handfish - Tunable heterogenous multi-core simulators on multi-core hosts

Author:
Donker, J.H (s0088412)

Supervisors:
Smit, prof.dr.ir. G.J.M. (UT-CAES)
Kokkeler, dr.ir. A.B.J (UT-CAES)
ter Braak, ir. T.D. (UT-CAES)
Sunesen,dr. K. (Recore Systems B.V.)

July 17, 2012

> "In the authors opinion, entirely too much programmers' time has been spent in writing such [machine language] simulators and entirely too much computer time has been wasted in using them"

Donald Knuth, renowned computer scientist and Professor Emeritus at Stanford University

> "Handfish are small fish, growing up to 15 centimeters (5.9 in) long, have skin covered with denticles (tooth-like scales), giving them the alternate name warty anglers. They are slow-moving fishes that prefer to 'walk' rather than swim, using their modified pectoral fins to move about on the sea floor."

Wikipedia on Handfish

Abstract

In recent years we have seen the rise of the heterogeneous System on Chip. It is no longer uncommon to include multiple divergent cores in a single chip. These cores might be from very different vendors who each ship simulators for their core. This makes creating a simulator of the whole chip difficult since these simulators can not be efficiently combined. Most current approaches do not allow the user to take advantage of the multi-threaded capability of such systems or they do not allow the user to make a trade off between simulation accuracy and performance.

The goal of the Handfish framework is to facilitate the creation of heterogeneous multi-core System on Chip simulators. The focus lies on developing techniques to combine simulators and to allow them to take advantage of the multi-threaded nature of today's host computers. Since not all simulations need the highest level of accuracy, the framework allows the strictness to be decreased in exchange for more performance. This is achieved by changing and/or tuning the synchronization strategy of the constituent simulators.

A simulation algorithm has been developed that uses networking patterns to communicate between the components and that allows different instruction set simulators to be combined into a single simulator with a tunable level of accuracy. The implementation of this algorithm is combined with a simple instruction set simulator to get a better overview of the effects of changing the simulation strictness and the performance of the framework.

The simulator framework has also been used to combine two different instruction set simulators to create a heterogeneous simulator of a systems with two digital signal processors and one general purpose processor besides various homogenous simulators.

Acknowledgments

What you see here is the end of a long journey at the University of Twente, longer than I care to mention (2004). While I started out as an Applied Physics major, now I am finishing as a master of computer science. And for that I want to thank my parents for always supporting me unconditionally throughout the years in whatever crazy thing I did. Without your unconditional support I would have never gotten here.

Secondly I would like to thank my supervisors André, Timon, Kim and Gerard for letting me have a free reign over my thesis subject and for reading some of my long and terrible drafts yet still having meaningful comments and the discussions about the work. I would also like to thank my employer Recore Systems for allowing me to work there part time after my internship and allowing me to use/abuse the Xentium simulator in some of my experiments.

I am also indebted to my friends. I would like to thank Nico Vermeulen for taking the time to read this thesis from time to time. Which was probably a very confusing thing to do with a background in philosophy and industrial design. I am also thankful to Arnoud Mulder for the technical discussions about some aspects of the work. Of the friends that did not help out in a technical or writing sense I would like to thank Fred Kaggwa, Recep Altun, Teo Willink, Jennifer Naumann and Robin Peters for their moral support during this period. To rest of you that I did not mention, I am also grateful that all of you where there, and listened to my stories and frustrations. (Probably more than a few of you thought that was going mad or at least losing my mind)

In terms of work, I can honestly say working on Handfish was in general fun, there were times when I thought I had seriously gone into a very deep swamp. The problem with creating such a piece of software yourself is that you can only blame yourself when you screw up. And of course I did a couple of times (The "nicest" one was the inverted inner loop condition with Dekkers algorithm which made it unfair under lax synchronizations...). In the end I was often surprised by the behavior of Handfish , so it only seems appropriate to name it after a somewhat strange fish that walks with its hands instead of swims.

João Henrique Donker, Enschede

Contents

Abbreviations

ABI	Application Binary Interface.
ADC	Analog-to-Digital Converter.
ASIC	Application Specific Integrated Circuit.
CRISP	Cutting edge Reconfigurable ICs for Stream Processing.
DSP	Digital Signal Processor.
IP	Intellectual Property.
ISA	Instruction Set Architecture.
ISS	Instruction Set Simulator.
NoC	Network on Chip.
RISC	Reduced Instruction Set Computing.
RTL	Register Transfer Level.
SoC	System on Chip.
TLM	Transaction Level Modeling.
UART	Universal synchronous receiver/transmitter.
VLIW	Very Large Instruction Word.

Chapter 1

Introduction

Process miniaturization and the decoupling of Intellectual Property (IP) design from manufacturing of Application Specific Integrated Circuit (ASIC)'s has given rise to the System on Chip (SoC). Increasingly a whole system is composed from multiple IP blocks to create a bespoke chip. This is possible since digital design languages like VHDL and Verilog have made it easy to share designs and it is now possible to build a complete computer in a single chip package. Consider a company that wants to make a chip for a GPS receiver. The company takes several off the shelf IP blocks (designs) and then uses these to create a dedicated GPS receiver chip. The used IP blocks might be a general purpose processor IP from one vendor, a Network on Chip (NoC) interconnect from another vendor and some custom ASIC to accelerate specialized operations. But without software the SoC is just a piece of silicon of which the potential is left untapped. So like most modern systems a SoC contains hardware and software that are designed together.

Sourcing IP blocks from multiple vendors has the the advantage that a vendor does not have to design everything from scratch. Yet it also means that the integrator has to try and integrate different pieces of work scrambled together from various sources. Firstly all the hardware components have to communicate with each other. Secondly many IP blocks come with software to simulate and/or debug them. But often these pieces of software do not communicate well with each other. (For example the provided Instruction Set Simulator (ISS) is written in an esoteric language and the sources are not available) So it is hard to create a whole system simulation in which all aspects are simulated. Software that is written for a SoC with multiple architectures might also need to take into account that the endianness differs among architectures and that they might have different behavior with regard to simultaneous memory accesses. All these factors must be taken into account when creating a designing and simulating a modern SoC.

First we will have a look at the types of SoC's and look at how they are developed. We also take into account how the software and the hardware are designed together. But in this report we will mainly focus on how pieces of

software are developed in the absence of real hardware.

1.1 Types of SoC's

A SoC is a chip that contains multiple diverse IP blocks that together form a system. Such a system can take on multiple forms. A simple SoC might just include a simple processor, some memory and a number of peripherals. While the most advanced SoC's include a large number of heterogenous IP blocks connected via a NoC. This does not mean other interconnects do not exist, but they are far less common.

There are three mainstream types of SoC's

- Single-core
- Homogenous multi-core
- Heterogenous multi-core

A single-core SoC will in general only have a processing core with some memory and some peripherals like a Universal synchronous receiver/transmitter (UART) and an Analog-to-Digital Converter (ADC) . A multi-core SoC that is homogenous will normally have multiple processors but they will all be of the same architecture. An example of such a multi-core SoC would be a quad core ARM SoC that is used in a tablet PC. Heterogeneous multi-core SoC's can contain many different types of processors that do not even have to share the same instruction set architecture. Because certain architectures or different implementations of a architecture might have properties that are desirable in different situations. An example of a SoC that uses different architectures is the TI OMAP chip [4], which contains an ARM core and a proprietary Very Large Instruction Word (VLIW) Digital Signal Processor (DSP) core. The ARM core handles general purpose computing tasks while the VLIW DSP handles the computational intensive tasks like video decoding.

One of the challenges within the domain of SoC design is interconnecting all of these components in a way that each individual component gets the resources they need to function optimally. Resources might be memory or even bandwidth to access memory. The components can be interconnected in the following ways.

- Hierarchy of buses
- Network on Chip
- Combination of a hierarchy of buses and a network on chip

Using a hierarchy of buses is a traditional technique for creating computing systems. A bus means that there is a shared medium to which the devices

are connected an example can be seen in figure 1.1 This interconnect can even be something as simple like a wire with multiple connections. Because not all component are equally fast often a hierarchy of buses is created. A hierarchy of busses compensates for speed differences between components. It gives each device the bandwidth it needs without interfering with devices operating at different speeds.

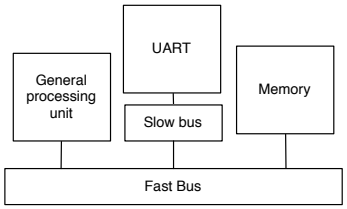


Figure 1.1: Example of a simple SoC with a bus

The NoC paradigm is newer and is used more often in embedded systems. [22] [12] [30]. The principle behind a NoC is that the interconnection between the different devices is seen as a switched network. In this switched network all components are connected to a router which contains the connection the the neighbors. An example can be seen in figure 1.2. The main difference between a bus and a NoC is that the NoC is fully connected to its neighbors but can only reach farther away components by traversing multiple routers. This allows a system to have more parallel transactions since communication does not happen in the same shared medium like with a bus.

Parallel transactions occur when the processors are doing simultaneous work and are reading and writing the data to each other like for example in a streaming application. Examples of NoC s include the work of Pascal Wolkotte and Nikolai Kavaldjiev at the university of Twente [57], the Cutting edge Reconfigurable ICs for Stream Processing (CRISP) consortium General Stream Processor (GSP) [12] and the Intel's polaris chipset [30]. The GSP RFD (Reconfigurable Fabric Device) chip contains 9 Xentium cores connected via a NoC , while the Polaris chip contains 80 simple cores connected via a NoC . The big advantage that a NoC brings is that traffic that travels nearby does not affect the rest of the system. The smaller distances that the wires have to cover also have the advantage that the latencies for nearby communication are shorter. The disadvantage that a NoC has is that in larger systems the data has to travel through many steps, which makes long distance communication slower. Since the signal has to pass trough multiple wires and multiple pieces of combinatorial logic to reach the final destination.

In a combination of a Network of Chip and a bus system one can imagine that the high performance components are connected to a NoC while slower components are joined together on a bus connected to the NoC.

Often a SoC contains two distinct parts, namely the control part and the data processing part. The control part deals with IO and the monitoring of the application and normally runs on a general purpose processor. While the data processing part does the numerical processing and is normally done by an ASIC,

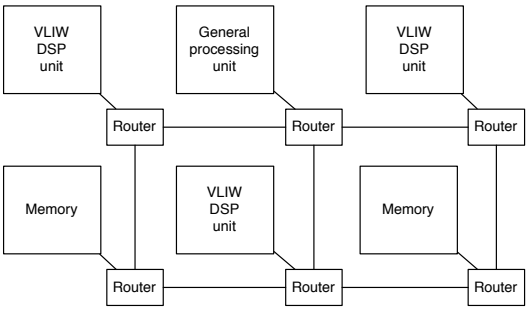


Figure 1.2: Example of a SoC with a NoC

DSP or a reconfigurable part. Often the control part and the data processing part use different architectures with incompatible Instruction Set Architectures (ISA). An example of such a heterogeneous platform is the CRISP consortiums GSP platform. The control part is handled by an ARM processor while the actual data processing happens on the 5 RFD chips, which together house 45 Xentium cores.

1.2 Development of SoC's

In SoC development software and hardware are designed together. The design often begins with a specification of the desired goals for the SoC.

The development of a new System on Chip for a given application goes through a number of phases. These phases are described in the book by Ghenassia [20].

First the specifications of the SoC are determined based on the commercial or scientific requirements. These specifications and requirements dictate the performance and the feature set of the SoC. In the second stage the design space is explored by running theoretical simulations of the workload. During this stage it is also determined which part of the application will be implemented on which kind of hardware (ASIC, DSP, Reconfigurable tile) or if it will be implemented in software.

From this stage onwards the design teams are split into separate hardware teams and software teams. The hardware teams set out to integrate the IP blocks and to create the new blocks needed. While software teams set out to create the algorithms needed for the data processing part, the control part of the application and the software glue to tie it all together. In an ideal world there is plenty of communication between both teams to ensure that the integration between hardware and software is seamless.

It is at this stage the first challenges begin. Since hardware without any type of software is not going to do anything, new software has to be written for the new chip or old software has to be ported. Since the chip is still in development

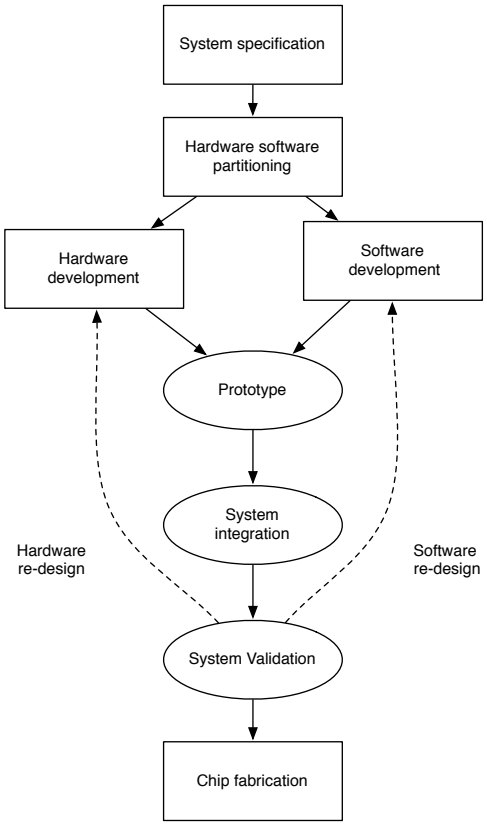


Figure 1.3: **Soc design flow**

it does not exist yet so the obvious solution is to simulate the hardware at different levels of detail. The hardware team run very accurate simulations that simulate everything the chip is doing with so-called gate level simulations. The software team uses less accurate simulators namely instruction set simulators which just simulate the programmer visible details but in exchange are faster.

Since often the data processing and the control part of the software running on the SoC are of different architectures, the complete software stack is often simulated in an RTL simulator. This has the major disadvantage that these simulations contain too much detail for software development. To get some feeling of how the actual hardware will function the hardware is prototyped with the help of FPGA's. The integration of the hardware and the software often leads to redesigns of the software and hardware. Since the software might have become dependent on the timings of the simulation and has unexpected behavior when operating on the real hardware, while the hardware might have subtle bugs that are triggered by the software.

One of the bigger problems with simulating new and advanced SoC's is that the number of cores often exceeds the number of cores in the system hosting the simulation. Currently most simulation environments do not take advantage of

multiple cores, since it is harder to get repeatable results and the overhead of keeping all the cores in synchronization can become large when the workloads are interconnected.

1.3 Simulation in general

While simuxlation is currently a mature subject in computer science, this does not mean that no research is being done into improving simulation. One area of research is to achieve a better balance between speed and cycle accuracy. There are many types of simulation possible depending on the level of detail required. These approaches span from low level (simulating all the gates) to high level (functional simulation of the program).

At the moment there are a number of techniques available for simulating processors

- Gate level
- Register transfer level
- Instruction set simulators

Gate level simulation is the most detailed and most true to actual hardware. In a gate level simulation all the details are simulated ranging from the delays in the gates to the exact timing of a system. In even more elaborate simulations power consumption can also be taken into account to get a realistic overview of how the future chip will perform. This is often done in the later stages of development of a chip since running such simulations is very time consuming due to the level of detail.

The Register Transfer Level (RTL) approach simulates the whole chip down to the the digital logic, but physical elements like propagation times are not simulated. This allows a design to be tested quite extensively for logic design errors but unlike gate level simulation, it abstracts from the realities of a real chip. However the level of detail is still too high to do effective simulation of software running on such a platform. On a real system not all the state is visible for a programmer so these kinds of simulations often provide too much information for software development.

An example of an RTL simulation is ModelSim [37]. ModelSim simulates a whole digital design down to the actual register level. Since it can be used to simulate a whole processor and is often used to get a precise indication of how a new processor will perform in real life. This technique produces the closest result to real life, with the the disadvantage of being slow due to the amount of detail provided by the simulation.

Most RTL and gate level transfer vendors provide an integrated suite which contains tools to do both kinds of simulations. But both kinds of simulation

suffer from the same set of drawbacks, since running real workloads on them is often too slow to do software development on real life designs.

1.3.1 Instruction Set Simulators

Interpretation and compilation can usually be grouped together as instruction set simulators. An ISS is a computer program that simulates enough of an instruction set and the surrounding programmer visible detail to run programs designed for that computer. In some simulators just a subset of the architecture is simulated while other simulators simulate enough of a system to run an operating system with applications on them.

First it is important to know that there are three types of instruction set simulators, namely:

- Source interpreters
- Binary interpreters
- Compiled simulation

The first type of simulation takes in the assembly files. The assembly instructions are interpreted from their textual form that might include some high level constructs like labels, register names and maybe some extra debug statements. The interpreter maintains a model that includes the registers and the memory of the system. This allows the whole state of the system to viewed at each step, which allows the user of the interpreter to get an understanding what the program does. The main usage of these kinds of simulators is to debug the initial programs of the project. This approach is quite slow since each instruction has to be parsed and interpreted. But it these kinds of interpreters have the advantage of being more user friendly for programmers since they give a more familiar perspective on the programs that are running. This perspective is more familiar since the developer sees the program interact on the level at which they are programming. An example of such a simulator is the MIPS assembly simulator MIPSim [39].

Binary interpreters accept binary instruction images as their input. The instructions are decoded and interpreted one by one in such a fashion that the the simulated results are identical to results achieved on real hardware. This technique is often used in cases where the architecture simulated is very dissimilar from the target architecture or when trying to simulate a large number of processors. This technique allows for precise simulations because all details can be simulated in software. In general, binary interpreters are faster than source interpreters since binary execution images are simpler to decode than a textual representation. But in general they are more complicated to use since high-level details are stripped, making it more difficult to see what is happening. This can be overcome by using a debugger that has some knowledge about the source and how the application was mapped to it. An example of a binary simulator is the Xentium simulator made by Recore Systems[50].

Compiled simulation is a technique that can be used to increase performance. What happens is that the assembly file is translated to the native execution format of the underlying host. This allows for a speed up since the conversion only has to happen once. Some simulators and virtual machines will compile the most often used parts of the simulated environment while allowing less used or more complicated parts to be interpreted. This results in a speed up because code normally sticks to well defined code paths and will only occasionally sway from them. Examples of virtual machines that use binary compilation are the Java HotSpot virtual machine[46], VMKit [19] and Qemu [2]. Another example of binary compilation is Recore's libMontiumC which compiles montiumC to native x86 executables. This gives the user a fast functional simulation of the application [52].

The trend of SoC's had lead to many solutions being proposed since there is a need to simulate early on in the design process. Popovici et all. [48] take a look at the simulators that are commercially available for the development of SoC's. They show that the major tooling suppliers provide pieces of the solution, some provide tools to join different building blocks together via user friendly tools while others just claim support for interoperability via Transaction Level Modeling (TLM). They also show that the directions for future research could be: increasing performance, automatic generation of simulation models from off the shelf building blocks and simulators that allow a tunable level of detail. Directions for increasing performance include looking at improved just in time compilation techniques and looking at different levels of abstraction in the simulation.

1.4 Handfish

In this report we will describe the Handfish framework, which is a simulation framework that allows a heterogeneous multi-processor SoC to be simulated by composing a simulator from different instruction set simulators. The framework is build on the concept of cooperating processes that communicate with each other via network patterns. Another feature of the Handfish framework is that it allows the strictness of the simulator to be changed, meaning that the points when the simulator synchronizes are a configurable property of the system. If more advanced synchronization is needed the Handfish framework also allows for the scheduling strategy itself to be changed. These features give the user of the framework the choice to decide between more accurate simulations or more performant simulations while taking advantage of the underlying multi-threaded nature of the host system.

In this report we will show the architecture of Handfish, how it performs and that it can be used to create varying types of simulators ranging from dual core homogenous simulators to many core heterogeneous simulators.

1.5 Conclusion

In this chapter we looked at the basics of SoC development and at the basics of simulating a processor. In the following chapters we will use the concepts first described in this chapter and extend on them via existing literature and own research.

In chapter 2 we show what is the state of the art in simulation techniques. This includes a look into multi-core simulation frameworks and configurability. Chapter 3 presents the requirements for the to a proof-of-concept simulator that should address some of the shortcomings of current approaches. In that chapter each of the requirements will be matched with a verifiable goal to verify the proposed and implemented solution at the end. Chapter 4 describes the principles of multi-core execution in depth to form a basis for the next chapters.

Chapter 5 describes the basic design of a multi-core simulator that satisfies the demands presented in chapter 3. In chapter 6 we take a look at the how the Handfish framework performs with the Simplium simulator and in chapter 7 the framework is tested with a real of the shelf simulator. Afterwards we look at results as a whole in the conclusion in chapter 8 and in the discussion in chapter 9 we take a look at questions raised by this research.

Chapter 2

Related work

Simulation has been around in multiple forms for about 40 years in practice and even longer in theory. Turing equivalence states that any Turing complete architecture can simulate another Turing complete architecture. This means that given enough memory and computation time any computation that can be done on a complicated computer can be done on a simpler computer or the other way round. The earliest use of an ISS was to provide backwards compatibility with older machines. An example of this is the simulation of the IBM 1401 mainframe on the IBM S/360 in the seventies. Later on the concept was applied to software development for new hardware, architectural research, hardware partitioning and to ease hardware architecture transitions.

Simulation and virtualization are two related concepts and often techniques can be shared. Virtualization is simulating the same architecture as the host architecture. While simulation can also be simulating a different architecture from the host architecture. The big difference is that a simulator is slower because an actual translation step between the guest and the host is needed. A virtualizer can get away with some carefully intercepted instructions or even using special hardware support if the architecture supports the Popek Goldberg virutalization requirements [47], but an in depth discussion of these requirements is outside of the scope of this thesis. Current desktop and server processor have dedicated hardware support for virtualization that allows virtualized hosts to reach nearly the speeds of direct execution. Examples of this include Intel's VT-X [44], AMD AMD-V [6] and SPARC's hyperprivileged mode [56].

