NTNU – Trondheim
Norwegian University of
Science and Technology

SHMACsim

A Cycle-accurate Simulation Infrastructure
for the Heterogeneous SHMAC Multi-Core
Prototype

Yaman Umuroglu

Embedded Computing Systems
Submission date: July 2013
Supervisor: Magnus Jahre, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

A Cycle-accurate Simulation Infrastructure for the Heterogeneous SHMAC Multi-Core Prototype

Current multi-core processors are constrained by energy. Consequently, it is not possible to improve performance further without increasing energy efficiency. A promising option for making increasingly energy efficient CMPs is to include processors with different capabilities. This improvement in energy efficiency can then be used to increase performance or lower energy consumption.

Currently, it is unclear how system software should be developed for heterogeneous multi-core processors. A main challenge is that little heterogeneous hardware exists. It is possible to use simulators, but their performance overhead is a significant limitation. An alternative strategy that offers to achieve the best of both worlds is to leverage reconfigurable logic to instantiate various heterogeneous computer architectures. These architectures are fast enough to be useful for investigating systems software implementation strategies. At the same time, the reconfigurable logic offers the flexibility to explore a large part of the heterogeneous processor design space.

The Single-ISA Heterogeneous MAny-core Computer (SHMAC) project aims to develop an infrastructure for instantiating diverse heterogeneous architectures on FPGAs. A prototype has already been developed, but an important remaining challenge is to design a flexible memory system while retaining high performance. This and other architectural trade-offs are best evaluated in a simulator. With simulation, it is possible to evaluate a large number of policies without an excessive implementation effort.

The task in this assignment is to develop a simulator for the SHMAC prototype. In addition, the student should implement a set of micro-benchmarks that stress important aspects of the architecture. If time permits, the student should propose improvements and evaluate the performance impact of these improvements.

Supervisor: Assoc. Prof. Magnus Jahre, IDI

Abstract

The fast-paced development trend in microprocessor performance characterized by Moore's Law can no longer continue unperturbed. Shrinking semiconductor node size still translates into increasing transistor count but not directly into performance, since thermal and power constraints are limiting the amount of transistors that can be used simultaneously. One way of exploiting this "dark silicon" is building heterogeneous systems containing specialized accelerators and cores. The SHMAC project aims to provide a research platform for heterogeneous systems research. An FPGA prototype has been constructed for the SHMAC, but to have a rapid implement-evaluate cycle for system policies, software simulation is needed.

This thesis covers the design and implementation of a cycle-accurate simulation infrastructure for the SHMAC. Additionally, the current state of the architecture is evaluated with a set of micro-benchmarks and several improvements are proposed. The constructed infrastructure offers a highly configurable, cycle-accurate simulation of the SHMAC FPGA prototype. A micro-benchmark-based analysis of the current state of the architecture exposes the router hop latency and throughput as the greatest bottlenecks. To address this a dual-port RAM slave with router bypass is implemented, resulting in 3.5× instruction fetch speedup and contributing to overall system performance. Improvements contributing traffic independent clock counting and bootstrapping functionality, and a network packet lifetime instrumentation method are also described.

Preface

This thesis is submitted to the Norwegian University of Science and Technology in partial fulfilment of the requirements for the European Master in Embedded Computer Systems (EMECS) degree.

This work has been performed at the Department of Computer and Information Science, NTNU, Trondheim, with Assoc. Prof. Magnus Jahre as the supervisor.

Acknowledgements

I would like to extend my gratitude to my supervisor Magnus Jahre for his extensive support throughout the entire process and for enabling me to start a research career on heterogeneous multi-core platforms, to Håkon Marthinsen for his extensive moral support as well as invaluable help on LaTeX and Inkscape, and to all my instructors and friends at the Erasmus Mundus in Embedded Computing Systems programme for a fantastic two years.

The "Virtual Heterogeneous Multicorn" logo is built upon the work of Dee Dreslough with post-processing by Karl Johan Heimark and myself.