While a simulated guest can be a completely foreign architecture like a simulator for VLIW running on a Reduced Instruction Set Computing (RISC) host. Depending on the difference between the architectures and the level of detail of the simulation the speed can be quite different. For example a high level simulator for an ancient architecture can be many times faster than the real hardware while a very detailed simulator for a new processor can be many orders of magnitude slower than the real hardware.

2.1 Multi-core simulation

Most instruction set simulators start out by simulating a single instance of a processor and often multi-core support is an afterthought. But recently a number of simulation frameworks have started to appear that take in to account the fact that multi-cores are becoming increasingly common. A few examples of multi-core simulators are SlackSim [11], Graphite [38], GEMS[35] and the work at National Tsing Hua University [58] [59].

In some cases, the detail of the simulation is limited to allow a larger work set to work on the simulator. In such cases often the workload of the simulator is described statistically. These kinds of simulations are most often used to explore the architecture initially since they give some insight into the theoretical performance.

The first simulator described in depth is Slacksim. It is presented first since it provides concepts that can be used to explain what strictness means in terms of simulation.

2.1.1 Slacksim

Slacksim implements an 8 core system that is simulated with 8 threads[11]. They begin their analysis by defining the concept of time in their simulator. In a multi-core simulator there are multiple "times". First there is the global time which describes the state of the whole system, there are the local times of the individual processors. And there is always the time on the clock. Slacksim simulates a maximum of 8 SimpleScalar PISA processors.

To describe these concepts they use so called time diagrams. In figure 2.1 an example of time diagram is given. In the time diagram the time on the clock is displayed along the X - axis. The horizontal lines represent simulated cores, and they are named sim #1 until sim #3. The points and vertical lines represent a single point in simulated time. In the picture they are named t1' to t4'. The figure shows that this simulation is initially synchronized. But later on the simulators start to go astray. For example simulator 2 t4' occurs earlier than simulator 3 t3'. This situation could lead to unexpected behavior if the applications executing on the simulator depend on the ordering in which read and writes occurs.

In their paper they sketch the violations that are possible in a simulated environment. They sketch out the following violations:

- Simulation state violations
- Simulated system state violations
- Simulated workload state violations

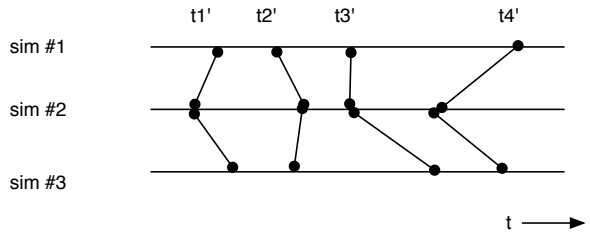


Figure 2.1: **A simple time diagram**

Simulation state violation are violations where events happen inside the simulator in a way that would be impossible in real life. An example of this is that two processor that are temporally in synchronization write to the same bus at the same time. These kinds of violations usually result in a different access order than on a cycle accurate simulator. Such violations are not observable from the code directly.

Simulated system state violations are not visible to the software running inside. Meaning that the software running inside the simulator can not detect that it is not running on real hardware. These might include violations where things like caches are in a different but not observable state than in a real system. Thus the hardware state of the simulator does not match with what would happen in real hardware.

Simulated workload state violations are violations that are visible from inside the software running on the simulator. An example of this would be that a thread sees a write that happened in the future according to its local time perspective. These are the most serious violations. These violations cause programmer visible errors in the simulation. Often these errors can be detected fairly easily by comparing the results against the result obtained from real hardware.

Violations like simulated workload state violations can be potentially hazardous since wrongly synchronized software might get into situations that are not possible on real hardware. An even more treacherous variant is that the software functions correctly on the simulator but fails on hardware. Well synchronized software will continue to function but will have slightly different behavior. An example is that a shared locking algorithm has to spin for more cycles to enter a critical section while running in a simulator. The effects on badly synchronized software are far more unpredictable, since the hidden data ordering dependencies might cause problems. They might cause the synchronization to break, which can lead to problems like non deterministic data corruption.

In their paper they mention a few techniques available to synchronize the simulation threads:

- Cycle by cycle
- Quantum based
- Bounded slack

- Unbounded slack

They describe slack as the difference between local time on the slowest simulator instance versus the local time on the fastest simulator.

In the cycle by cycle technique (also know as lockstep), the threads are synchronized with a barrier at the end of every cycle. This will result in accurate results but has the disadvantage that if the workload over the threads is asymmetric that certain threads will have to wait longer until they can continue because the other threads have to finish their processing.

The quantum based technique synchronizes the instances after a given number of cycles. This technique has the disadvantage that it allows simulation violations but has the advantage that less time is spend waiting for the threads to synchronize. The accuracy can be set by changing the length of the quantum.

Bounded slack is similar to quantum-based in the sense that that the slack of a single simulator instance will not be bigger than a set maximum. They define a window of how far ahead any instance may be from the global time, whereby the global time is defined as the lowest local simulator time. If some threads are too much ahead of the global time these instances have to wait. When the slowest instance advances, the window also advances so the other instances can continue. This is different from quantum based since in quantum based techniques the synchronization point is fixed, and the threads will wait until they all reach that point while in bounded slack only the fastest threads wait for the slowest.

Unbounded slack is the technique of not using any synchronization in the simulator itself and letting the code running in the simulator handle the synchronization. This scheme does give the maximum performance but it has the disadvantage that it will create a large number of violations since the simulator does not synchronize itself.

2.1.2 Simulating a 1000 cores

An example of a simulation framework designed to simulate a massive multicore computer is given by HP's Moncheiro. [41]. In their paper they sketch out a simulator that can simulate one thousand cores. They are able to do this by separating the functional behavior from the timing behavior. But they lack the ability to simulate IO behavior like for example a UART. Their focus is to see how certain pieces of code would scale when they would run on systems with around a 1000 cores. In their simulator they simulate up to a thousand x86-64 cores using a modified version of AMD Cotson [8]. This allows the simulation to run real life workloads like operating systems and benchmarks.

Their simulator consists of a functional timing model and separate processor model. In their simulation the functional timing models feeds the separate processor model instructions. The processor model executes in parallel, the functional model adds instructions to the processor model. This approach has

the advantage that it is quite simple to get synchronization to function correctly, since they can simply halt cores until everyone is at the right point in time.

Synchronization constructs are handled by annotating the instruction stream and waiting until every simulator reaches the synchronization point. But the system is not able to execute polling loops, so in the benchmarks these loops are removed to get a fairer overview of the performance.

This approach has the advantage that synchronization constructs can be detected and executed correctly. Because this approach also handles the timing information correctly, it allows the simulator to be used to get an idea how much faster an application will perform when running on a large scale multi-core system.

2.1.3 GEMS

GEMS (General Execution-driven Multiprocessor Simulator) [35] is a modular simulation framework where the system is divided in two parts. A memory interconnect simulation and a processor simulator. In their simulations they simulate any number of SPARC V9 cores.

The processor simulator is split up in two parts namely a highly accurate simulator for the most common instructions and a functional simulator for rare instructions. This allows the GEMS to to run real code that uses uncommon and hard to implement instructions while still being able to run cycle accurate most of the time. This allows it to run full operating systems and application benchmarks.

The memory subsystem of the simulator was designed so that it can be replaced with compatible components that can be used to simulate everything from a NoC interconnect to a bus system with caches.

2.1.4 Graphite

Another example of a simulator designed for multi-core simulation is Graphite from MIT [38]. What makes graphite an interesting system is that the system was designed to simulate very large multi-core machines, in the order of thousands of cores. What makes Graphite different from the previously described system is that the goal of Graphite is to simulate a large multi-core system with an interconnect, while the goal of the previous project was to see the effect of large multi-core systems via simulations on applications. Graphite uses an innovative approach that in the sense that it does not use a full interpreter. It uses PIN [1] to modify the running binary so that all instructions and operations that are affected by multi-core execution are intercepted and interpreted in software. This means that all memory accesses instructions are intercepted while operating systems operations like thread creation are emulated by the system. The underlying simulation only simulates how the cores are intercon-

nected. Graphite does not have inherent limitation to the number of cores that can be simulated since the simulation can even be split over multiple machines.

What makes the Graphite approach interesting is that it allows unmodified x86 binaries to be executed on simulated large multi-core computer. Thus the results provided by the simulator are achieved with real workloads. But they can only handle applications and can not simulate a whole operating system due to the reliance on PIN. Another even bigger disadvantage of this technique is that it can not be used to simulate a different architecture.

Graphite has also been used as a platform to explore various synchronization techniques for multi-core on multi-core simulators. The following techniques have been explored:

- Lax synchronization
- Lax synchronization with barrier synchronization
- Lax synchronization with point-to-point synchronization

The first two techniques are also present in SlackSim but they use different names. Lax synchronization is equivalent to SlackSim's unbounded slack and Lax synchronization with barrier synchronization is a tunable variant of quantum based synchronization. The real contribution they make is the introduction of point-to-point synchronization. In point to point synchronization the simulation continuously checks random simulation threads and compares the time between them. Pairs of simulations instances are randomly matched against each other. They use random sampling because it is a cheap heuristic. If one thread is too far ahead of the other it will pause the thread that is running to far ahead of the others.

In their research pure lax synchronization resulted in a very scalable and performant simulator but it has the problem that it will not faithfully represent interactions between cores. So in some cases violations will not appear and in other cases they will appear more pronounced than in real life. With barrier synchronization the performance and scalability are diminished while the accuracy improves. Point to point synchronization has the advantage that it is more accurate than plain lax synchronization while only having a small impact on the performance compared to using Lax Synchronization (10 percent slower).

2.1.5 Distributed synchronization

Most of the current literature on instruction set simulator focuses on accelerating single core simulators, and multi-core simulators are often ignored. The work that has the most in common with this work is the work of Meng-Huan Wu et al [58] [59] at National Tsing Hua University. In their papers they show a distributed temporal synchronization scheme. Their approach approach consists of two different ideas. First they use static analysis of the binaries to get

the information about the synchronization points in the software and they use distributed scheduling to keep the cores in synchronization.

Via static analysis of the executables before the simulation starts they precompute the points in time where each simulator needs to be synchronized with another simulator. With static analysis they first create a control flow graph of the executable which they then use to generate the earliest synchronization point for each simulator at runtime. Each simulator then monitors the other simulators if they are reaching a common synchronization point, and waits for the other simulator to reach this point if necessary.

The authors claim that this approach can increase the performance of multicure simulations because it allows all cores to remain busy which they claim is not possible in a model with a centralized scheduler. Their approach has been tested on dual and quad core simulators.

2.1.6 Other simulators

Recore Systems[49] has a multi-core simulator for simulating the CRISP GSP platform [1]. This simulator is currently not available publicly but it uses lax synchronization in combination with blocking memory to synchronize the simulation. Thus the simulation can not completely go out of control since many synchronization actions happen with the help of blocking memory constructs. The CRISP GSP[12] platform allows the memory system to block a Xentium until the request has been handled. This simulator has been used to debug and extend a large multi-core application that used 39 cores. One of the features inside this simulator is that it can be down scaled up or down easily with the help of a configuration file. This allowed pieces of the software to be debugged in isolation. The system has been used to show that some pieces of software written for the platform had some synchronization problems.

Wieferink et al. [55] present a simulator that can use multiple ISA's at the same time. It was tested by having a MIPS processor and an accelerator interacting with each other. Their main contribution is the automatic generation of suitable instructions set simulators that can be combined with a NoC. They do not mention if the simulation can take advantage of multiple host processors. They use their simulator to build an accelerated JPEG decoder that takes advantage of the MIPS and the accelerator chip.

Other simulators exist that can simulate multiple processors such as Simics [54], Qemu [2], SimpleScalar[51]. But they all rely on sequential execution. This has a few advantages for example that each run is exactly reproducible and in the case of Simics the execution can even go backwards in time. (which can be useful in debugging some situations) The disadvantage of these simulators is that they can not use more than one thread. Which prevents a simulator from fully taking advantage of modern multi-core hosts. Another disadvantage is that because they always provide the same deterministic run they will not be able to simulate all situations that might occur in real life.

[1] The author is intimately familiar with this piece of software.

A good example of a virtual machine tuned to debugging is Valgrind [45]. Valgrind translates real life x86 binaries to a RISC like format. This allows the program execution to be monitored in great detail. The main use of Valgrind is to determine if memory allocations are correctly deallocated.

But Valgrind also includes a tool to designed to find data races and deadlocks, which they call the Helgrind tool [42]. It does this by using the Eraser algorithm. The Eraser algorithm checks the memory accesses at runtime to see if they cause races. It does this by checking if each operation to memory happens while the operations are locked. If this is not the case, the algorithm will report a data race. In Helgrind, the algorithm is extended with some knowledge about the code running on top to prevent false positives and the lock mechanism is also extended to model the bus of the system. If the operations happen while that piece of memory is unlocked it will alert the user that there is a race condition. But a race condition does not imply that a piece of code is flawed. Since for example locking algorithms depend on race conditions to function, but in general most data races in software are unintentional.

2.2 Interoperability

Another development of the recent years is that SoCs are getting more heterogeneous. Often it is the case that a simulator has support for only a single Instruction Set Architecture (ISA). This often leads to trouble when trying to write software for a heterogeneous SoC since each ISA uses a slightly different tool-chain that can be hard to integrate with other tool-chains.

In principle all blocks of IP can communicate with each other in a RTL or gate simulation. Since if something can work together in the real world it can work in the RTL simulation. This kind of simulation is usually done since the results will be identical to the actual hardware. It is however not very practical for software development since these two types of simulations will be too slow for iterative development. What is often done is to make RTL compatible models that are functionally equivalent but that can be simulated quicker.

One example of a system used to describe high level IP blocks together is SystemC. SystemC [27] is strictly speaking a set of C++ classes that provides a discrete event simulation that can be used to simulate a digital design but it is often seen as a language on its own. SystemC is nowadays often used as a sort of glue for linking many IP blocks together in a simulation. Where the glue is needed to create a workable system. SystemC ships in multiple variants as shown by Popovici [48] but most of those solutions are based on the reference implementation of the SystemC runtime.

To facilitate creating higher level models than just using RTL simulations , with the TLM library for SystemC has been developed [53]. SystemC in conjunction with this library provides a framework in which different components of different vendors can be coupled together by using standardized interfaces. The latest version of TLM 2.0 has been specifically designed to link up different

pieces of IP together.

TLM 2.0 allows the user to run the simulation at different levels of detail. TLM 2.0 defines two broad categories of timing detail namely:

- Loosely-timed
- Approximately-timed

Loosely-timed implies that the transactions are un-timed with the access protocols (like bus protocols) modeled with not so much detail. This allows parts of the simulation to run ahead of the general simulation but this time remains bounded. This prevents the simulation from becoming too loose but it also allows the simulation to perform faster. In this timing mode the events do not need to happen in order as long as it does not run too far ahead.

Approximately-timed is defined as a stricter approach but it is not cycle accurate. In an approximately timed simulation the protocols are modeled in more detail. In this timing mode all events are handled in order, meaning that the simulation occurs sequentially.

The default SystemC discrete event kernel can only run on a single core but there has been some research on making a more concurrent variant of SystemC in conjunction with TLM 2.0. An example of this is SystemC SMP in conjunction with TLM DT (Distributed time) [36]. In such a system the timing is more relaxed compared to loosely-timed but the simulation can run in parallel. This allows the SystemC simulation to take advantage of having a host computer with multiple cores, which in some cases can increase the speed of the simulation at the expense of detail.

2.2.1 TRAP

TRAP (TRansaction level Automatic Processor generator) is a simulator generator designed by Luca Fossati at the Politenico di Milano. It is used for example by ESA to generate a SystemC model for the Leon2/3 processor [16]. The TRAP framework is designed to automatically generate different variants of an instructions set simulator for a specific architecture based on a high level specification written in python with some C++ for the parts of the simulator that need to be customized like interrupt handling.

TRAP only requires some architectural details (like is this architecture big endian, how many bits go into a byte and how many bytes form a word), a declaration of the registers, alias bank declarations (how are the register mapped in memory), a declaration of how interrupts are handled, a description of the pipeline and the Application Binary Interface (ABI). The ABI information is used to allow an OS to be emulated by the simulator to allow things like loading in files.

Currently TRAP can generate instruction set simulators for ARM7 and Leon processors that can operate as a separate ISS or can operate inside of loosely

timed and approximated timed SystemC simulations. Since these simulators use SystemC they can be used to create heterogeneous simulations. By default they can not take advantage of a underlying host parallelism since that requires modification of the SystemC scheduling mechanisms. The framework can at the moment only generate simulators for RISC like processors.

2.3 Conclusion

In terms of multi-core simulators most effort has been focused on simulating homogenous platforms. The area of simulating heterogeneous platforms has been largely left aside. Due to the increasing number of heterogeneous SoC's it could prove useful to integrate simulators together to simulate a whole SoC at a higher level than currently available to allow software to be tested earlier. Also none of the simulators discussed in this section have tried to see what are the effects of changing the simulation accuracy, which might be able to improve the speed of the simulation. And only some of the simulators discussed try to take advantage of the multi-core nature of today's host computers.

Chapter 3

Research questions

Looking at the related work we can see a number of trends in current research. One trend is that the focus has been on improving on the performance of single core simulators and on simulating homogenous multicore architectures.

The open areas in the field that we want to tackle with Handfish are the following:

1. How can a simulator be built with a tunable level of simulation strictness? (Tunable)
2. What is required to combine multiple instruction set simulators together to create a simulator of a heterogenous System on Chip? (Heterogeneous)
3. How can such a simulator be built such that it takes advantage of the multithreaded capability of the host? (Multithreaded)

These three question summarize the requirements of the framework. In appendix A the precise requirements of the system are formulated. It also specifies some secondary requirements for the framework. In that appendix the reason for each requirement is given and the means how the requirement can be verified.

Chapter 4

Models of multi-core execution

Before we have a look at the architecture of the multi-core simulator we first need to have a look at a more formal model of multi-core execution. We do this by defining a number of concepts based on the execution of a program.

Most computational cores have an instruction set that consists of three types of instructions:

- Control flow instructions
- Instructions with external side effects
- Instructions with internal side effects

Control flow instructions are instructions that control the flow of a program, designating if a program should branch or repeat a number of instructions. Examples in high level programming languages are the 'if' and 'while' statements and in low level instructions goto and branch. The handling of interrupts can also be considered a type of control flow instruction, except that in that case the control flow is changed from the outside. Interrupts outside of timed interrupts are also unpredictable since they can happen at any time while the processor is executing.

Instructions with external side effects change the computing environment outside the processor. Examples of such operations are loads [1] and stores to memory or in older processors I/O instructions.

Instructions with internal side effects change the state of the processor but do not change the state of the surrounding computing environment. Examples of these kinds of instructions are mathematical operations on registers.

[1] Technically load instructions do not change but environment but to execute a load, the environment has to be queried. This may or may not have side effects.

There are processors that mix and merge these types of instructions into a single instruction (namely CISC like processors) for the rest of this chapter we will not consider these types instructions since they are rarely used nowadays and these instructions can be rewritten into combinations of smaller instructions that would result in equivalent behavior. An example would be instruction that increments a value and writes it to memory. This can be rewritten as an add instruction followed by a store instruction.

When we have multiple processing elements there will be multiple streams of instructions. As we have previously seen these instructions can have side effects that are visible to the outside or not. For example a stream of stores is visible while continually incrementing an internal register will have no side effects from the outside.

Take for example two instruction streams consisting of two instructions. If we execute these concurrently there are six possible inter-leavings as seen in figure 4.1. This abstraction is valid because with multiple execution cores, the side effects of a computation can only be observed in a shared resource. This is usually a shared memory and due to all accesses being atomic, all operations appear serialized [2].

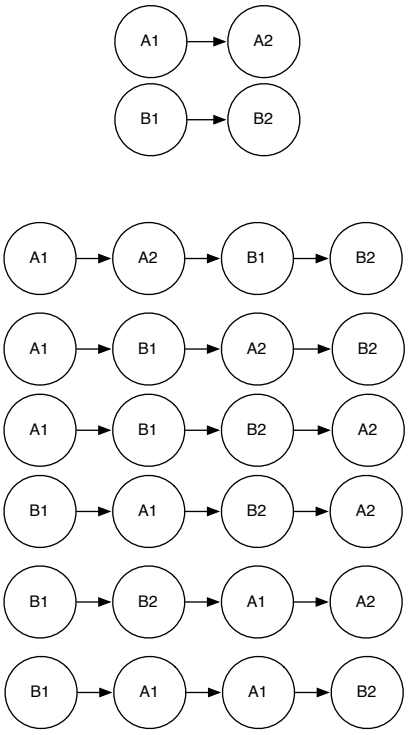


Figure 4.1: **Possible inter-leavings of two instruction streams**

Each of these traces is a valid execution but each can result in a different

[2]This assumes normal DRAM instead of specialized dual-ported DRAM like that used in graphic adapters and even those do not allow two simultaneous writes.

output. The effect is amplified on larger programs. It is also possible for two different runs of a program to produce the same output, these traces are called output equivalent. Meaning that from the perspective of an outside observer the two executions are identical.

The discussion is further complicated by the fact that instruction can have side effects on which instructions in the future depend. Data dependencies between instruction streams are especially complicated when knowledge from multiple instructions streams is required to analyze them. For example a read instruction might expect that another instruction from another core has written a value out. If this write did not happen the following control flow might change and the program might get into a state the developer did not anticipate.

Introducing control flow makes the discussion more complicated because the instruction flow can branch and jump. An example can be seen in figure 4.2 This type of graph is called a control flow diagram since it shows the control flow of a program. A control flow graph does not show an execution but it shows all the branches that a program can take. Thus is can also show dead code paths since it is a direct transformation of the program code[3].

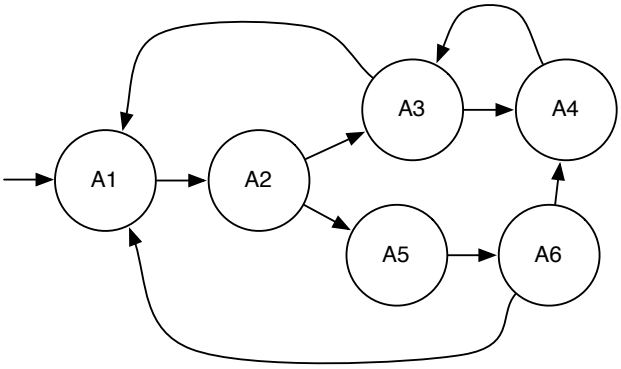


Figure 4.2: **Example of a control flow graph with branches and loops**

Control flow graphs as we have currently described them do not describe an execution of a program, concurrent or not. If we take a program that runs concurrently over two cores we will have two control flow graphs that execute at the same time. If we assume the worst case scenario the resulting graph would include nodes that are combinations of every possible combination between the nodes of the two control flow graphs and the resulting edges that connect these nodes to each other. This leads to very large graphs since the number of nodes is the multiplication of all the nodes and the number of edges is even more. Any run of the concurrent application can be seen as a walk through this graph where an edge can be passed multiple times.

This does mean that the graph contains nodes and edges that never occur during execution in hardware and some of these nodes and edges are only traversed during execution in a non strict simulator. Later on in this chapter we

[3]This could be the binary image or the assembly source code.

will formalize these graphs. The number of taken states is limited by such things as data dependencies, conditional branching and start up effects to name but a few. For example a program writes out a fixed sequence of values to memory will have less interactions in a given run than a program that loops continuously. Shared memory synchronization algorithms are a good example of such programs since they will forbid interactions of critical sections. This removes a large number of nodes and edges from the graph. In appendix D the control flow diagram of a implementation of Dekker's algorithm on the Simplium[4] architecture can be found. It shows that the control flow is more or less the same for both processes, save for a slightly different conditional in the branches. It is unfortunately not possible to generate a sensible control flow graph of all the possible interleaved states since this graph would become to large to display and generating such a graph is non trivial as is shown by model checkers like spin [26] and java pathfinder [31].

4.1 Properties of trace graphs

In our analysis we use trace graphs. A trace graph is a graph generated from the simultaneous trace of a multi-core simulator and is conceptually similar to a state space but it is not the same. We will explain the differences in depth after we have described the trace graphs.

Trace graphs are created from the simultaneous memory access traces. Since the simulators have to fetch their instruction memory this can be used as a indication what the programs are executing[5] at a point in time.

A trace graph consists of nodes and edges. Formally $G = (N, E)$, where G represents the graph and N represents the nodes and E represents the edges in the graph. Each node in the graph is consists of a tuple that describes the current executing instruction of each process at that point in the execution. So when one process fetches another instruction this leads to a different node. An edge represents that there was a transition from one node to another, implying that one core is executing a different instruction at that point. The edge can optionally contain a number that represents how many times this transition was made during the trace. We call an graph without such numbers on the edges a structural trace graph and with these numbers an ordinary trace graph.

To illustrate such a graph we have included two structural trace graphs of two cores executing. We also include the control flow diagrams to show the functioning of each program separately.

In figure 4.3 we see the two control flow graphs. Control flow graph 1 contains a choice at instruction A while control flow graph 2 does not contain any data dependent logic. In figure 4.4 we see the two possible trace graphs for a concurrent execution of the two programs.

[4]The Simplium architecture is described in appendix C and is used extensively in the following chapters.

[5]This assumes an architecture without any instruction caches The Simplium architecture is an architecture without any caches.

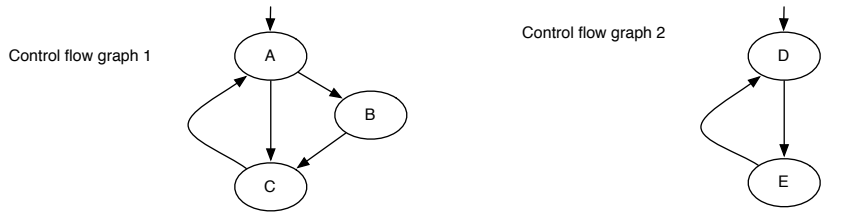


Figure 4.3: **Control flow graphs of two programs**

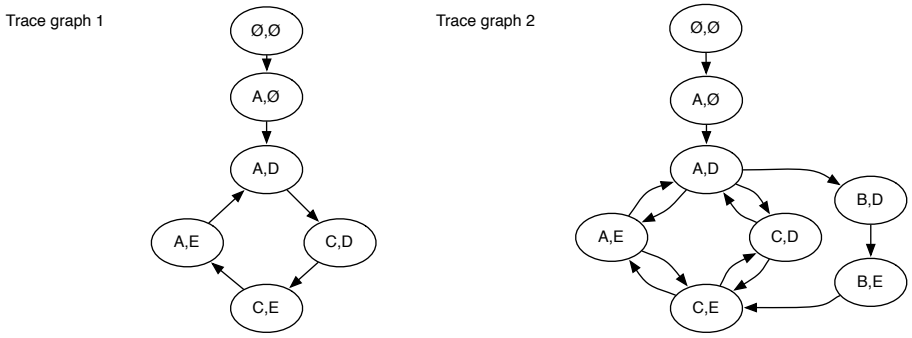


Figure 4.4: **Two possible trace graphs**

As we can see the trace graphs can be different depending on a number of things, the strictness of the simulation, the length of the simulation and other external factors like host system load.

Structural trace graphs have certain mathematical properties, namely their maximum number of nodes and edges, based on the underlying control flow graphs.

$$|N| = (n_1 + 1) \prod_{P-\{p_1\}} n_p$$

$$|E| = 2|N||P|$$

In this formula $|N|$ represents the maximum size of the trace graph in terms of nodes and P stands for the set of all processes present in the execution. To include the initial node we increment the first process node count with 1, which represents the starting node of the trace graph. n_p represents the number of instructions in the control graph of process p. So the maximum size of the trace space is the product of the number of instruction of each process.

$|E|$ represents the number of edges for the whole graph which is equal to the number of nodes times the number of processes in the simulation which is denoted with $|P|$. This is because in the worst case the next instruction fetched can be any instruction from any process. There are at most $2|P|$ edges

coming from each node because from each node in the graph another process can execute. The number of edges is limited since we can only advance one process at a time because the memory can only be accessed by one core at a time. Since a branch can jump to two location the number of edges is also multiplied by 2 since any process can be executing a branch at any time. Trace graphs also allow self edges, like a branch instruction going back to itself.

The only thing that changes for ordinary trace graph is that the edges contain a traversal count property, but this does not change the number edges nor the number nodes in the graph. This can be used to determine hot code paths in the graph.

There are a number of trace graph types. First there is the maximum trace graph, of which the properties have been described before. This graph does contain combination of nodes and edges that can never be reached. This structural graph can be mechanically generated from the control flow graphs. An interesting subset of this graph is the reachable graph which takes maximum trace graph and then strips of all the nodes that are impossible due to control flow and their dependencies. This is similar to the notion of reachability in Petri Nets where finding out which states are actually reachable is a proven hard problem [32]. With a trace from a run we can create an observed trace graph (both structural and ordinary). This graph can in theory be as big as the reachable graph but this would require a program that would run for a long time and repeats itself in its entirety.

Trace graphs are related to Petri Nets [43] and state spaces but they are not the same. The main difference lie in how they are used, Petri Nets and state spaces are used mainly to determine a priori if an algorithm can have bad states while a trace graph is used to examine the same program under different conditions with the same input. Because trace graphs do not contain information about the state of variables they can be smaller at the cost of making it impossible to do a reversible operation from a trace graph to a trace (since the transformation loses information).

Trace graphs also have the advantage that they feature some compression in the sense that repeated transitions between edges will only increment the number at the edge and not actual size of the graph, since no edges or nodes are added. This makes trace graphs efficient for comparing different runs since when comparing the two different graphs we can compare the structure of the graph and easily determine which parts of the execution are different. This is how they are used in appendix F.

4.2 Trace to trace graph transformation

Our definition of a trace is a list of structures which lists an identifier of a core and a disassembled instruction and whereby the order in which these structures are present in the list represents the order of execution of a program. In practice a trace can contain more information but this the minimum we need to create

an observed trace graph.

The algorithm described in algorithm 1 shows how a trace graph can be created from a trace.

$oldNode = array[|P|]$
$newNode = array[|P|]$
for all $traceLine \in trace$ **do**
  $newNode[traceLine.id] \leftarrow traceLine.content$
  if $containsNode(graph, newNode)$ **then**
    $addNode(graph, newNode)$
  end if
  if $containsEdge(graph, oldNode, newNode)$ **then**
    $incrementEdgeCount(graph, oldNode, newNode)$
  else
    $addEdge(graph, oldNode, newNode)$
  end if
  $oldNode \leftarrow newNode$
end for

return G

Algorithm 1: Trace graph from a given trace

The function *addNode* adds a node to the graph and the function *addEdge* adds an edge to the graph. The function *incrementEdgeCount* increments the edge count if the node already exists.

The algorithm described in algorithm works as follows. Before starting the algorithm creates two arrays with as their size the number of processes. These are called *oldNode* and *newNode* and represent the node being created and the previously created node while passing through the trace. When passing through the trace we create a new node for every new combination of disassembled instructions per process we encounter. If the edge that connects *oldNode* and *newNode* has not yet been added to the graph we add this edge. If it has already been added, we increment the count of this edge.

In appendix E we show an observed trace graph of the 'example' program that is discussed in chapter 6 and that graph is generated with the algorithm described in algorithm 1.

This technique has one disadvantage to create a trace graph in this fashion we need to access to simultaneous code fetches from the sub simulators, which limits this technique to certain architectures. This technique is mainly suitable for Von Neumann uniform memory architectures and would be unsuited for NUMA and Harvard architecture machines because it depends on viewing the whole system state in a single point.

4.3 Conclusion

In this chapter we looked at the properties of trace graphs. In the next chapter we will discuss the architecture of Handfish the proposed simulator framework based on some of the ideas presented in this chapter.

Chapter 5

Architecture and implementation

In the previous chapters we looked at what is currently available in terms of simulators, what are the requirements for a heterogenous multi-core on multi-core simulator and a theoretical model behind multi-core execution. In this chapter these are combined into a basic design of a simulation system composed of the simulation framework called Handfish and several off-the-shelf instruction set simulators.

The goal of this project is to create a software system that can simulate a heterogenous or homogenous multi-core system with the help of multiple off-the-shelf instruction set simulators. These are joined together to create a simulator of the whole system in such a way that the underlying multithreading capability of the host system is used. A sub-simulator can be an instruction set simulator or a device simulator. The "glue" to tie these simulators together is called the simulation framework. This includes mechanisms for the sub-simulators to communicate data amongst themselves and for them to remain synchronized with each other. In the text the resulting simulator is also sometimes called the system. A run of the whole simulator is called the simulation while runs of the sub-simulator are called sub simulations. In figure 5.1 the general architecture is shown together with the concepts. The way the simulator is split up in components resembles how it is done in GEMS [35], in the sense that the processing and the storage are kept in separate subsystems.

The simulation framework itself consists of the data interconnect component, scheduler and strategy. The data interconnect component is the simulation of a bus but it can also be the simulation of a NoC. Part of the data interconnect can also belong to the sub-simulators since some kinds of simulators can benefit from having a timed interactions, an example would be the simulation of a NoC, so it is partially drawn with the sub-simulators. The scheduler and the strategy together implement the sub-simulator synchronization on the side of the scheduler while the distributed waiting component implements it on the side of the sub-simulators. The data interconnect component is also the most plat-

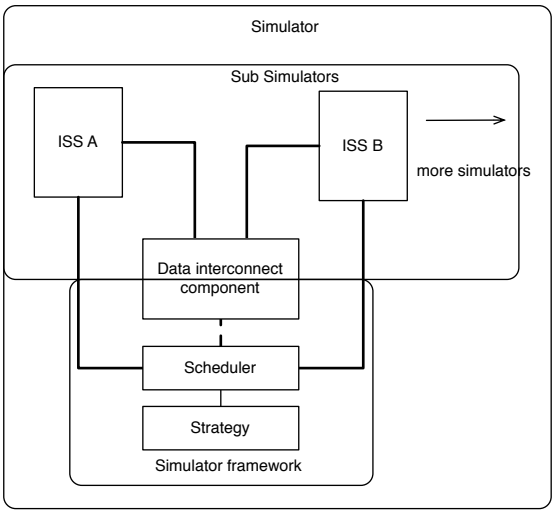


Figure 5.1: **Component overview**

form specific part of Handfish. To simulate a platform the memory map of the platform also has to be simulated. This part is handled by data interconnection component. For example is a system has a memory and a frame buffer these devices would be simulated in the shared data interconnection component.

The scheduler interrupts the main loop of the sub-simulators to ensure that they do not run too far ahead from the policy defined by the strategy. The strategy defines the synchronization point for each simulator registered with the scheduler. The scheduler allows multiple strategies to be implemented by the framework without changing the sub-simulators or the scheduler. The possible strategies range from simple strategies like synchronizing every fixed number of cycles to context dependent strategies.

The term synchronization is used often this text and can have slightly different but related meanings, so in the rest of this text we will distinguish between two types of synchronization. Namely the synchronization on the cycle level of the simulators and the synchronization of the software running inside the simulator.

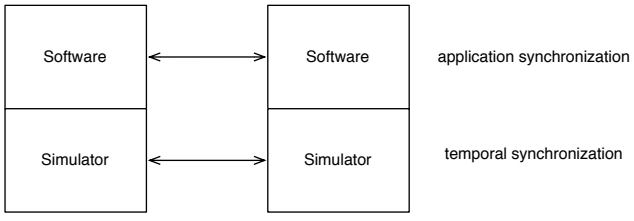


Figure 5.2: **Types of synchronization**

The first concept that needs clarification is that synchronization happens on multiple levels in a simulated system. This is shown in figure 5.2 Software

runs on top of the simulators. We call this application synchronization. The simulators are running on a host with multiple cores, which means that they also need to be synchronized amongst themselves. This type of synchronization is called temporal synchronization and is different in nature. The goal of temporal synchronization is to keep the simulators around the same point in simulated time, while the goal of application synchronization is to prevent the simultaneous access of data which can cause data corruption. In the next chapter we will look at the interplay of these two types of synchronization with an implementation of the design.

5.1 Architectural component breakdown

The framework consists of a number of components. The design of the framework is such that major components are isolated in separate processes and they do not share any state. The components only communicate with each other via messages over channels that resemble FIFO's. This is done since one of the goals of Handfish is to make a simulator out of off the shelf simulators. Because of this the number of modifications to the sub-simulators have to be kept limited, it is best to treat them as processes that communicate with each other via messages instead of having direct access to the state of each sub-simulator. This design has been well discussed in theory by Hoare [25] and has more recently been reintroduced by the ZeroMQ framework [24]. We chose this approach to simplify the design and implementation of the system.

The simulated cores are provided by instances of the instruction set simulators. The shared communication mechanism represents a bus or NOC that is used to store information and communicate between the simulators. A graphical high level overview is given in figure 5.1. The shared communication is used to communicate data between the sub-simulators, and in the current design it does not require any synchronization communication.

Another concept that is used is the concept of network patterns [24]. These are patterns that describe how processes interact with each other. The request-reply pattern, describes the interaction where one process contacts another process and gets a reply. This corresponds to the the traditional client server model. Another pattern is publisher subscriber. With this pattern one publisher sends a message to multiple subscribers. The subscribers cannot send something back over the same channel. The communication channel is then optimized for single direction multi-cast communication. These two patterns are used in the design of the simulator. There are also other patterns like for example push-pull which can be used to create pipelines of chained processes. But this pattern is not used by Handfish.

5.2 Algorithms

The designed temporal synchronization mechanism consists of two parts, namely a centralized scheduling component and a distributed cycle waiting component integrated within each sub-simulator. The scheduling components sets the pace of the simulation, while the distributed cycle waiting component deals only with the localized execution of the policy for each sub-simulator. The distributed cycle waiting components take over the main simulation loop of the instruction set simulator and for more advanced simulations it could also send information to the strategy. This approach is practical since interpreted instruction set simulators are usually written as a loop that executes an instruction for each iteration.

Conceptually we need a local time for each sub simulation. Each simulation will have its own time and there is of course a whole system time. The whole simulation time is determined by scheduling component. The policy describes how much the local times may vary from the central system time.

The basic communication between the sub-simulators and the scheduler consists of two signals. One is the central clock signal and the other signal is the sync signal. The clock signal is broadcast to all sub-simulators and each of them needs to be received and processed by the clients. The sync signal is designed as a communication channel between the sub-simulators and the scheduler. It is used to communicate to the scheduler where each sub-simulator is at the current time and when it should synchronize again with the scheduler.

5.2.1 Assumptions

There are many interesting discussions about the nature of time in philosophy and physics, but these unfortunately fall out of the scope of this work. But to understand the algorithm, the assumptions that are made of the nature of time, need to be discussed.

Assumptions about time

- Time always goes forward
- Time is discrete
- There are N+1 times (where N stands for the number of sub-simulators)

The assumption that time always goes forward has a solid basis in reality. In physics this is referred to as the arrow of time [15]. While in principle it is possible for a simulator to go backwards in time, this goes beyond the scope of this work. We define time to be composed of indivisible atoms called cycles. Each cycle can therefore not be split up in more phases. In our simulator we assume that the each sub-simulator has an independent time, while the system has a time that trails the slowest sub-simulator. The user is free to decide how

much time a cycle represents, in general it would correspond with one tick of the system clock.

This definition does allow for a couple of interesting side effects. For example these definitions do allow for a form of limited time travel in the simulation. By time traveling we mean that information from the "future" can travel to the "past". An example of this is when two simulators are both ahead of the system time but one is slightly further ahead. The one farthest in the future writes to the simulator that is not so far ahead. In this case information flows from the future to past if we look at the represented cycle count. Depending on the application this may or may not be a problem. Applications that make strict assumptions about the timing of each operation can have unintended side effects in such circumstances.

The interpretation of time in the context of trace graphs means that we assume that the simulator is in a given node at one time. For each unit of time (cycle) we can make P steps through the graph. Whereby P is the number of parallel operating sub-simulators.

We also assume that the sub-simulators can only interact with their environment with a limited number of channels. For a processor sub-simulator that might be memory accesses for code fetching and data access and interrupt requests. This means that messages coming from these channels are the only means by which the internal state of the sub-simulator can be changed from outside the sub-simulator. The reverse is also true, the sub-simulator can not change anything in the rest of the system without going through these channels. This is desirable since the sub-simulators can come from different sources and having them interface with each other through well defined interfaces prevents hard to track down problems like memory corruption of the sub-simulator itself. It also makes the development easier since the workflow to build the sub-simulator only needs to be extended with making it Handifsh compatible instead of making Handfish compatible with sub-simulator. For example it is less work to make two different simulators Handfish compatible than it would be to make both simulator compatible with each other such that they could work together.

The algorithm to keep the simulator in the desired synchronization is composed of two pieces. The scheduler component and the part that wraps the simulator loop. First we will discuss the scheduler component and afterwards we will discuss how the main loop of a sub-simulator has to be modified to fit into the simulation framework. The algorithms are based on the assumption that instruction set simulators are designed as programs that execute the same loop over and over again and that each iteration of this loop advances the time of the simulator. This makes the approach most suited for an interpreted simulator and less suited for instruction set simulators that use just in time compilation and binary translation techniques, because these techniques no longer include the concept of time like a interpreted simulator.

Summarizing Handfish expects the following properties from a sub-simulator:

- Time in the sub-simulator can advance programmatically per discrete

units of time. (preferably cycles)

- Part or all of the memory accesses can be intercepted and rerouted programatically.
- These features are programmatically reachable from a language that has a ZeroMQ binding.

The first property and third property allows the sub-simulator to do temporal synchronization with the Handfish framework, while the second and third properties allow the the sub-simulator to communicate with the data interconnect component of Handfish. And the third properties makes sure it can communicate with the framework.

5.3 Scheduling algorithm

In this section we take a detailed look at the actual temporal synchronization algorithm. The two parts of the synchronization algorithm share many variables whose names contain *cycle*. These variables are described in table 5.1.

Variable	**Description**
$simCycle_{id}$	Cycle count of sub-simulator with identifier id
$cycle$	Represents the cycle of the system
$syncCycle_{id}$	Represents the cycle where the sub-simulator id must synchronize with the scheduler
$waitCycle_{id}$	Counts how many cycles the simulator is waiting for the rest of the system to catch up

Table 5.1: Description of cycle variables used in the algorithms

The definitions of these variables result in the following invariants for the system:

$$cycle \leq min(simCycle)$$
$$simCycle_{id} \leq syncCycle_{id}$$
$$waitCycle_{id} = (simCycle_{id} - cycle)$$

The $cycle$ always follows the lowest $simCycle$ in the system. This follows from the fact that it represents the system time, and the system time by definition the slowest. The $simCycle$ is by definition always lower than the $syncCycle$. Since it can not exceed the $syncCycle$ if we want the sub-simulator to synchronize back with the scheduler. The $waitCycle_{id}$ is simply a counter of how many cycles the simulator has been waiting for the rest of the system to catch up.