Contents

List of Figures

List of Tables

List of Abbreviations

AMP Asymmetric Multi-Core Processor

AT Approximately Timed

BFM Bus Functional Model

CA Cycle Accurate

CPI Cycles Per Instruction

CPU Central Processing Unit

DSP digital signal processor

EECS Energy Efficient Computing Systems

ELF Executable and Linkable Format

FPGA Field-Programmable Gate Array

FPU Floating Point Unit

GCC GNU Compiler Collection

GDB GNU Debugger

GPGPU General-Purpose Graphics Processing Unit

HDL Hardware Description Language

HMP Heterogeneous Multi-Core Processor

ILP instruction-level parallelism

ISA Instruction Set Architecture

LL Load Linked

LL/SC Load Linked/Store Conditional

LT Loosely Timed

NUMA Non-Uniform Memory Access

OS Operating System

RILF Remote Impact on Local Fetch

RTL Register Transfer Level

SAM System Architectural Model

SC Store Conditional

SHMAC Single-ISA Heterogeneous Many-core Computer

SHMACsim SHMAC Simulator

SIMD Single Instruction Multiple Data

SMP Symmetric Multi-Core Processor

SoC System on a Chip

TLM Transaction-Level Modelling

TLP task-level parallelism

TSV Through-Silicon Via

U-core unconventional core

UT Untimed

1. Introduction

Since the birth of computers, the electronics and computer industries have strived forward to construct devices with more computing power. The capability to integrate an ever increasing number of transistors onto the same area, commonly referred to as "Moore's Law" [56], has led to dramatic improvements in terms of computing power. However, taking advantage of this increase in transistor count to deliver more performance is becoming increasingly difficult. In order to understand the need for heterogeneous computing, it is necessary to understand the historical trend in the development of computing power, and why the same trend is unlikely to continue.

1.1. Historical Trends in Computing Power

Central Processing Units (CPUs) have come a long way since their invention. As can be observed in Figure 1.1, the period 1995–2004 saw a doubling of single-threaded integer operation performance every two years. The increase in the number of transistors was utilized by processor designers to allow faster execution in a single processor core. Transistors that could run faster and deep pipelines with many stages, which could run at higher clock frequencies, were the driving forces behind this trend [13]. More transistors in the core made it possible to construct complicated hardware such as instruction reordering engines to exploit instruction-level parallelism, and branch predictors to avoid pipeline flushing penalties.

However, also visible in Figure 1.1 is a drop in the increase of performance after 2004. Sutter [59] attributes this to problems arising from the physical nature of shrinking transistor technology, namely power consumption and high heat production. Indeed, the increase in transistor density (as predicted by Moore's Law) does not translate to performance gains if the transistors cannot be powered. The lower power requirements of downscaled transistors (also known as "Dennard scaling") described by Dennard et al. [20] has made it possible to power the increasing number of transistors while keeping the energy and cooling budgets fixed. Unfortunately, the sub-130 nm transistor

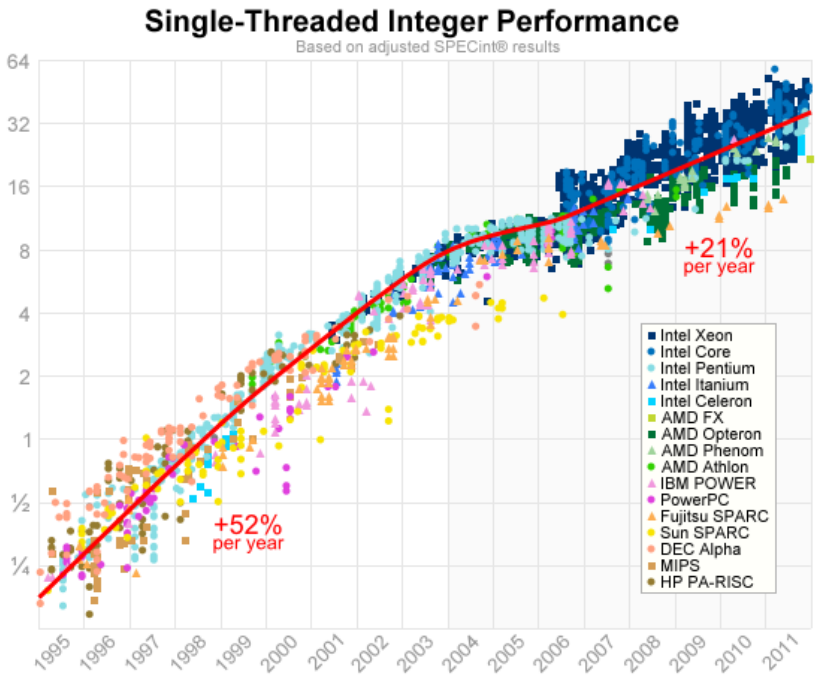


Figure 1.1.: Historical trend of single-threaded integer operation performance. Reproduced from [49]

dimensions have brought an end to Dennard scaling [35]. Further scaling down of the transistor dimensions and threshold voltage causes increased leak current, whose impact on power becomes even larger due to exponentially increasing transistor count. This means it is no longer possible to power more transistors without either increasing the power budget or leaving parts of the chip unpowered (called the "dark silicon" [22] effect).