The scheduling algorithm is described in algorithm 2. The algorithm uses abstraction functions to simplify and generalize implementation dependent behavior. There are functions that implement the network patterns and there are two

functions which handle the strategy dependent computations. The algorithm uses the publish subscribe and the request patterns. The $send(BROADCAST, ...)$ implements the publisher side of the publish subscribe pattern, the $send(ID, ...)$ and $recv()$ functions implement the request reply pattern and the $strategy\_*$ functions contain the synchronization strategy, whereby the $strategy\_get\_sync\_cycle$ returns the precomputed synchronization points for a single sub-simulator. The $futureWork$ variable maps every cycle to a collection of sub-simulator identifiers. So for each cycle it is known, which sub-simulator will synchronize back with the scheduler. So it will look as follows $\{0 \rightarrow \{0, 1\}\}, 2 \rightarrow \{1\}\}$ if sub-simulator 0 and 1 synchronize on cycle 0 and sub-simulator 1 synchronizes again on cycle 2.

The algorithm as described in algorithm 2 assumes it will run forever, meaning that it does not contain any phases after it has started and has no termination phases [1]. During each iteration it will publish its current cycle count to all the sub-simulators via a broadcast. Afterwards it will wait for all the sub-simulators that are required to respond back, have completed that cycle. When all of sub-simulator check back in the *cycle* is incremented with one. Then the strategy is allowed to compute the next synchronization cycles for the sub-simulators. This is done with the function called $strategy\_get\_sync\_cycle(id)$. The *cycle* variable represents the lowest cycle in the whole simulator. If $cycleDone$ is true the scheduler will increase the *cycle* variable, meaning that all sub-simulators have responded back for this *cycle*. The completion of each cycle is announced to all the sub-simulators since the publish subscriber pattern does not allow individual messages to be send. A possible optimization would be to scan the future work for possible gaps in the cycles and not send out these messages. This is currently not being done since it would make simulation termination more complicated in the implementation.

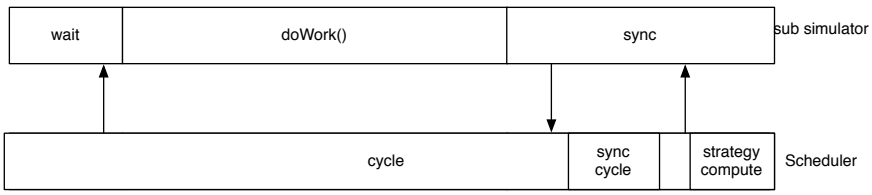


Figure 5.3: **Simplified overview of interaction between strategy, scheduler and a sub-simulator**

After a sub-simulator checks in with the message that it completed a cycle the scheduler gets the next sync cycle from the strategy and sends this to the sub-simulator. The upcoming $syncCycles$ are computed by the $strategy\_compute\_future()$ function. As can be seen the $futureWork$ data structure is modified when a sub-simulator signals it has completed a cycle and when the future work is computed. The strategy is free to specify the synchronization cycle for each simulator individually. In figure 5.3 an simplified graphical overview of the

[1] In practice it is limited by a fixed number of cycles, but this does not affect the algorithm. This is just done to measure how long a fixed number of cycles takes to execute in our experiments.

work done by the scheduler and the sub-simulator is provided. It also shows the times when the strategy is invoked by the scheduler and when the sub-simulator communicates with the scheduler.

$cycle \leftarrow 0$
while *true* **do**
  send(BROADCAST,*cycle*);
  $cycleDone \leftarrow |futureWork_{cycle}| = 0$
  while *cycleDone* **do**
    $(id, simCycle) \leftarrow recv()$
    $futureWork_{simCycle} \leftarrow futureWork_{simCycle} \cap \overline{id}$
    $syncCycle \leftarrow strategy\_get\_sync\_cycle(id)$
    $send(id, syncCycle)$
    $cycleDone \leftarrow |futureWork_{cycle}| = 0$
  end while
  $futureWork \leftarrow strategy\_compute\_future()$
  $cycle \leftarrow cycle + 1$
end while

Algorithm 2: Scheduler algorithm

The scheduler also gives each sub-simulator an unique identifier. This is done during the setup part of the simulation which is omitted from the pseudocode for clarity. The only thing that happens during the setup of the simulation is that the sub-simulators announce themselves to the scheduler and that the scheduler gives the sub-simulators an identifier. The sub-simulators then wait until the scheduler allows them to continue.

In the pseudo code the strategy is only interfaced through function calls. This means that the strategy can be changed. In the current implementation the strategy is given some time in the scheduler to precompute the future workload of the simulation, this is done in parallel with the sub-simulators.

5.4 Sub-simulator modification

To take advantage of the scheduling component of the framework, the sub-simulators need to have their main simulation loop modified. This assumes that a simulator is composed of a loop that process each instruction. The modification to the main simulation loop is described in algorithm 3.

The algorithm defines a new variable the *waitCycle*. The *waitCycle* represents the number of cycles that the sub-simulator has already received from the scheduler while system cycle count catches up with the sub-simulators cycle. Each sub-simulator maintains their own cycle counter called *simCycle*. This variable increments with each iteration of the simulation loop. The sub-simulator will signal completion of the cycle to the scheduling component when it reaches the cycle that was designated as the synchronization point. At that point it will also receive its next synchronization point.

```
syncCycle ← 0;
cycle ← 0;
while true do
  if simCycle ≥ syncCycle then
    waitCycle ← 0
    while simCycle > cycle ∧ (simCycle = 0 ∧ waitCycle = 0) do
      cycle ← sub_recv()
      waitCycle ← waitCycle + 1
    end while
  end if
  doWork();
  if simCyle = syncCycle then
    msg ← (id, simCycle)
    send(msg)
    syncCycle ← recv()
  end if
  simCycle ← simCycle + 1
end while
```

Algorithm 3: ISS algorithm

The scheduler allows the sub-simulator to run freely for a given number of cycles. After this given number is cycles is reached the wait loop is executed. This loop is triggered when the sub-simulator cycle count exceeds the system cycle count. Since cycle messages are buffered, the sub-simulator will then consume all the cycle messages from the scheduler component until it is back in sync with the scheduler.

The *doWork* function represents the place where the sub-simulator does its actual simulation work in the simulation loop. In a instruction set simulator here each instruction is processed.

The previous algorithm shows how the main loop of a compatible sub-simulator has to be modified to be able to integrate into a larger Handfish based simulator. Advanced strategies might require more modifications since in such a case the strategy might want to receive information from the sub-simulator.

In figure 5.4 a sequence diagram of two sub-simulators interacting with the scheduler is shown. The lines from the scheduler that are bold represent publisher messages while the thinner lines represent request reply messages. As can be seen the system only continues if all the sub-simulators synchronize back, but the order in which sub-simulators respond back does not matter. It only matters that all the sub-simulators respond back.

In figure 5.5 the interaction between the scheduler and two sub-simulators is shown with a strategy where the sub-simulators only synchronize every N cycles with each other. This means that one simulator can finish the workload sooner than the other one. This is exactly what happens in figure 5.5, since sub-simulator 2 finishes the workload earlier than sub-simulator 1. sub-simulator 2 still needs to process all the cycle messages from the scheduler before it can continue. In the current version of the algorithm and the implementation, all

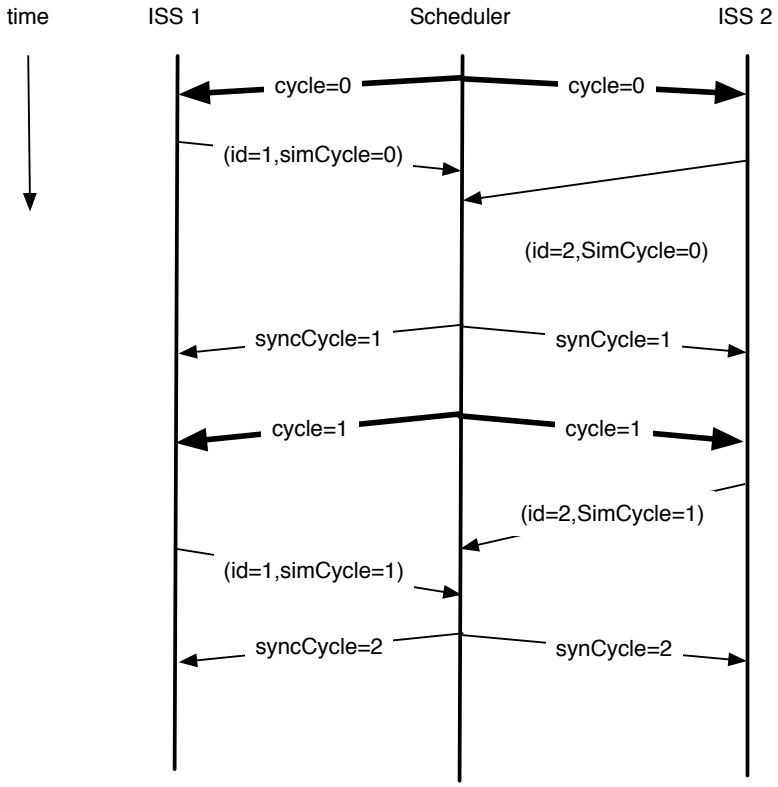


Figure 5.4: **Strict synchronization strategy**

cycle messages are send even when no sub-simulator has a synchronization point at that time.

The mechanisms of the scheduler and distributed waiting component are flexible enough to allow many different types of strategies to exist. The framework can however not preempt a simulator nor can it query a simulator preemptively. Thus in this sense the scheme resembles cooperative multitasking found in older operating systems. There is one very important distinction, the tasks run in parallel over multiple cores instead of serially on a single core. So if a sub-simulator does not yield its $doWork$ function the whole simulation can come to a standstill. This is less problematic with simulators than it is in cooperative multitasking since the workload of a sub-simulator is usually quite limited and is usually well defined unlike with general purpose systems. And even if the code running inside the sub-simulator goes into a infinite loop the simulation will not be halted, only if the $doWork$ function itself loops infinitely will the simulation come to a standstill.

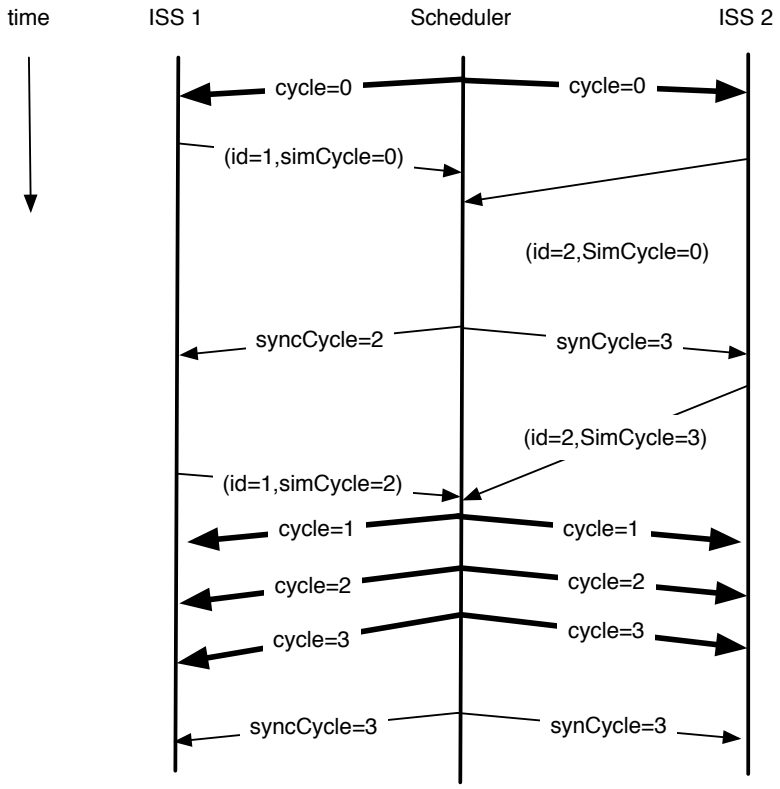


Figure 5.5: **Free synchronization with n=3**

5.5 Implementation

In the previous sections we have described the architecture from a theoretical point of view. In the next section we take a look at how the Handfish framework is actually implemented.

The framework provides the scheduler and the data interconnect. The current implementation is tested on Linux on the x64 architecture and is written in C++ 98 [7] and is compiled with Clang 2.9 compiler [34]. The system should be portable to any other system that can compile the libraries used by the framework. The external dependencies of the system are to the ZeroMQ library [24] and the Protocolbuffers library [21]. These two libraries are used to simplify the development because together they provide ready made network patterns. ZeroMQ does the communication between processes and the Protocolbuffers handles the serialization of the data messages.

ZeroMQ was chosen because it allows network patterns to be implemented in a straightforward way over multiple transports. In the simulation system we have tested Unix domain sockets and TCP. While TCP gives more flexibility to the user since it used to link multiple hosts together, Unix domain sockets have the advantage of faster performance at the cost of making all communi-

cations local to the host[2]. ZeroMQ was also chosen because it is available in many programming languages and targets many systems. This makes it easier to satisfy the requirement that it should be possible to integrate nearly any type of sub-simulator into the system. Since not all instruction set simulators are written in the same programming environment. Protocolbuffers was chosen to avoid the need to define a text based protocol. Protocolbuffers defines a serialization format and it is able to automatically generate appropriate serialization for multiple languages.

The scheduler is an implementation of the algorithm described in algorithm 3. The scheduler is implemented as a separate Linux process. The $futureWork$ set described in the algorithm is implemented as a map, that maps an integer to a list of integers. The key for the map represents the cycle and the list contains the sub-simulator identifiers. Strategies are implemented with a strategy design pattern [18] and are hosted in the same process as the scheduler. This means that a each strategy implements a common interface that the scheduler understands and that the communication between the scheduler and the strategy happens via method calls. This was done to satisfy requirement 1 in appendix A since it allows the framework to support multiple strategies without changing the sub-simulators.

The scheduler component and the strategy are included in the same process since those two are tightly coupled together since they use function calls for communication. The strategy can be exchanged by setting a command line argument to the schedule process, but adding a new strategy does require recompilation of the scheduler.

Currently there is only one strategy available in the framework. The implemented strategy is an adjustable quantum based slack strategy. It only allows the number of cycles between synchronization to be changed. So it can be used to implement cycle by cycle behavior or a completely free simulation by changing the number of cycles between synchronization moments in the simulator. It does not depend on any context from the sub-simulators to decide when it should increase accuracy or when it should decrease accuracy as it the number of cycles can currently only be set before hand.

The cycle waiting component of a sub-simulator is custom for each sub-simulator. This is because it has to modify the main loop and as such it needs to be written in the same language as the sub-simulator. For the Simplium simulator which is written in C++ the sub-simulator specific component is also written in C++. For the Xentium simulator which is written in Java [23] the component is written in Java. This sort of interoperability is possible due the usage of ZeroMQ which has bindings for several languages. Each sub-simulator continues to run as a separate process in the system. This was done to keep the interactions between the sub-simulators minimal. Since often the sub-simulators come from divergent sources this prevents them from interacting in non-obvious ways with each other. By isolating each sub-simulator in processes, the underlying operating system can at least protect the sub-simulators from each other

[2]For all experiments described in this report we have used Unix domain sockets, since we did not set out to create a distributed simulator.

at the cost of reduced performance caused by the process overhead and it allows the components to run concurrently with one another.

The data interconnect component of the framework is also implemented in C++ and simulates a byte addressable single ported memory. At the moment it is not connected to the scheduler component. This means that all requests from sub-simulators are handled immediately and only if another sub-simulator accesses it concurrently the request will have to wait before it is handled. In a future version it might also be possible for the data interconnect component to be cycle accurate. But this would require a redesign of how the sub-simulators interface with the data interconnect component.

The data interconnect component also serves as an observation tool for the applications running on a Handfish based simulator. Since in principle all communication between sub-simulators is done here, thus all side effects are measurable here. It is thus possible to generate a trace from this component. These traces can be analyzed by the tools described in appendix F.

Currently the Handfish framework includes support for 2 instruction set simulators, namely the author's Simplium simulator and Recore's Xentium simulator. The Simplium simulator has been developed in conjunction with the Handfish framework and was used to verify the core principles. For a detailed overview of the Simplium architecture we refer to appendix C. The Xentium simulator was included to show that Handfish is capable of creating a heterogeneous multi-core simulator from an of the shelf simulator. For an in-depth look at how it was integrated we refer to chapter 7.

The current implementation also does not use any process priorities to tune the performance. All process are started at the same priority [3].

5.6 Conclusion

In this chapter we have shown the architecture behind the strategy independent synchronization mechanism and we also included a simple strategy namely the quantum based slack strategy that takes advantage of this framework.

The architecture discussed here has most in common with the algorithms of Meng-Huan Wu et al [58] [59]. The difference is that in our design; strategy and execution are split and the that we use a centralized scheduler to synchronize our sub-simulators. Our approach has the advantage of scaling without having a quadratic increase in the number of connection between the sub-simulators since our approach only requires N connections. Our approaches uses $N-1$ for topological reasons plus one for clock synchronization instead of the $\frac{N(N-1)}{2}$ connections that their approach requires. Their approach requires that each simulator is connected to each other simulator. In our approach the synchronization signals are only shared by the scheduler and the sub-simulator thus

[3] As we will later see this causes some unfortunate scheduling situations.

requiring less connections instead of creating connections among all the sub-simulators. This does create a single hot spot in the simulator but this can be mitigated with smart scheduling strategies.

They claim that a centralized approach can not take full advantage of a multi-core host, which we prove to be false in the next chapter, where it shows that Handfish performs faster with more cores. Our approach allows more simulators to be connected with less connections. While it is true that the currently discussed strategies can lead to some cores idling while they wait for the completion of others. This can be prevented in our approach by using smarter scheduler strategies than currently implemented in Handfish.

5.6.1 Requirements

The architecture sketched out in this chapter should satisfy the most of requirements sketched out in chapter 3 and appendix A.

By allowing multiple strategies and having a configurable strategy the simulation strictness can be adjusted. This is specified in requirement 1. In this chapter we have seen how it can be designed in the next chapter we will look at the effect of the tunable simulation strictness. The scheduler can be combined with a different strategy and because the strictness of such a strategy can vary this decoupled strategy design allows for further tune-ability of performance and strictness.

By separating a multi-core simulator into different components namely the framework and the sub-simulators, existing simulators can be easily integrated into the resulting simulator. This satisfies requirement 3 which specified that existing simulators should be easily integrated into a resulting simulator. In chapter 7 we will take a more in depth look at integrating other sub-simulators. By splitting up the components over multiple processes the resulting simulator will automatically take advantage of the multi-threading capability of the underlying host system. This satisfies requirement 5 that the simulator framework should be able to take exploit the concurrency in the simulator if the host can provide it.

Chapter 6

Experimental case studies

In this chapter we combine Handfish with the Simplium simulator to explore how the framework functions and performs. First we discuss the methodology of our experiments and afterwards we discuss how Handfish performed and what is the effect of running applications on such simulators.

In this chapter we will focus more on general purpose IO bound loads of the simulator instead of simulating a completely realistic load of computation bound and IO bound applications [1].

6.1 Expectations

We expect that applications that are tied to the timing of a system to start experiencing problems when the temporal synchronization of the simulator is relaxed. These problems range from data corruption to fairness issues. Applications with well designed synchronization constructs are expected to work as expected.

In terms of performance we expect that the performance will increase with more cores assigned to the simulation. We expect that adding more cores than sub-simulator also has a positive effect since Handfish has more processes than just the sub-simulators. Adding more cores allows these other sub-systems to operate concurrently with the sub-simulators.

6.2 Methodology

For the experiments we used two Simplium programs and the same simulation environment of a data interconnection component that only contained memory.

[1] The following chapter will look at a more compute bound load.

The simulation environment consists of two Simplium simulators that communicate via a shared memory. The first simplium program used is a program that is only functional correct under specific timing conditions and the second program is an implementation of Dekker's algorithm [14] which is designed to be correct under all circumstances.

The first example is a program in which two cores access a memory location in an interleaved fashion and both cores try to increment the value stored at this memory location. The source code for this Simplium [2] program can be found in listing 6.1 This program is an example of a program that functions only under a strict synchronization strategy. When this program is run under a non strict strategy the application will misbehave since it assumes that each cycle occurs around the same point in time, which is only guaranteed under a lockstep synchronization scheme. We run this example program to see what are the effects of changing the simulation strictness on a unsynchronized program. In the rest of this chapter we will call this program simply "example".

```
0 DATA 1

#program Simplium 1
4 LOAD 0 0        # load address 0 into register 0
8 LOAD 1 50       # load  address 50 into register 1
C ADD 0 1 1       # add register 0 to register 1
10 SAVE 1 50      # store register 1 to address 50
14 NOP
18 NOP
1C NOP
20 NOP
24 BRANCH 8       #branch to address 8

#program Simplium 2
28 LOAD 0 0        # load address 0 into register 0
2C NOP
30 NOP
34 NOP
38 NOP
3C LOAD 1 50       # load  address 50 into register
40 ADD 0 1 1       # add register 0 to register 1
44 SAVE 1 50       # store register 1 to address 50
48 BRANCH 2C       # branch to address 2C
```

Listing 6.1: Simple simplium program 'example'

We can see the behavior of this program by looking at the memory accesses. What we will see is that the memory is accessed by both Simpliums for both data and code. By looking at the code accesses we can get a trace and by looking systematically at the data accesses we can determine if the program is

[2] For a short description of the Simplium architecture we refer to appendix C.

operating correctly. If the program is working correctly we would expected the value in memory location `0x50` to increment sequentially. It should be noted that the system was being observed. So each memory access was written out to a log file for the functional analysis. This does affect the measurement since the values have to be written out which makes the system slower. The second program used is an program that also writes a regular pattern to memory. But unlike the other program, in this program the critical section is protected by an implementation of Dekker's algorithm [14]. The actual implementation can be found in appendix D where it is presented in a control flow graph. The application is designed as an infinite loop where accesses to the critical section are protected by a protocol. Inside the critical section the shared resource is accessed and afterwards the resource is freed for usage by another core by the post protocol.

All experiments are done in the following fashion. The simulation is started and left to run for 100,000 or 200,000 cycles depending on what was tested. After that amount of cycles the simulation is stopped by the scheduler. For each setting the simulation was run 5 to 10 times and the average was taken to from these times to prevent outliers from interfering with the results. For the functional verification in appendix F the simulation ran 10 times to get a bigger data set, while the performance analysis was done with 5 samples per data point in some cases and with 10 samples per data point when it was done simultaneously with the functional analysis.

All experiments in this chapter were done on a quad-core Intel Core i5 2300 [28] running at 2.8 Ghz with 4 GB of DDR3 RAM running the x64 Linux 3.0.0 kernel. During the performance testing, all power management features of the processor where disabled since they can interfere with the benchmarks. To simulate less cores the superfluous cores where disabled in the kernel. This allowed us to get an indication on how Handfish would perform on single, dual, triple and quad core host without having to take into account other differences in host system since only the number of active cores changed while the rest of the system and software remained the same.

6.3 Functional

Before we explore the performance of the applications running on Handfish we will also take a look at how well applications function with different synchronization settings.

6.3.1 Example

We begin by taking a look at the output from the example application under various simulation settings. The processed trace of 4 different runs writing to the same memory location is show in table 6.1. Since they all output to the same memory location (namely 0x050) we have omitted the memory address in the

table. The traces include different host configurations (single-core vs dual-core) and different Handfish synchronization settings(lockstep vs 100 cycles between synchronization points).

Single-core 1 cycle			**Dual-core 1 cycle**			**Single-core 100 cycle**			**Dual-core 100 cycle**		
Cycle	id	value	Cycle	id	value	Cycle	id	value	Cycle	id	value
263	0	16	263	1	16	263	1	13	263	0	16
275	1	17	275	0	17	287	1	14	275	1	17
287	0	18	287	1	18	203	0	11	287	0	18
299	1	19	299	0	19	227	0	12	299	1	19
311	0	1a	311	1	1a	251	0	13	311	0	1a
323	1	1b	323	0	1b	275	0	14	323	1	1b
335	0	1c	335	1	1c	299	0	15	335	0	1c
347	1	1d	347	0	1d	311	1	16	347	1	1d
359	0	1e	359	1	1e	323	0	16	359	0	1e
371	1	1f	371	0	1f	335	1	17	371	1	1f
383	0	20	383	1	20	347	0	17	383	0	20
395	1	21	395	0	21	359	1	18	395	1	21
407	0	22	407	1	22	371	0	18	407	0	22
419	1	23	419	0	23	383	1	19	419	1	22
431	0	24	431	1	24	395	0	19	431	0	23
443	1	25	443	0	25	407	1	1a	443	1	23
455	0	26	455	1	26	419	0	1a	455	0	24
467	1	27	467	0	27	431	1	1b	467	1	24

Table 6.1: Output traces for example program

The traces in table 6.1 show at what cycle each core wrote which value to memory. Running the application in lockstep will result in the same output regardless of the host system but changing the synchronization distance results in incorrect output [3] or very strange temporal results. With the lockstep scenario we see that there are no simulated workload state violations as described in chapter 2 but that these kinds of errors do appear with relaxed synchronization settings. In lockstep it is however possible for there to be simulation state violations and simulated system state violations in the lockstep scenario, since within a cycle certain events can happen in a different order than in real life.

In the single core scenario with a synchronization strictness of 100 cycles we see that temporal ordering is broken since core 0 is behind core 1 in terms cycles. We also see that certain outputted values are duplicated and we also saw the outputted values jumping around. Indicating that a value was used that was read before it was overwritten by another value.

In the dual core scenario the application also produces faulty output. But in the dual core scenario the concurrency of the host at least prevents the cycle count from diverting too much.

The example shows that an application that depends on the timing can not handle a less temporally strict simulator without experiencing problems.

[3] This can also occur in systems with very loose memory models, see appendix for more information.

6.3.2 Dekker's algorithm

Unlike the example program previously described Dekker's algorithm will by design never produce invalid output on the shared resource. In our experiments we used automated testing to see if in some cases the algorithm would generate faulty output. In no case was faulty output detected in terms of output, which is expected since Dekker has been mathematically proven to be correct [14].

Well synchronized programs suffer less from the effects of less strict temporal synchronization, while programs that are implicitly dependent on the timing can start to behave functionally erratically. A quantitive approach towards the functional behavior is explored in appendix F and in the next section we will explore the performance of the Handfish framework.

6.4 Performance

In this section we take a look at how Handfish performs in various circumstances. We do two types of experiments. First we simulate a realistic dual-core application with different strictness settings. Second we do an experiment with a simple many-core application to see how well Handfish is able to simulate larger systems.

In all our tests we will vary the cycle between the synchronization points to see how the strictness affects the performance. We will also vary the number of cores made available to the simulation by disabling these in the kernel. This allows us to see the effects of different host configurations.

6.4.1 Dual-core simulator

In our dual core simulator performance analysis we use the previously described Dekker's algorithm since it is the most realistic currently available program for the Simplium architecture. We also use the dual-core simulator as a test case to see what is the effect of different simulator architecture.

We test two different simulator architectures, namely a Von Neumann and a Harvard architecture. The Von Neumann architecture does all the code fetching in the shared data memory in the data interconnection component of Handfish while the Harvard simulator stores the code in sub simulators for extra performance. The performance increases because the simulator does not have to do a context switch to fetch the executing code. In figure 6.1 we see on the left the architecture of the Harvard simulator and on the right the Von Neumann architecture simulator.

The number of cores made available varies between 1 and 4. It might appear counter intuitive that a dual-core simulator is hosted on a quad core host but we have to remember that Handfish has multiple components besides the sub

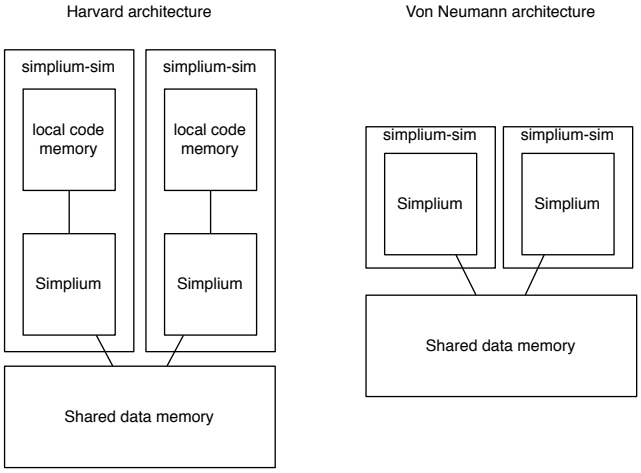


Figure 6.1: **Simplium simulator architectures**

simulator running in parallel and by adding more cores we can get some extra parallelism in the simulator since the shared components can reside on their own core thus freeing up the sub simulators cores for computation.

In figure 6.2 and 6.3 we see the performance of the Simplium Handfish based dual-core simulator. In figure 6.2 the performance of the Von Neumann simulator is shown and in figure 6.3 shows the performance of the Harvard simulator.

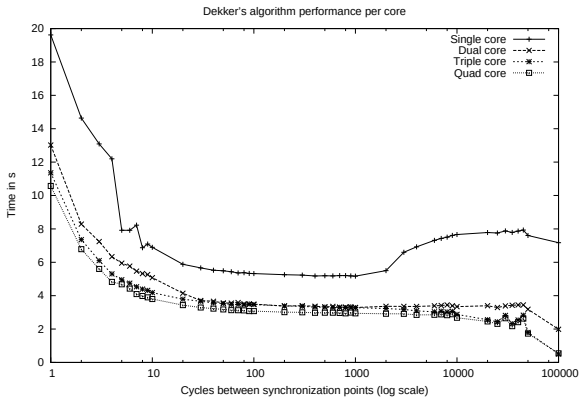


Figure 6.2: **Performance van Neumann architecture**

In both graphs it can be seen that the simulation is sped up by increasing the number of cycles between synchronization points and by adding more cores to the simulation. Increasing the number of cycles between the cycles has at first the largest effect on the performance. But this reaches a asymptote after which increasing the number of cycles will not lead to any meaningful increase in performance because other factors besides the number of cycles between syn-

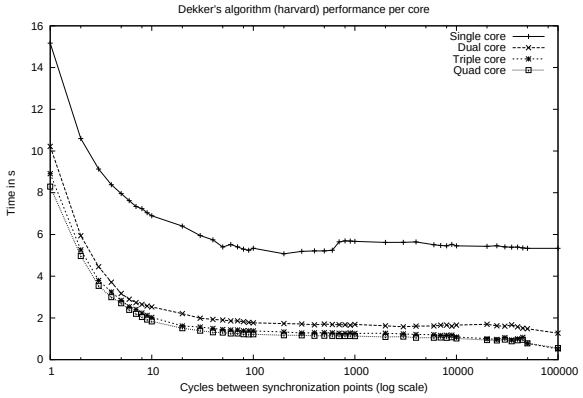


Figure 6.3: **Performance Harvard architecture**

chronization points start limiting the speed.

Increasing the number of cores will also increase the performance of the simulator. This can be most clearly seen when going from a single core host to a dual core host. While going to a triple core increases the speed further. Adding a fourth core does not increase the performance of the simulation any more for this dual-core simulator. The increase in performance between the dual-core host and the triple-core host can be explained because Handfish has two other processes running in this simulation, namely the data interconnection component and the scheduler. By having these run in parallel with the sub simulator the performance is increased since when a sub simulator requires the services of either, they can be served immediately.

The performance of the single core hosted Von Neumann simulator is peculiar at higher number of cycles between synchronization points. Instead of the simulation taking less time it starts to take more time. Some analysis into this phenomenon points the cause to the host operating system. The host operating system (Linux) aims to schedule the processes fairly if they have the same priority[40]. Processes started by the user are normally given the same priority as is the case with Handfish. But the load of the different processes over which the simulator is spread out is not fair. For example the sub-simulators and the data interconnection component do more work than the scheduler in the case of a Von Neumann simulator. If the process still get an equal amount of time they will perform slower as a whole since some processes are doing no real work while they wait for the other process who can get time because they are waiting on the other process. This was not further explored since the phenomena only appears on a single-core configuration at very loose strictness which is a strange edge case for using Handfish.

6.4.2 Multi-core simulator

For this experiment a very simple application was written, namely one that only executes NOP's and loops every 65 instructions. This application had the advantage that it can run on a multi/many core simulator and give us an indication of the performance in terms of raw instructions per second.

For our experiments we ran this application on quad-core host where we disabled the cores to allow us to determine how Handfish would perform on a system with less cores while simulating a multi-core system. The results of simulating a quad-core guest on a multiple host configurations are shown in figure 6.4 and 6.5. Each data point was computed 5 times to prevent outliers form interfering with the result. As with the dual core we ran both architectural types of simulators to see the effects of shared code fetching on the performance of the simulator. In figure 6.4 we show the performance of the Von Neumann simulator and in figure 6.5 we show the Harvard simulator. To compensate that the simulator has to do less actual work in the program we ran the simulation for 200,000 cycles instead of the usual 100,000 cycles.

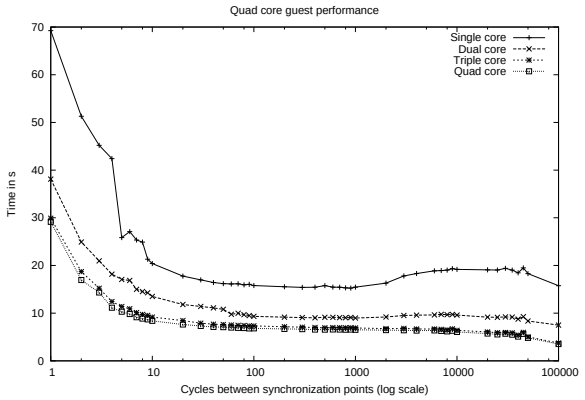


Figure 6.4: **Performance van Neumann architecture**

As with the dual-core simulator the performance increases when adding more cores. What is surprising is that the performance does not increase much when running on actual quad core computer. This appears to be caused by how the publish subscribe pattern is implemented and the lack of load on the cores since the application is just executing nops and jumps. The synchronization becomes slower because it has to send a pulse to all the sub simulator simulators and the scheduler has to wait for 4 simulators to sync back.

For the Harvard architecture the performance is slightly higher than the Von Neumann architecture but the characteristics do not change. Adding more cores still makes it perform faster but more than 3 cores does not lead to any performance increase. But we have to consider that the application does not represent a real load of the simulator, since the memory is not accessed at

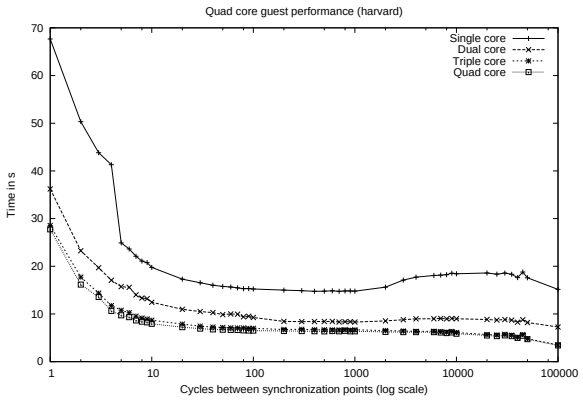


Figure 6.5: **Performance Harvard architecture**

all, and nearly no computation is done unlike with the dual-core experiments. Maybe with a more computation intensive load all the cores might be used.

6.5 Conclusion

In this chapter we have looked at how the Handfish and how simple strategies affect the functioning and performance of small example programs. We looked at different configurations of the host and different strategy settings.

The performance experiments show that Handfish benefits from having multiple cores available up to an extent. In our experiments we saw that can take advantage of a 3 cores effectively, while adding a fourth core to the simulation does not lead to a meaningful increase of performance in the configurations we tested it in.

The execution time of the simulator becomes less when we increase the synchronization time. But as is often the case in computer science the law of demising returns applies. While running a simulation in lock step is expensive in terms of time, running a simulation where the simulators synchronize every 10 cycles is far less expensive in terms of time. But synchronizing every 1000 cycles does not lead to a hundred fold increase since other factors in the simulation become limiting factors like the actual simulation work that a sub-simulator has to do and the communication time between the various other components of the simulator.

When we take the functioning of the guest application into account the results are as predicted. Unsynchronized applications behaved unpredictably with higher synchronization settings and well synchronized software behaved like expected. This shows that the choice for a synchronization setting depends

on how the application is written.

6.5.1 The host

During our experiments we tried at first to run Handfish on a virtualized multi-core. This lead to very unpredictable performance due to the various schedulers of the operating systems interfering with each other, leading it to slow down with more cores. It also lead to very sporadic results in functional measurements. Due to the sheer complexity of the this configuration it was decided to move the experiments to dedicated hardware to prevent interfering factors. We thus recommend to run Handfish based simulators on real multi-core hardware and to avoid using virtualized hardware for performance and functional measurement. Since unlike most applications Handfish derived simulator are more sensitive due to the focus on exploiting the multi-threaded capability of the host system.

Handfish is more sensitive in terms of performance than most pieces of software since it depends on the host systems operating system scheduler to provide concurrency and because the workload presented by Handfish is both computationally intensive (the sub-simulators are decoding and executing instructions) and communication intensive (the components constantly talk to each other, while running at full speed) We also saw that Handfish is not suitable for a single core host, both in terms of performance and in terms of functioning since the application running on the simulator shows more erratic behavior than on a single core [4].

The underlying operating system also plays a role in the performance of Handfish. Since a Handfish simulator is composed of a number of small processes that communicate with each other over channels like TCP sockets and Unix domain sockets which are implemented via system calls in the operating system. This design is not optimal for a single core since the applications have to wait for the operating system to schedule the process that can provide the missing data.

6.5.2 Requirements

In terms of requirements in this chapter we have seen how changing the number of cycles between synchronization points leads to functionally different results and an increase of performance. This matches the goals of question 1 which is that Handfish should allow for tunable of synchronization strictness. Since we do see a decrease in similarity from the lockstep scenario for an increase of performance when the strictness of the simulation was lowered.

We also saw in this chapter that Handfish performs faster on multi-core hosts compared to single core hosts. This means that Handfish can take advantage of

[4] For a more in depth analysis see appendix F. In the opinion of the author running Handfish on a single core with non-strict synchronization is only useful for testing algorithms in strange concurrency situations.

the multi-threaded capability of a host, although adding more than 3 cores does not seems to lead to a speed up with the current guest applications and Handfish configurations. But it does mean that we satisfy question 3, that Handfish takes advantage of the underlying multi-threaded capability of the host.

Chapter 7

Integrating other sub-simulators

In the previous two chapters we looked at basic architecture of the simulator and how it performs. In this chapter we combine Handfish with an another simulator to see if Handfish can be used to create a heterogeneous simulator. The simulator chosen for this is Recore's Xentium simulator due to the familiarity of the author with this simulator and because this simulator has native multi-core capability, which can be used to compare the performance of Handfish on the same piece of code.

First a heterogeneous simulator will be created out of the Xentium simulator in conjunction with the Simplium simulator to prove that Handfish can be used to create heterogeneous simulators and afterwards it will be used to create a homogenous Xentium multi-core simulator to compare the performance of the native Xentium multi-core simulator against the performance of Handfish to get an indication how expensive the using Handfish is compared to an unsynchronized multi-threaded simulator.

The Xentium simulator of Recore Systems is a different simulator from the Simplium. Instead of simulating a RISC like architecture it simulates a VLIW DSP and it is not written in the same language as the Simplium simulator nor the Handfish framework. The framework is currently written in C++ while the Xentium Simulator is written in Java [23]. These factors make it good test case simulator, since it is very different in its implementation and is currently in production. From a conceptual point of view it is however quite similar to the Simplum simulator since it uses the concept of cycles and it is an binary interpreted instruction set simulator. The Xentium Simulator used for these experiments is based on a pre-production version of the simulator.

Changing the Xentium simulator to function as a sub-simulator did not involve big changes to the source code of the simulator. The contents of the main loop of the simulator was isolated and put into a compatible wrapper for Handfish. This was done after porting the sub-simulator interface to Java

from the original C++. To communicate with the data interconnection components of the Handfish framework the memory sub system was expanded with a memory mapped device. The Handfish version of the simulator also has a different initialization routine compared to the regular shipping simulator since the connection with the rest of the Handfish simulator needs to be setup and the Xentium Simulator needs to be configured for operation inside the Handfish framework.

The initial bring up of the Xentium Simulator took 2 days with the source code of the Xentium simulator available and did not require substantial modification to the simulator outside of the parts that describe the world surrounding the Xentium. The Xentium simulator was fully functional except for the WAIT instruction on the C0 unit. The implementation of this instruction in the simulator is currently incompatible with the Handfish framework since it halts the simulation loop and was thus not included in the Handfish derived simulation. Other sub-simulators could also include incompatible instructions if some instruction requires that the simulator loop has to be halted.

7.1 Xentium multi-core simulator

The Xentium multi-core simulator is an internal simulator at Recore Systems that simulates a sub set of the CRISP GSP [12]. (only the Xentiums present on the GSP board are simulated) Each individual Xentium core is simulated in a separate thread. These threads are not synchronized between each other outside of interactions with special memory mapped devices which are implemented using java synchronization mechanisms. This is practical for applications on the CRISP platform since that platform has special hardware support for the synchronizing the processes running on the Xentiums. This makes it a fast system for simulating a large subset of applications written for the GSP platform.

This multi-core simulator has the advantage of being fast since it takes advantage of the underlying multi-threading capability of the host and the components communicate via each other via function calls. This prevents process context switches and system calls, which are heavier than thread context switches since thread context switches do not have to change the whole virtual memory.

7.2 Heterogeneous simulator

The heterogeneous architecture is shown in figure 7.1 and includes two Xentium simulators and a single Simplium simulator. The Simplium simulator runs control code for the two Xentiums that are acting as digital signal processors. The three processors share a data memory, and the Xentiums each have a local data memory of 4 megabytes [1] Each sub-simulator has a separate instruction

[1] A reader familiar with the Xentium will notice the lack of tightly coupled memories in this configuration. The simulation of the tightly coupled memories was removed to simplify the bring up of the simulator.

memory for performance and simplicity. This was done because the Xentium is a Harvard architecture and this approach required the least amount of modifications to the Xentium simulator. The Simplium was also given a separate instruction memory to ensure that the instruction would not be overwritten by the Xentium or the Simplium. This was done to prevent unfortunate accidents where by the processors start writing to each others code memories and corrupt each others code. While this configuration is not entirely similar to real world hardware the main goal was to see how to integrate an existing simulator into the Handfish framework. A more realistic simulator would not require more integration work on the side of Handfish but would just be more work on the individual sub-simulator side and the application to prevent the Simplium and Xentium from interfere with each other in terms of locations of code.

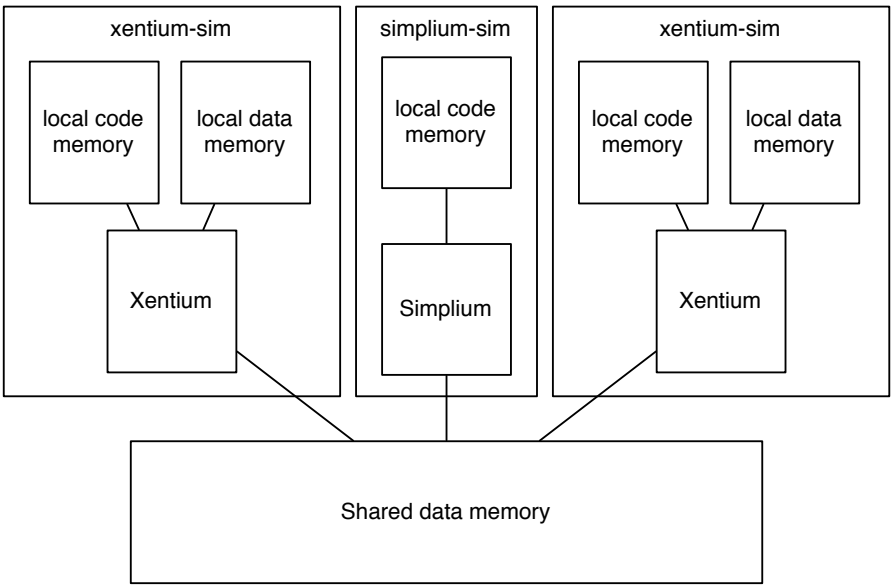


Figure 7.1: **Example heterogenous architecture**

On this heterogeneous architecture a small example program was run where the Xentiums computed the average value of two buffers which where filled by the Simplium. The Simplium also told each Xentium, of which buffer they needed to compute the average. The results of the two Xentiums was compared to see if both performed the job as specified. When the Xentiums had performed their computation the Simplium will fill the buffers with new values and the Xentiums start their computation again. The program only has one termination state, that is when the results of the two Xentiums do not match. If they continue to match the program will continue to run. The source code of the program can be found in appendix G.

With the heterogenous simulator we have shown that Handfish can be used to create heterogeneous simulator out of two different simulators that together simulate a hypothetical multi-core composed of a Simplium and two Xentiums cores.

7.3 Homogenous Xentium simulator

In the previous sections and chapters we looked how Handfish performs in terms of performance, functionally and how it can be used to create a heterogeneous simulator. But we have not yet determined the overhead that the Handfish approach brings to a simulation.

To test this we will use the Xentium multi-core simulator together with a shared memory locking algorithm as an application. The shared memory algorithm was developed at Recore systems and is designed to operate on two Xentiums on the CRISP GSP platform [12]. The data interconnect component was made compatible enough with the CRISP GSP platform so the application could run on the Handfish based simulator. The resulting simulator consisted of two Xentium sub-simulators, a custom data interconnected and the standard Handfish scheduler.

The shared memory locking algorithm is an implementation of Lamport's Fast algorithm [3] with modifications from Boehm, Demers and Uhler [9] and the same Xentium binary was run on the native multi-core simulator and the Handfish based simulator. For the Handfish derived simulator we ran the simulation at different levels of accuracy like previously done for the Simplium experiments. The shared memory locking algorithm protected the shared UART, to which both cores try to write a small message. If the synchronization is broken they will display a garbled message. The native multi-core simulator was run with the default settings. All experiments where run on a dual-core Intel Pentium D at 2.66 Ghz [29] with 4 GB of RAM running 32 bit Linux 2.6.32. In all cases the simulation was left to run for 100,000 cycles before the simulation was terminated.

Figure 7.2 show the results of the previously described experiment. In terms of shape we notice that the performance curve has the same shape as Simplium experiments in chapter 6. The observant reader will also notice comparing against the performance of native multi-core simulator we see that Handfish is an order of magnitude times slower with loose synchronization. Simulating the same simulation in lockstep is 40 times slower than the native unsynchronized simulation while running the same application. The overhead for the equivalent synchronization is explainable because Handfish separates the components of the simulator in processes and uses inter process communication instead using threads and function calls for communication.

7.4 Other sub-simulators

We also looked at other simulators to see how well they would fit into a Handfish framework. We looked at the SWARM [13] and GRSIM [5] as examples of off-the-shelf simulators that are conceptually similar to the Simplium simulator and the Xentium simulator [2].

[2]The authors also looked at a simulation of the Apollo Guidance Computer (from the NASA moon missions)[10] and this simulator is also conceptually similar enough for inclusion.

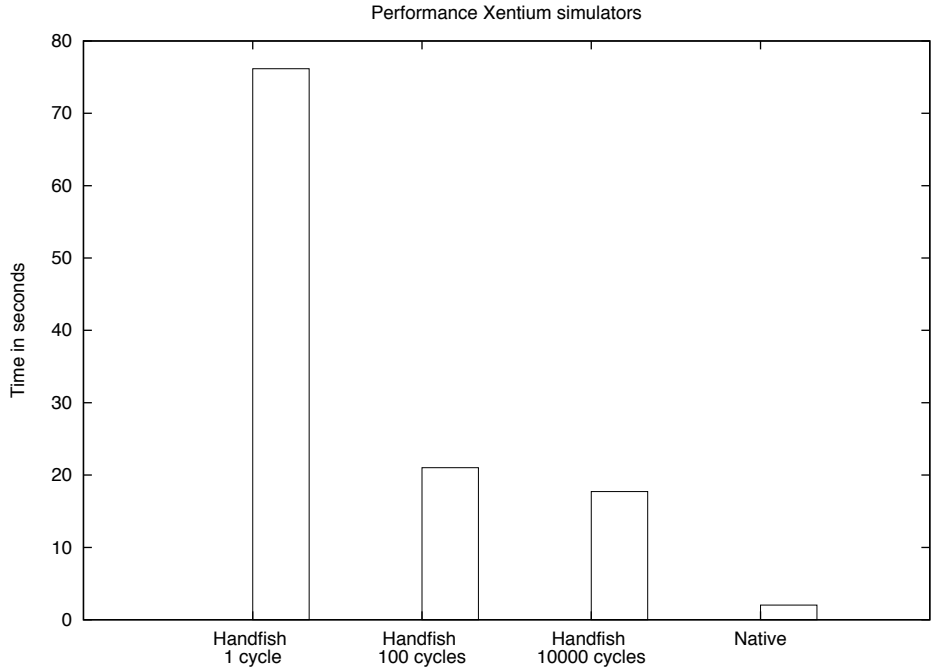


Figure 7.2: **Performance Handfish derived homogeneous Xentium simulator**

SWARM is an open source arm simulator and as such it can be modified for integration into the framework by modifying the source code. Since SWARM has the notion of a cycles it can be integrated easily into Handfish since the contents of the main loop can be put in a $doWork()$ function as described in chapter 5 and can then be surrounded with the ISS cycle algorithm described in algorithm 3. The SWARM simulator allows devices like memories and UART's to be defined via C++ classes. A compatible Handfish interface could thus be devised that bridges between SWARM's memory interfaces and data interconnect component of Handfish.

GRSIM is a closed source Leon (SPARC) simulator from Gaisler Research. For GRSIM the source code is not available, but the simulator can be used as a library inside a bigger application. While the simulator API does not include the cycle notion that the Xentium, Simplium and Swarm simulators have, it does include the concept of time. Since the concept of time and cycles are related. GRSIM could be wrapped with the Handfish framework by defining a cycle as a amount of time and incrementing the simulated time of the simulator by the time defined as a cycle. To communicate with the rest of the simulator

Although including this simulator might not be very practical.

it is required that the data interconnect component is exposed to the AHB bus simulation of GRSIM. Doing this would allow it to work in conjunction with other Handfish sub-simulators.

7.5 Conclusion

In this section we looked at three aspects related to creating a heterogeneous simulator. First of all we looked at the possibility of creating a heterogenous multi-core simulator out of multiple other simulators. This was done by simulating a systems that contained a simplium and two Xentiums. This configuration showed that Handfish fulfilled question 2. Integrating this simulator proved to be relatively simple and straight forward. We also showed that we can simulate at different levels of strictness with different sub-simulators. We also showed that we can integrate existing simulators with Handfish.

Second, we tried to determine the overhead of the Handfish approach by creating a partial simulation of the CRISP GSP platform. These experiments show that integration with the framework does lead to a decrease in performance. But these are offset by the advantages of integrating with Handfish. Since a Handfish simulation can be extended with different sub-simulators and allows the synchronization strictness to be tuned. This makes the Handfish approach more flexible than the approach used in the native Xentium multi-core processor. It is also interesting to notice that while we only used one binary of the scheduler to create three different simulators, namely the homogenous Simplium multi-core, a homogenous Xentum multi-core simulator and a heterogeneous system composed of both Xentiums and Simpliums. The same data interconnection binary was used for the Simplium multi-core and the heterogenous multi-core. This shows that Handfish is flexible tool to create heterogenous multi-core simulators.

Third, we also briefly looked at two other simulators to determine the amount of effort it would require to integrate these into the Handfish framework. We saw that some simulators make different assumptions about time than our approach. Instead of the concept of cycles they use time in microseconds. It is however still possible to interface with these simulators by using a bridge definition, that maps one cycle to a number of microseconds.

To include other simulators we need to have two things. First we need to be able to synchronize with the simulator on a cycle by cycle basis or we should be able to step through the simulation in fixed time steps. Secondly we need to allow to communicate with the data interconnection component of Handfish. Most simulators allow masters and slaves to be created that communicate with the simulator. So normally we just need to create a device inside the simulator that serves as bridge between these interfaces and the data interconnection component.

Since most computing platforms include their own memory map, to simulate such a system a custom data interconnect has to be written that maps all

devices in a Handfish compatible fashion. This was the case for the homogenous Xentium system where a shared UART like device was needed, the other configurations simply used a data interconnection component that simulated a shared memory.

If the source code of the simulator is available integration is easier, but this effect could also be achieved by using the GDB interface of a simulator to step through the simulator if that is available, since that allows the simulation to be incremented one instruction at a time. Most simulator do include the capability to be extended with extra devices and that can be used to interface with the rest of the simulation. If these interfaces are not available and the source is not available, a simulator can not be integrated with Handfish.

Chapter 8

Conclusion

In the previous chapter we looked at pieces of the Handfish framework and drew conclusions from them. By looking at each requirement as specified in chapter 3 and appendix A we will see how Handfish has satisfied these requirements.

8.1 Tunable

Handfish allows a tunable of level simulation strictness via a two fold approach. First it allows sub-simulators composing the simulator to be temporally desynchronize with each other via dynamically computed synchronization points. These synchronization points can be different for each sub-simulator. By choosing the distance of the simulation points the accuracy of the simulation becomes tunable. Furthermore the computation of where these synchronization points are located in time is separated from the enforcement of these synchronization points by the scheduler. This is done by letting a strategy decide where the synchronization points should be. The scheduler and the strategy communicate with each other via a well defined interfaces which allows the strategy to be replaced with another one. The default strategy in Handfish allows the user to set the number of cycles between synchronization points for all sub-simulators.

The goal of making the simulation strictness tunable is that we want to make trade off between performance and accuracy. In chapter 6 we saw that a Simplium based simulator allows such a trade of to be made between 1 to 10 cycles between synchronization points. Looser synchronization than that will not lead to a meaningful increase in performance. Looser settings can be useful to gain confidence that an application has been written correctly if it still behaves as expected under different temporal synchronization settings.

8.2 Heterogeneous

To combine multiple instruction set simulators to create a simulator of a heterogeneous system we propose to separate the communication between the components into two classes namely: synchronization and data communication. Handfish provides support for both types of communication.

The synchronization information keeps the sub-simulators around the same point in simulated time. This part is provided in Handfish by the scheduler and the distributed waiting component. These implement the policy set out by the strategy which is also user changeable if the user needs a custom scheduling policy.

The data communication between the sub-simulators is implemented by the data communication component, which simulates a bus or network on chip interconnection between the simulated cores.

By keeping the sub-simulator and the sub components each in their own separate process and only allowing them to communicate via channels like TCP sockets or Unix domain socket we make it easier to combine various simulators together as one simulator. This also has the advantage that the sub-simulators can remain mostly unmodified besides integrating the distributed waiting component and an interface to the data interconnect component. This means that sub-simulators can be replaced independently of the various other components. By separating the system in such a way we can combine different instruction set simulators together to create a simulator for a system on chip. We saw this with the heterogeneous Xentium and Simplium simulator and the Xentium multi-core simulator.

We also explored what would be required to get other simulators integrated with Handfish and saw that the assumptions behind the framework are workable.

8.3 Multithreaded

By isolating each sub-simulator in a separate process and only interfacing through a limited and defined interfaces the Handfish based simulator will be able to take advantage of the multi-threaded capability of the host. The underlying host operating system scheduler will automatically schedule the different processes to different cores. Thus the property that allows Handfish to easily include other simulators as sub-simulators also allows it to take advantage of the multi-core capabilities of the host. In our experiments we saw that Handfish can take advantage of to 2-4 cores but that with the guest applications and settings we used that adding more than 3 cores did not result in any speedup. We where not able to test how well Handfish scales to bigger systems or more exotic systems, since the authors did not have a suitable test suite available nor did we have suitable many core host available (with more than 4 physical cores).

8.4 Other requirements

While the Handfish framework satisfies all the requirements set beforehand as primary it does not satisfy all the secondary requirements as defined in appendix A. The only non satisfied requirement is that it does not provide an interface to a system C TLM 2.0 based simulation environment. This is however not a fundamental problem since it would simply be a matter of implementing the relevant interfaces in the data interconnect network. It was not included due to the direction of the research changing from creating a general framework to using this framework to get a better grasp of the effects of changing the temporal synchronization of a simulation and how this could be used to determine the quality of a strategy.

Handfish can also be used to thoroughly test programs in different scenario's. Since it is possible to simulate parallel algorithms under different synchronization strictnesses. An example would be to test synchronization mechanisms under very loose synchronization settings to see if the implementation is faulty. While it will not give a definitive answer if the implementation is flawless it might give some relevant insight into how the algorithm would perform since the execution is easier to analyze.

8.5 Conclusion

Handfish satisfies all primary requirements and mostly satisfies the secondary requirements. The current design does not impede the secondary requirements to be integrated into the Handfish framework, they where omitted at the current time for completeness with regards to the primary requirements.

Concluding we can say that that the Handfish framework can be used to create tunable heterogenous multi-core simulators that can take advantage of the underlying multi-threaded capability of the host system. In the next chapter we will discuss how the framework can be extended.

8.6 Discussion

While the Handfish approach satisfies the requirements set out at the beginning it is not yet a practical system.

First of all compared to a system that is composed of threads instead of processes the performance is an order of magnitude slower with very loose synchronization settings. This is caused by the process overhead and the need to do expensive system calls to communicate between the processes. It can however by be used to test synchronization constructs between arbitrary heterogeneous cores in different scenarios.

Chapter 9

Future work

In this report we have discussed the Handfish simulation framework and in this chapter we take a look at the questions that the approach raises.

There are four areas in which future research could focus:

- Functional analysis
- Synchronization strategies
- Optimizations
- Expansion

9.1 Functional analysis

In chapter 6 and appendix F we did an initial exploration of the functional implications of relaxing the temporal synchronization of the simulator. The work there could be extended further into source code visualizations of which areas are most affected by changing the temporal synchronization[1].

Even the algorithms and structures used in the functional analysis could be further extended by using more information from a run. (Like for example variable states) Thus making the analysis techniques closer to formal state spaces. The same could apply for the similarity computation, which is currently done in an ad-hoc fashion compared to the state of the art in graph similarity research. Incorporating this research could be useful in extending the idea of using trace graphs and similarities better.

[1] The authors used the concepts behind trace graphs to detect what was causing an unfairness condition in the implementation of Dekker's algorithm.

9.2 Synchronization strategies

While Handfish includes the possibility of changing the strategy. This facility was not used extensively at the moment but it might be an interesting area to see how much performance could be gained. With the similarity of trace graph described in appendix F different strategies could be compared to each other. Interesting strategies to explore are ones that are aware of the code running inside of the simulator. An example would be a strategy that detects spin locks and slows down the simulator if one core nears a spin lock.

Another interesting area of research would be to see if changing the strategy can help scale the simulator to use more cores effectively. An example would be that in a many core simulator not allowing sub-simulators to synchronize on the same cycle but to synchronization points. So in a quad core scenario, core 1 and 3 would synchronize every 2 cycles and core 2 and 4 would synchronize every 3 cycles and the whole simulator would then synchronize every 6 cycles.

9.3 Optimizations

Handfish has not been optimized extensively at the current time. It might be interesting to experiment with the process priorities of different parts of the simulator to see how that would impact performance.

Another interesting avenue of research might be to see how problematic it would be to use the algorithms from Handfish but putting the whole system in a single process and the components in separate threads and using the ZeroMQ intra process communication transport. That would save the process context switch cost and the overhead caused by system calls. For a well behaved sub-simulator this might be a good technique to increase performance but this would require more engineering effort than the current approach.

9.4 Expansion

Currently the Handfish framework only has support for two simulators, namely the Simplum and the Xentium simulator. By combining Handfish with something like TRAP [16], it might be possible to generate more realistic system architecture since trap allows the user to define a simulator in a high level language and that simulator could then easily be integrated into Handfish by letting TRAP generate the necessary sub-simulator with the needed interfaces.

Another interesting area of research would be to try and integrate a Handfish simulator into a TLM 2.0 simulation to see how well that would fit or how well part of a system could be simulated in a TLM 2.0 environment while another part would be in Handfish. This would make the work more practical for real world development since currently such systems are often put together with SystemC TLM 2.0.

Appendix A

Detailed requirements

This chapter describes the high level requirements for the simulation framework. The description contains a rationale, an importance and a verification goal.

The high level requirements define the simulation framework that will be built. They sketch the desired properties and the assumptions that underly them. Not all requirements are seen as primary, so a few are considered secondary.

The tune-able strictness requirement is seen as the most important requirement. The requirements are meant as elements that have to be taken into account in the final product for it to be effective. The secondary requirements are features that would be nice to have in a simulation framework.

Requirement 1
The simulator framework should have a tunable level of simulation strictness.
Importance: Primary
Research question: 1
Rationale
This is important since it allows the user to choose if the simulation should behave more like the actual hardware but operate slower or behave less like the hardware but operate faster. A use of this could be to determine how strict the simulation synchronization for a certain applications needs to be. Some applications experience different behavior with different levels of strictness. Strictly timed applications using low-level synchronization can be especially vulnerable or applications that are dependent that events happen in a certain predefined order. Since differences in the strictness of synchronization can throw these kinds of applications off. Most well designed applications should not experience any functional difference when operating in a different environment.
Verification
This requirement can be satisfied if a known sensitive application can show different behavior at different levels of strictness. A good example would be a program that uses shared memory synchronization. At a very loose setting the temporal behavior might be very different and depending on how sensitive the application is, it might even fail.

Requirement 2
The framework should be able to link up different instruction set simulators, that simulate very different architectures.
Importance: Primary
Research question: 2
Rationale
This is important because the current MPSoC's are getting more and more heterogenous. Often there is no way to simulate these kinds of SoC's without resorting to low level simulations like ModelSim. A fast functional simulation early on can improve how software for such platforms is written.
Verification
This requirement is satisfied if the simulation framework can run an application that uses multiple architectures. An example can be DSP application that runs the numerical intense processing on a specialized DSP architecture while the control part runs on a general purpose processing architecture.

Requirement 3
To use the framework, existing and future simulators should not require extensive modification.
Importance: Primary
Research question: 2
Rationale
It is inevitable that some modifications are needed but they should be kept as limited as possible. We aim to take advantage of the fact that the work of creating an ISS has already been done.
Verification
This requirement is more of a guideline instead of a hard requirements. But a reasonable measure would be that the modifications should not be more than a couple of hundred of lines of source code for the more advanced synchronization strategies.

Requirement 4
The simulation framework should allow multiple synchronization strategies.
Importance: Primary
Research question: 1
Rationale
To compare the effectiveness of the new algorithms the results need to be compared fairly. This can be done by implementing the synchronization techniques from SlackSim and Graphite in the simulation framework so they can be compared against the new algorithms.
Verification
This requirement can be met by implementing multiple synchronization strategies and comparing them against each other to gauge their comparative performance.

Requirement 5
The framework should be able to take advantage of multithreading capability of the host if that is available
Importance: Primary
Research question: 3
Rationale
This requirement allows the simulator to have scalable performance. It allows the performance of the simulation to be improved by adding more cores.
Verification
This requirement is satisfied when the simulation running on the framework shows a speed up when adding more multi-threaded resources (like cores) to the simulation.

Requirement 6
The framework should be able to accommodate various different types of system components that are not processors. e.g. DMA controllers and NoC routers.
Importance: Secondary
Rationale
A SoC is nowadays contains more than just a processor and some memory. So the system has to be able to accommodate these devices to simulate a full platform. Often there are models available from the vendors and these can be integrated into the simulation. This can for example be done by making the simulation framework compatible with SystemC and TLM 2.0.
Verification
This can be verified by integrating the framework with a device model written in the TLM 2.0 standard and having the application inside of the simulator communicate with this model.

Requirement 7
The framework should provide the user some insight into tricky situations like deadlocks and data races.
Importance: Secondary
Rationale
The most difficult problems in multi-core / multi-threaded programming is that programs can produce different output for the same output and the different outputs are hard to reproduce. Pinpointing these bugs in source code is often very tricky, but pointing out potentially tricky situations at runtime is less difficult since there is some runtime context available.
Verification
A way to check this requirement would be to run a known defective algorithm on the simulator. The simulation framework should at least point out that something is wrong when running this algorithm. Good examples of known defective algorithms are failed attempts to implement shared memory synchronization.

Appendix B

Memory consistency models

Computer systems in general have a model that describes how interactions with memory are managed. Each different type of processor and programming language specifies a slightly different memory model. Usually these differences are then handled by the compiler by adding special instructions to handle different use cases.

A memory consistency model is specified by describing for each memory instruction what the constraints are for that instruction. Some architectures include various operations that are less loose to allow in order memory access. Other architectures implement a memory barrier instruction, which has to be inserted around the areas where the operations need to happen in order.

Well written programs should be insensitive to these problems. The advantage of having a looser memory consistency model is that it allows for hardware optimizations at the cost of determinism at the programmer side.

To get a better understanding of memory consistency models we first need to discuss the effects of parallel memory accesses. A good overview of memory consistency model is given in the technical report by Sarita et al.[17]. When two processor access the same memory location simultaneously this can create a data race. This means that the further execution of the program is affected by the order in which the two operations happened. Different types of memory models can cause such a situation to result in very different outcomes. Most programs should not be affected by these races since most software can handle such cases without any problems. But programs that use shared memory synchronization are affected since algorithms like Dekkers' [14] and Lamport's Bakery [33] rely on data races.

There are different types of memory consistency models. Some models strictly maintain the order of read and write operations in exactly the order they where issued. There are also other models that do not make this assumption, and allow memory operations to be cached and/or rescheduled at a different time.

Relaxed memory models have the advantage that they allow more optimizations in the memory systems and they allow for out of order execution scheduling of instructions. Within this class of memory models there are multiple variations namely:

- Relax write to read program order
- Relax write to write program order
- Relax read to read and read to write order
- Read others' write early
- Read own write early

These relaxations can be selected piece by piece. But it is important to remember that while most code is not affected by these relaxations some code (like a synchronization algorithm) is and that for that sort of code there needs to be a way to make memory operations synchronous. An example of such are memory barriers found in the instruction sets of various processors. But in some schemes it might even be doing a read-modify-write instead of only a read.

The first three orders are only valid on the same memory location while the other two are for the memory as a whole.

Relax write to read program order implies that reads can be treated less strictly than writes. An example is the Total Store Order where a processor can see its own writes earlier while a write from another processor is only visible to a processor when it is visible to all other processors.

Relax write to write program order implies that the relative ordering of writes can be done differently. An example is the Partial Store Ordering model of the SPARC V8. In this model writes to different locations can arrive at the memory out of program order.

Relax read to read and read to write order is the combination of both techniques above. This means that the that both reads and writes can happen out of order.

Read other's write early means that the processor can use the data of an other processor's write earlier than the rest of the system. Read own write early means that the processor can use the data from its own writes earlier than the rest the of system.

A discussion about memory consistency models would not be complete without a few words about memory barriers. Since memory barriers are the principle tool of getting memory ordering correct on modern out of order processors. Memory barriers serve as an mechanism to circumvent optimizations for the situation where sequential memory accesses are required. A memory barrier is a memory instruction that does not read or write to memory. It signals that the the memory should be sequentially accessed at that point. For example a write

barrier implies that writes have to be at least in progress by that point. These barriers play a crucial role in memory consistency models since they allow the programmer to escape the looseness in cases where it might be required. Examples include shared memory synchronization and reading and writing to and from registers of devices.

Appendix C

Simplium simulator

The simulator used to do experiments with is a home grown simulator for an home grown RISC architecture called the Simplium. The architecture was designed to be as simple as possible to allow simple programs to be written so the simulation system could be tested. The framework includes a Simplium simulator and a very minimalistic assembler. The simulator simulates three phases of execution, namely fetching, decoding and the execution of the instructions. It thus takes three cycles to simulate the work done for one instruction. The Simplium further more does not support any form of interrupts or instruction parallelism. The Simplium simulator can be configured to function as a Von Neuman machine or a Harvard architecture, whereby the choice can be made to keep the code memory inside or outside the simulator process. Keeping them inside can increase performance since code fetches happen more often than data accesses.

The Simplium instruction set has 16 general purpose word length registers. Addresses and words are 32 bits integers. The instruction set was designed so that each instruction fits inside of one 32 bit word. Depending on the instruction the layout changes, relating to how much information the arguments need to contain. Each instruction of the Simplium takes three cycles to complete. In the first cycle the instruction is fetched from memory, in the second cycle the instruction is decoded and in the third cycle the instruction is executed. The Simplium does not support unsigned and non word operations to keep tooling surrounding the Simplium simple, hence the name Simplium. The Simplium has support for generating a random value and putting this value in a register.

The type 1 encoding is used for instructions that require three arguments and the type 2 encoding is used for other instructions because it allows larger arguments.

The Simplium supports the instructions described in table C.1. The description of each instruction is given in a C like pseudo language.

As can be seen the Simplium architecture does not have any special instructions for synchronization. But it supports enough instructions for shared

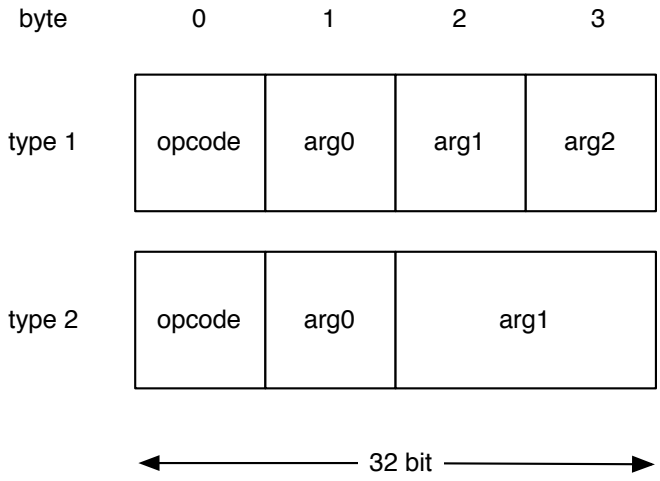


Figure C.1: **Simplium encoding**

memory synchronization algorithms like Dekkers.

Simplium assembly has a few peculiarities. Namely it requires that the user writes the memory location of the instruction before the actual instruction. This simplification made it so that the assembler did not need to have labels and data sections could be implemented with a pseudo instruction in the assembler. Included below is a simple program that reads in a value from memory, then adds this number to itself and writes the result to memory. The program will continue performing this operation forever since the last instruction is a branch to the first instruction.

```
0 DATA 5 #set data section to 5 at address 0
4 LOAD 1 0 #load address 0 into register 1
8 ADD 1 1 2 #add register 1 to register 1 and store in register 2
C SAVE 2 0 #save register 2 to address 0
10 BRANCH 4
```

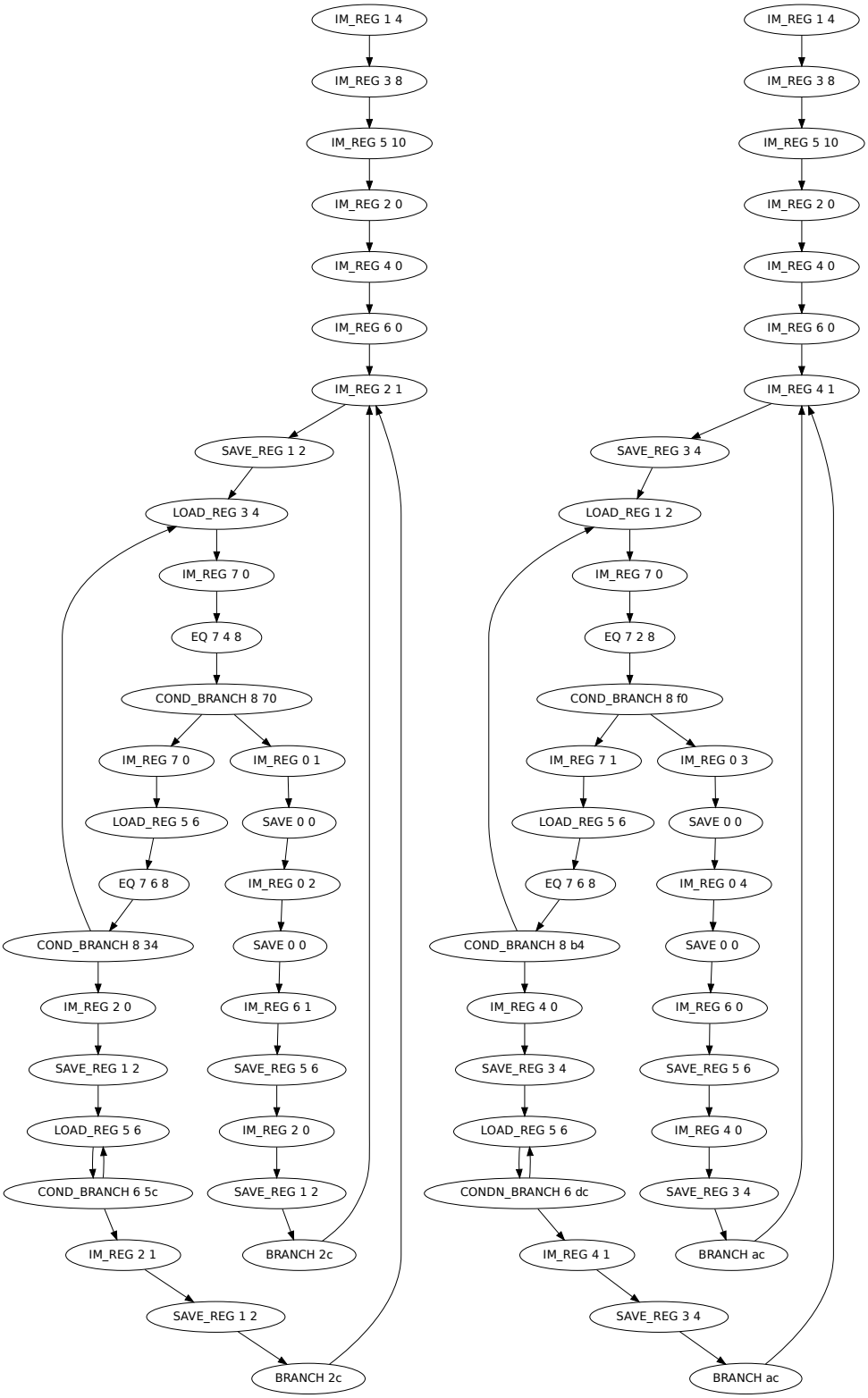
Instruction	**Arguments**	**Description**
BRANCH	address	Jump to address
COND_BRANCH	register, address	if register == 1 jump to address
CONDN_BRANCH	register, address	if register != 1 jump to address
ADD	register, register, register	reg2 = reg0 + reg1
MINU	register, register, register	reg2 = reg0 - reg1
AND	register, register, register	reg2 = reg0 AND reg1
OR	register, register, register	reg2 = reg0 OR reg1
NEG	register, register	reg2 = !reg0
LOAD	register, address	reg = *address
SAVE	register, address	*address = reg
IM_REG	register, immediate	reg = immediate
EQ	register, register, register	reg2 = reg0 == reg1
NOP		Do nothing
BRANCH_REG	register	Jump to address stored in reg0
COND_BRANCH_REG	register, register	if reg1 == 1 jump to register
DATA	value	pseudo instruction
HALT		stop simulation
LOAD_REG	register, register	reg0 = *reg1
SAVE_REG	register, register	*reg1 = reg0
RAND	register	reg0 = random();

Table C.1: Simplium instruction set overiew

Appendix D

Dekkers algorithm

Deckers algorithm is a well known synchronization algorithm. For the simulation it was needed to be translated to Simplium assembly. The control flow graph of the assembly implementation is shown below.


IM_REG 1 4
IM_REG 3 8
IM_REG 5 10
IM_REG 2 0
IM_REG 4 0
IM_REG 6 0
IM_REG 2 1
SAVE_REG 1 2
LOAD_REG 3 4
IM_REG 7 0
EQ 7 4 8
COND_BRANCH 8 70
IM_REG 7 0
IM_REG 0 1
LOAD_REG 5 6
SAVE 0 0
EQ 7 6 8
IM_REG 0 2
COND_BRANCH 8 34
SAVE 0 0
IM_REG 2 0
IM_REG 6 1
SAVE_REG 1 2
SAVE_REG 5 6
LOAD_REG 5 6
IM_REG 2 0
COND_BRANCH 6 5c
SAVE_REG 1 2
IM_REG 2 1
BRANCH 2c
SAVE_REG 1 2
BRANCH 2c
IM_REG 1 4
IM_REG 3 8
IM_REG 5 10
IM_REG 2 0
IM_REG 4 0
IM_REG 6 0
IM_REG 4 1
SAVE_REG 3 4
LOAD_REG 1 2
IM_REG 7 0
EQ 7 2 8
COND_BRANCH 8 f0
IM_REG 7 1
IM_REG 0 3
LOAD_REG 5 6
SAVE 0 0
EQ 7 6 8
IM_REG 0 4
COND_BRANCH 8 b4
SAVE 0 0
IM_REG 4 0
IM_REG 6 0
SAVE_REG 3 4
SAVE_REG 5 6
LOAD_REG 5 6
IM_REG 4 0
CONDN_BRANCH 6 dc
SAVE_REG 3 4
IM_REG 4 1
BRANCH ac
SAVE_REG 3 4
BRANCH ac

Appendix E

Observed trace graph

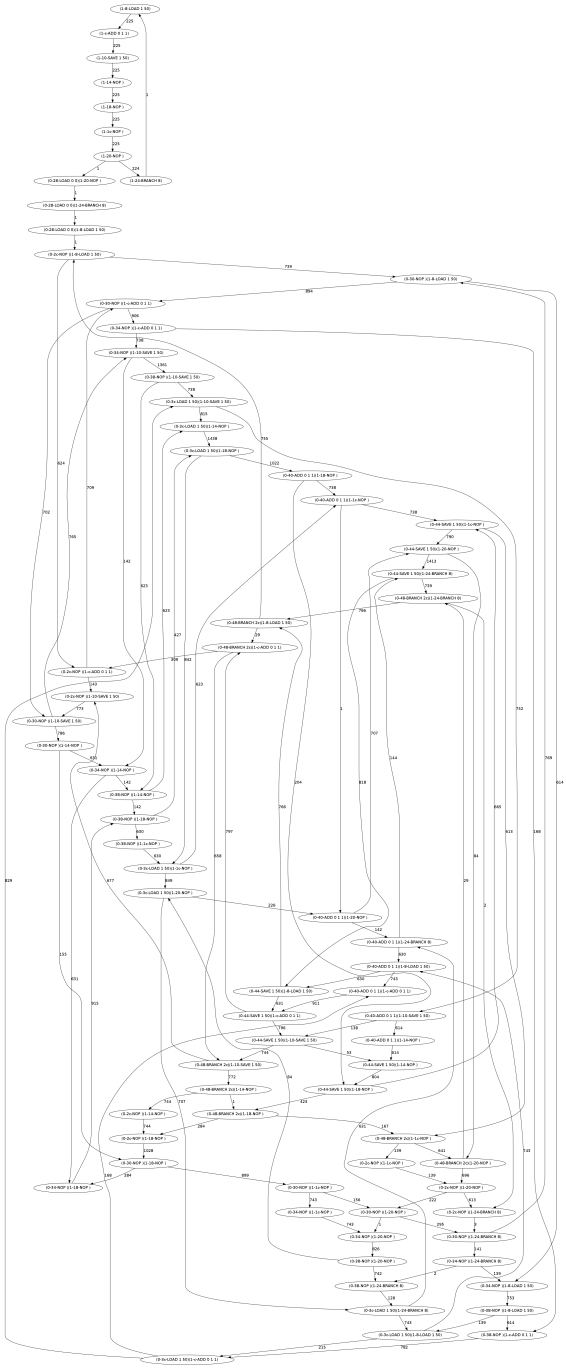


Figure E.1: **Observed trace graph Example program with relaxed synchronization**

Appendix F

Explorations of quantitative functional performance

In this appendix we explore the functioning of Handfish in a quantitative way instead of the qualitative way used previously in this report. The technique described in this appendix is the first step up to more advanced techniques that could point out to the user of the framework what pieces of the code are most sensitive to synchronization settings and it could also be used as a quality test for Handfish scheduling strategies. By comparing how similar the graph stays to lock step and taking performance into account, the developer can get an idea how effective a strategy is.

F.1 Graph similarity

To compare the functional effect of changing the number of cycles between synchronization points we need a measurement of how different the application behaves. The measurement chosen was the similarity of the edges of the observed structural trace graph. We have chosen this measure since trace graphs are efficient structures to compare different executions and by comparing the edges connecting the nodes we can easily get a grasp of differences in structure. Since a graph with similar nodes can be completely dissimilar, while a graph with similar edges is in principle similar.

We can distinguish between two uses cases for the similarity. Namely how similar are two given trace graphs as a whole and how similar is the structure of the graph. The first measure is useful to determine how much changed between two traces that where generated under similar scenario's while the later is useful in scenarios where we compare traces that come from runs with different settings. The first is useful since it focuses more on the differences in the run, by showing which paths have been taken more often. In the other case it is more interesting to look which edges exist and which do not exist in the other run. This is used

for example in determining if changing the simulation strictness results in a larger graph or not compared to a simulation run in lockstep.

For the first similarity all edges in the graph are compared in the graph, while in the second scenario duplicate edges are removed from the graph before the algorithm calculates the similarity [1].

First two memory access traces are needed to perform this comparison. These traces are first transformed to graphs in the manner described in the algorithm 1. The edges of the graph are compared to each other. This results in two numbers which represent how similar the two graphs are. The first number (S0) describes how similar the the largest graph is to the smallest graph and the second number (S1) describes how similar the smallest graph is to the largest graph. This allows us to distinguish between graphs that are subsets of each other. An example would be two graphs that have a similarity of (0.5, 1.0) which means that the large graph is proper superset of the smaller graph. The matching can be described mathematically with the following formula's.

$$S0 = \frac{|small \cap large|}{|large|}$$
$$S1 = \frac{|small \cap large|}{|small|}$$

Whereby small is the set of edges of the smallest graph and large is the set of the largest graph.

To compare two graphs from similar settings we first add all duplicate edges to graph, thus turning a ordinary trace graph in a structural trace graph. We then take the S0 as a our similarity score. This works because the graphs will roughly be the same size if they are made with similar settings so the difference of S1 is less interesting.

This technique also has a number of disadvantages. Since it has to compare all the edges between the two graphs it is computational expensive for large graphs since it requires $n*m$ comparisons to be done on a large dataset. Another disadvantage is that the similarity of the trace graphs can only be done for a Von Neummann architecture simulator since it needs to know all the code accesses to generate such a graph.

To illustrate similarity we will use the example trace graphs from chapter 4 found in figure F.1. First we classify all the edges in the two graphs in table F.1. These graphs are not necessarily realistic since a strategy can have strange effects on a piece of code.

By looking at figure F.1 we see that trace graph 2 is the largest graph with a total of 13 edges versus the 6 edges in trace graph 1.

With this information we can compute the two values that describe the similarity between the two graphs. The S0 for these two graphs is $6/13 = 0.461$ and the S1 of both these is $6/6 = 1.0$.

[1] In the analysis tools, duplicate edges are implemented as an integer property attached to the edge, thus allowing them to be discarded easily.

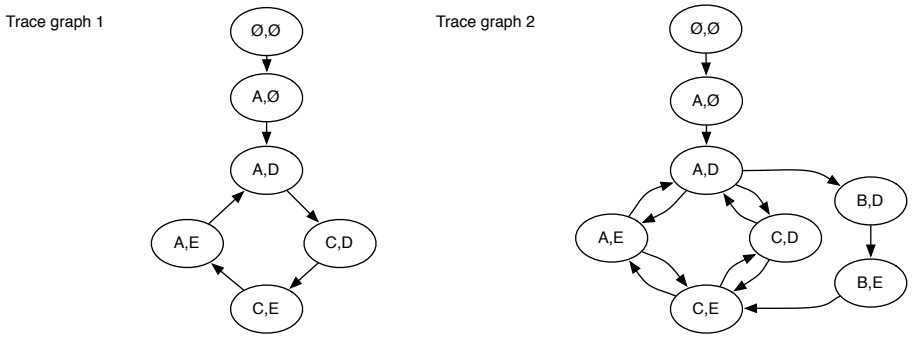


Figure F.1: **Two trace graphs**

In both	**Only in Graph 2**
$(A, D) \rightarrow (C, D)$	$(A, D) \rightarrow (B, D)$
$(C, D) \rightarrow (C, E)$	$(B, D) \rightarrow (C, E)$
$(C, E) \rightarrow (A, E)$	$(C, D) \rightarrow (A, D)$
$(A, E) \rightarrow (A, D)$	$(A, D) \rightarrow (A, E)$
(Ø,Ø) $\rightarrow$ (A,Ø)	$(A, E) \rightarrow (C, E)$
$(A, \emptyset) \rightarrow (A, D)$	$(C, E) \rightarrow (C, D)$
	$(A, D) \rightarrow (B, D)$
	$(B, E) \rightarrow (C, E)$

Table F.1: Edge classifications

F.2 What does similarity mean?

The values of the trace graph similarity go from 0 to 1. With 0 meaning that the graphs are completely dissimilar and 1 meaning that they are completely similar. While we already discussed that similarity is composed of two numbers one representing how the biggest fits in the smallest and the other number representing how the smallest graph fits in the biggest. Outside of the defined minimum and maximum how do we interpret similarity?

One important aspect to take into account is does the application generate the same output or is the output still correct between two runs. This is something similarity does not tells us but it is crucial to keep in mind when discussing the similarly of two state space graphs.

If we look at the example we see that graph 1 is a strict subset of graph 2, we see this reflected in S1 which 1.0, meaning that every edge in graph 1 is present in graph 2. But we only see a S0 of 0.4 meaning graph 2 is bigger than the other graph.

A good example would be a program that uses synchronization mechanisms like semaphores. If such an application is written correctly it will always produce the correct output but the state space graph similarity might be low. This could be interpreted as that the application handles many different kinds of

situations correctly. While having correct output and dissimilar graphs might give the author of an application confidence that it is correct it will never give 100 percent certainty since it might not reach all the nodes and edges in the maximum reachable trace graph.

There might also be applications that produce very similar trace graphs but do not generate the correct output in these cases. In such a case it might be interesting to look at the structural state space graphs to see where the problem is located.

Similarity is thus a low level means of determining how much two executions of program have in common. In some cases it can be used to debug applications but it can also be used to gain confidence that a program has been correctly written. But it can be combined with graph observation tools that try to determine if any forbidden nodes where detected.

Similarity can also be used to compare three executions and say which one one is relatively closest to the original execution since an execution with a similarity of 0.8 will resemble the original execution more than a similarity of 0.2.

In our experiments we also automatically checked Dekker's algorithm to see if faulty output was generated. This was our qualitative check that the implementation was functioning correctly which we combined with a quantitive measurement via the trace graph similarity. In terms of similarly we expect that the the similarity will drop faster with unsynchronized application than with a well synchronized application like Dekker's algorithm.

F.3 Experiments

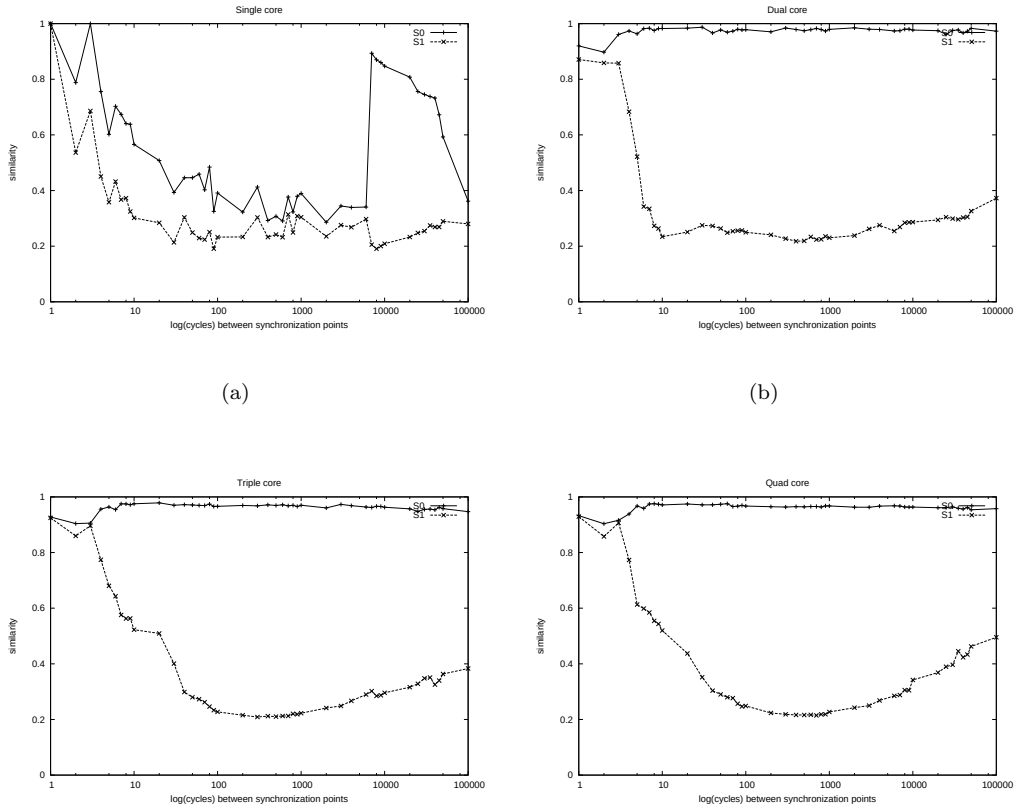
The experiment we do with trace graph similarity is looking at how the number of cores and the number of cycles between synchronization points affects the similarity.

We do this by comparing the S0 and S1 similarity between the runs at different syntonization distances with runs at lockstep.

Figure F.2 and F.2 [2] show how similar the traces are with lockstep scenario if we modify the number of cores and the synchronization distance for the example application and the implementation of Dekker's algorithm. The first thing we notice is that in the single-core scenario the graph is erratic while in the other scenario the trace graph similarity is more regular.

This erratic behavior is caused by the time sharing in a single core host. Because only one sub-simulator can make progress at a time the traces will be different from a multi-core host. Thus the trace might include more steps of a single sub-simulator before another sub-simulator makes a step. Because the

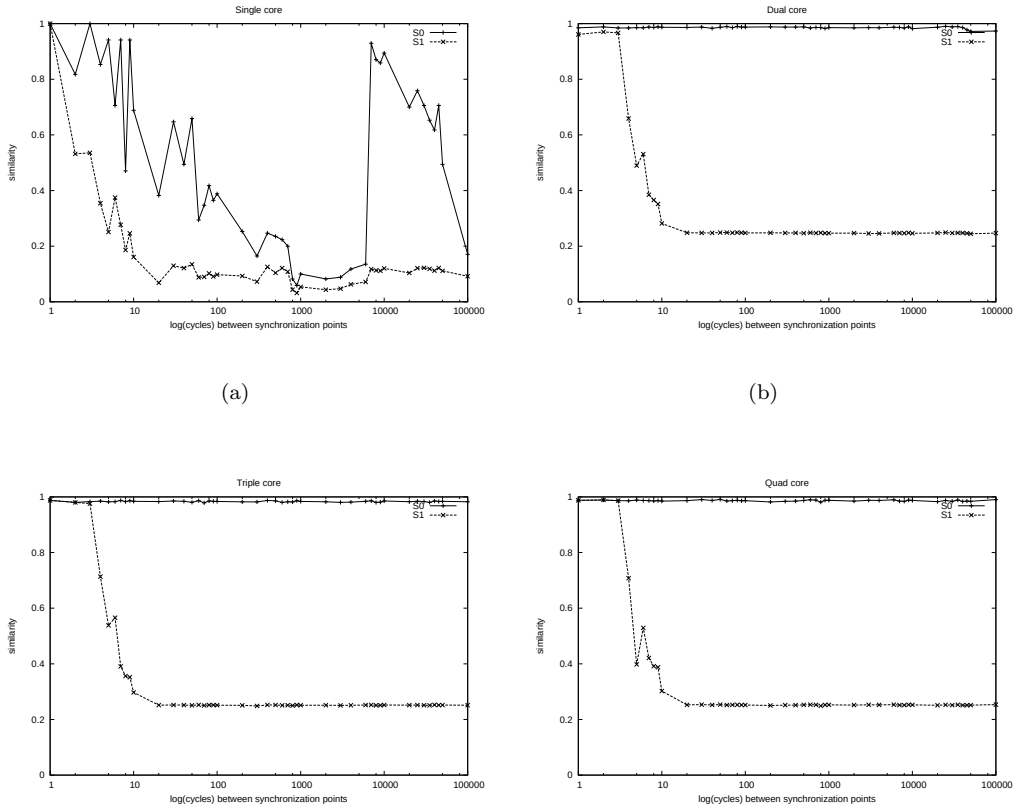
[2] In all graphs log(cycles) imply logarithmic scale, not the logarithm of the cycles.



(a) (b)

(c) (d)

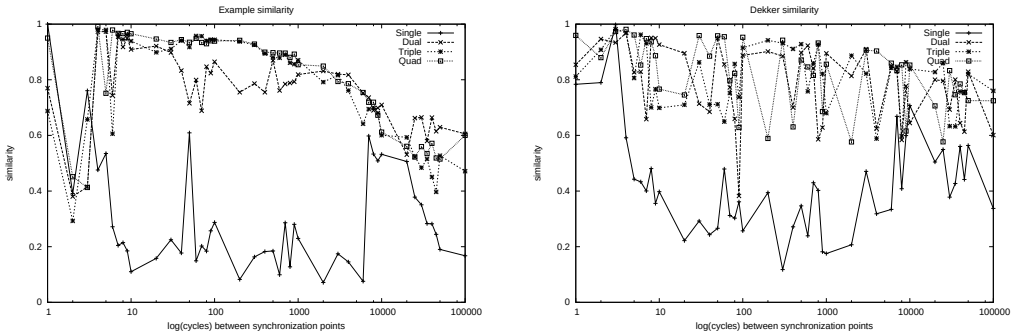
Figure F.2: Similarity Dekker's algorithm lockstep



(a) (b)

(c) (d)

Figure F.3: Similarity example algorithm lockstep



(a) (b)

Figure F.4: Similarity between runs

scheduler also changes the number of cycles between these points this results in runs that produce an erratic similarity, since the run will have less in common with each other for different settings.

When running on a multi-core host we see that the similarities are more well behaved. What we see is that in those cases that observed trace graph becomes a superset of the graph with lockstep settings. This means that more pieces of the program interact with each other than is the case with lockstep. This implies that the simulation passes through more of the reachable trace graph at a higher synchronization distance. The small upswing at the end is caused by the graph shrinking because due the distance between synchronization points increasing. This causes some nodes in the graph can not be visited anymore because they require the simulators to reach a certain point before they can be reached.

As with the lockstep functional experiments we notice that Handfish behaves more erratically on a single-core than on a multi-core. This is for the same reasons as specified above.

If we compare the similarities between the example application and Dekker's algorithm we can see that the inter run similarity drops faster in the example compared to Dekker's algorithm. This is caused by the nature of the reachable trace graph of Dekker's algorithm since it is limited by control flow, while in the example application all instruction combinations are possible. This means that the runs of Dekker's algorithm remain more similar even when we relax the temporal synchronization. Which shows that programs that are designed to be general remain well behaved with different synchronization settings while other programs will start behaving more erratically in a less strict simulation. The difference between true parallelism and time sharing can also be seen in the similarities. In the single core scenario the similarity is low because the each sub-simulator is given an amount of time to run an as such can issue multiple fetches before the other sub-simulator gets some time. Thus the trace contains

more changes of a single sub-simulator before another sub-simulator advances.

F.4 Conclusion

Functionally we can conclude the following. Running on a single core simulator results in erratic similarities and as such is not recommended. Adding more than two cores does not seem to affect the similarity when simulating a dual core simulator, which is not surprising since we are simulating a dual-core. The structure of an application also plays a big role in how the application behaves under a less strict simulator as the example application is more affected by the change in simulation strictness than Dekker's algorithm.

Trace graph similarity can be used to compare different strategies since it can be used to derive properties of the execution like "does it cause different pieces of code to interact compared to lockstep" and "how repeatable is a run with a strategy". We can use the result of the free strategy as a benchmark for other strategies since it allows us to explore fastest simulation possible with Handfish (synchronization distance length equals simulation length) and the slowest (lockstep).

Appendix G

Heterogenous simulator

On the heterogeneous system contain a Simplium and two Xentium simulators a simple application was run that combined the a control part on the Simplium with a some data processing part on the Xentium.

The Simplium assembly representing the control code is listed below. The Xentium first creates a random buffer and then signal the Xentiums that they can get the information from the buffer. The control part then waits until it the Xentium completes the computation. Then the buffers are swapped again. The Simplium code is written in assembly since the Simplium lacks a C compiler.

```
0 IM_REG 0 0          # set register 0 to 0
4 IM_REG 1 28         # set register 1 to 28
8 RAND 2              # write random value to register 2
c SAVE_REG 0 2        # write register
10 IM_REG 3 4         # set register 3 to 4
14 ADD 0 3 0          # r0 + r3 => r0
18 EQ 0 1 4           # compare register 0 and 1 write to 4
1C COND_BRANCH 4 24   # branch to 0x24 if 4 is true
20 BRANCH 8           # else goto 0x8
24 IM_REG 0 2         # set register 0 to 2
28 SAVE 0 30          # save register
2c LOAD 1 30          # load 0x030 into register 30
30 CONDN_BRANCH 1 2c  # branch to 0x2c if r1 is false
34 IM_REG 0 0         # set register 0 to 0
38 SAVE 0 30          # write register 0 to 0x30
3c BRANCH 4           # go back to 0x04
```

Listing G.2: Simplium code heterogenous

The Xentium data processing part is written in C and is listed below. The Xentium waits for the Simplium to write a value to the status variable. Then computes the sum and the average and tells the Simplium that it is done.

```
#define MEMORY  4 * 1024 * 1024

int main()
{
    // configure write pointer
    volatile int* writer = (int*) (MEMORY + 0x34);

    // loop forever
    while(1)
    {
        // configure status wait pointer
        volatile int* wait = (int*) (MEMORY+ 0x30);

        // await on status pointer
        while(*wait != 2) {}

        // configure memory pointer
        volatile int* read = (int*)(MEMORY);

        // actual sum
        int i = 0;
        int sum = 0 ;
        while (i < 10)
        {
            read++;
            sum+=*read;
            i++;
        }

        // write out average
        int avg = sum / 10;
        *writer = avg;
        writer++;

        // signal that we are done
        int val = 1;
        *wait = val;
    }
}
```

Listing G.3: Xentium code in C, heterogeneous

Bibliography

[1] Pin - a dynamic binary instrumentation tool. `http://www.pintool.org/`.

[2] Qemu open source emulator and virtualizer. `http://wiki.qemu.org/Main_Page`.

[3] a fast mutual exclusion algorithm. *ACM Transactions on Computer Systems 5*, 1 (February 1987), 1–11.

[4] *OMAP 2 Architecture: OMAP2420 Processor product bulletin.* Texas Instruments, 2005.

[5] *GRSIM User's Manual.* Aeroflex Gaisler AB, 2011.

[6] Advanced Micro Devices, Inc. Amd-vtm nested paging. Tech. rep., http://developer.amd.com/assets/NPT-WP-1

[7] ANSI X3. *ISO/IEC 14882 Programming Languages C++.* ISO/IEC.

[8] Argollo, E., Falcón, A., Faraboschi, P., Monchiero, M., and Ortega, D. Cotson: infrastructure for full system simulation. *SIGOPS Oper. Syst. Rev. 43* (January 2009), 52–61.

[9] Boehm, H.-J., Demers, A. J., and Uhler, C. Implementing multiple locks using lamport's mutual exclusion algorithm. *ACM Lett. Program. Lang. Syst. 2*, 1-4 (Mar. 1993), 46–58.

[10] Burkey, R. Virtual agc home page `http://www.ibiblio.org/apollo/`.

[11] Chen, J., Kumar Dabbiru, L., Wong, D., Annavaram, M., and Dubois, M. Adaptive and speculative slack simulations of cmps on cmps. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on* (2010), pp. 523 –534.

[12] CRISP Consortium. Crisp project `http://www.crisp-project.eu/`.

[13] Dales, M. Software arm `http://www.cl.cam.ac.uk/~mwd24/phd/swarm.html`.

[14] Dijkstra, E. Ewd123- cooperating sequential processes. `http://www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD123.html`.

[15] Feyman, R. *Feynman's lectures on physics.* Addison-Wesley, 1964.

[16] Fossati, L. *LEON2/3 SystemC Simulator: User Manual*. ESA & Politecnico di Milano.

[17] G., S. V. K. Shared memory consistency models: A tutorial. Tech. rep., WRL Research Report 95/7, 95.

[18] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[19] Geoffray, N., Thomas, G., J.Lawall, Muller, G., and Folliot, B. VMKit: a Substrate for Managed Runtime Environments. In *Virtual Execution Environment Conference (VEE 2010)* (Pittsburgh, USA, March 2010), ACM Press.

[20] Ghenassia, F. *Transaction-level modeling with SystemC*. Springer, 2005.

[21] Google Inc. Google protocolbuffers `https://developers.google.com/protocol-buffers/docs/overview`.

[22] Goossens, K., Dielissen, J., and Radulescu, A. Aethereal network on chip:concepts, architectures, and implementations. *IEEE Design and Test of Computers 22* (2005), 414–421.

[23] Gosling, J., Joy, B., Steele, G., and Bracha, G. *The Java Language Specification third edition*. ADDISON-WESLEY, 2005.

[24] Hintjens, P. Ømq the guide.

[25] Hoare, C. A. R. Communicating sequential processes. *Commun. ACM 21* (August 1978), 666–677.

[26] Holzman, G. J. The model checker spin. *IEEE Transactions of software engineering 20*, 5 (May 1997).

[27] IEEE Computer Society. Ieee standard systemc® language reference manual. electronic, March 2005.

[28] Intel Corporation. Intel core i5-2300 `http://ark.intel.com/products/52206/Intel-Core-i5-2300-Processor-(6M-Cache-up-to-3_10-GHz))`.

[29] Intel Corporation. Intel pentium d processor 805 `http://ark.intel.com/products/27511/Intel-Pentium-D-Processor-805-(2M-Cache-2_66-GHz-533-MHz-FSB)`.

[30] Intel Research. Intel's teraflops research chip. `http://download.intel.com/pressroom/kits/Teraflops/Teraflops_Research_Chip_Overview.pdf`.

[31] Javapathfinder. Javapathfinder `http://babelfish.arc.nasa.gov/trac/jpf/`.

[32] Kungas, P. Petri net reachability checking is polynomial with optimal abstraction hierarchies. In *Proceedings of the 6th International Symposium on Abstraction, Reformulation and Approximation,* (2005).

[33] Lamport, L. A new solution of dijkstra's concurrent programming problem. *Communications of the ACM 17*, 8 (1974).

[34] LLVM developer. Llvm 2.9 release notes `http://llvm.org/releases/2.9/docs/ReleaseNotes.html`.

[35] Martin, M. M. K., Sorin, D. J., Beckmann, B. M., Marty, M. R., Xu, M., Alameldeen, A. R., Moore, K. E., Hill, M. D., and Wood, D. A. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News 33* (2005), 2005.

[36] Mello, A., Maia, I., Greiner, A., and Pecheux, F. Parallel simulation of systemc tlm 2.0 compliant mpsoc on smp workstations. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010* (2010), pp. 606 –609.

[37] Mentor Graphics. Modelsim asic and fpga design `http://www.mentor.com/products/fv/modelsim/`.

[38] Miller, J., Kasture, H., Kurian, G., III, C. G., Beckmann, N., Celio, C., Eastep, J., and Agarwal, A. Graphite: A distributed parallel simulator for multicores. cited By (since 1996) 0; Conference of 16th International Symposium on High-Performance Computer Architecture, HPCA-16 2010; Conference Date: 9 January 2010 through 14 January 2010; Conference Code: 80323.

[39] MIPSIM. Mipsim - mips assembly language simulator `http://www.mipsim.com/mipsim/`.

[40] Molnar, I., and Andrews, J. Linux: The completly fair scheduler `http://kerneltrap.org/node/8059`.

[41] Monchiero, M., Ahn, J. H., Falcón, A., Ortega, D., and Faraboschi, P. How to simulate 1000 cores. *SIGARCH Comput. Archit. News 37* (July 2009), 10–19.

[42] Mühlenfeld, A., and Wotawa, F. Fault detection in multi-threaded c++ server applications. *Electron. Notes Theor. Comput. Sci. 174* (June 2007), 5–22.

[43] Murata, T. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE 77*, 4 (apr 1989), 541 –580.

[44] Neiger, G., Santoni, A., Leung, F., Rodgers, D., and Uhlig, R. Intel® virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology Journal 10*, 3 (2006).

[45] Nethercote, N., and Seward, J. Valgrind: a framework for heavy-weight dynamic binary instrumentation. *SIGPLAN Not. 42* (June 2007), 89–100.

[46] Oracle Inc. Java hotspot vm `http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html`.

[47] Popek, G. J., and Goldberg, R. P. Formal requirements for virtualizable third generation architectures. *Commun. ACM 17* (July 1974), 412–421.

[48] Popovici, K; Ahmed J.A. Ahmed, J. Virtual platforms in system-on-chip design. In *Design automation conference conference 47* (2010).

[49] Recore Systems BV. Recore systems `www.recoresystems.com`.

[50] Recore Systems BV. Xentium tools.

[51] SimpleScalar. Simplescalar llc `http://www.simplescalar.com/`.

[52] Smit, G. J., Rosien, M. A., Guo, Y., and Heysters, P. M. Overview of the tool-flow for the montium processor tile. In *International Conference on Engineering of Reconfigurable Systems and Algorithms, ERSA 2004* (2004), CSREA Press, pp. 45–51.

[53] The Open SystemC Initiative. Osci tlm-2.0 language reference manual. electronic, July 2009.

[54] Virtutech. Simics `http://www.virtutech.com/`.

[55] Wieferink, A., Doerper, M., Kogel, T., Leupers, R., Ascheid, G., and Meyr, H. Early iss integration into network-on-chip designs. In *In In Proceedings of International Workshop on Systems, Architectures, Modeling and Simulation (SAMOS), Samos* (2004).

[56] Wilson, M. Hypervisor and virtualization presentation.

[57] Wolkotte, P. T. *Exploration within the Network-on-Chip Paradigm.* PhD thesis, University of Twente, Enschede, January 2009.

[58] Wu, M.-H., Fu, C.-Y., Wang, P.-C., and Tsay, R.-S. An effective synchronization approach for fast and accurate multi-core instruction-set simulation. In *Proceedings of the seventh ACM international conference on Embedded software* (New York, NY, USA, 2009), EMSOFT '09, ACM, pp. 197–204.

[59] Wu, M.-H., Wang, P.-C., Fu, C.-Y., and Tsay, R.-S. A high-parallelism distributed scheduling mechanism for multi-core instruction-set simulation. In *Proceedings of the 48th Design Automation Conference* (New York, NY, USA, 2011), DAC '11, ACM, pp. 339–344.