It is undesirable to increase the power budget seeing that data centers accounted for 1.1–1.5 % of global power usage in 2010 [34] and the impact of ICT on the environment is already significant [47]. Another reason would be the energy sensitivity of the mobile computing devices such as smartphones and tablets, which are becoming more popular [24] – a mobile device consuming several watts of power would suffer from overheating and have a rather short battery life. Borkar and Chien [13] predict that energy will be the key limiter of performance for the next 20 years, and point to large-scale parallelism and heterogeneous computing for the solution.

1.2. Era of Multi-Cores: Symmetric, Asymmetric and Heterogeneous

The limitations brought by power and heat ushered the trend of multi-core processors. Intel Corporation [62] name heat production, power, memory latency and RC delay to explain their move to multi-core architectures. Another motivation for the move to multi-cores is the limited amount of instruction-level parallelism (ILP) available in typical applications. Wall [65] reports the median parallelism for a mix of real-life programs and benchmarks that can be exploited with practical hardware as 5; investing further resources into single-core CPUs capable of more ILP would have limited performance benefits. Instead, integrating N processor cores onto the same die can offer a theoretical N times increase in computing power, provided that the applications are able to exploit task-level parallelism (TLP) by the way of either using multiple threads, or running multiple applications simultaneously. Replicating an existing core multiple times to form a more powerful processing unit is also appealing in terms of reusing engineering resources. The name *Symmetric Multi-Core Processor (SMP)* [23] reflects the identical nature of each processing core on this type of processor.

The energy-sensitive nature of mobile computing devices such as smartphones and tablets has given rise to another type of multi-core processor, the

Asymmetric Multi-Core Processor (AMP) [23]. These processors typically include some simple cores for performing computationally non-intensive tasks and some powerful cores for tasks with high performance requirements. This mapping of tasks to cores has the potential to increase energy efficiency without sacrificing performance.

However, while the move to symmetric multi-cores has helped to uphold the performance growth demands and asymmetric multi-cores promise more energy efficiency, the problem of dark silicon still lurks on the horizon. Esmaeilzadeh et al. [22] argue that the failure of Dennard scaling will continue to plague multi-core designs, forcing 21 % of the transistors in a fixed-size 22 nm chip to be powered off, regardless of core asymmetry and topology. Therefore, in order to keep utilizing the increase in transistor count for higher computational power, it is of utmost importance to achieve better energy efficiency.

This is where *Heterogeneous Multi-Core Processors (HMPs)* have the potential to carry the flag further. Similar to an asymmetric multi-core, a heterogeneous multi-core processor can contain regular cores of different sizes, but it can also so-called *unconventional cores (U-cores)* [16]. These U-cores can range from vector processors to application-specific accelerators, General-Purpose Graphics Processing Units (GPGPUs) or programmable logic such as Field-Programmable Gate Arrays (FPGAs). The primary advantages of such a mix of hardware in the face of dark silicon challenges are twofold:

- Different cores and accelerators can be powered on or off according to the processing requirements of the current task. This selective powering can ameliorate the dark silicon limitations.

- Customized U-cores can perform specific types of computation with higher energy efficiency than pure software solutions running on general purpose cores [64] which can contribute significantly to the overall energy efficiency of the system.

Chung et al. [16] conclude that U-cores are very likely to find their way into future processors. Indeed, heterogeneous system-on-a-chip solutions in the form of mobile application processors and microcontrollers are already quite widespread in the embedded systems market and demonstrate the energy efficiency potential [30, 37, 50].

1.3. EECS and the SHMAC project

Established with the goal of improving energy efficiency in computing at different levels of abstraction, the Energy Efficient Computing Systems (EECS) strategic research area at the Norwegian University of Science and Technology has heterogeneous computer architectures as one of its primary research foci. Despite the rising popularity of the topic in the computer architecture community, two primary research questions still remain unanswered:

> **Heterogeneous Hardware:** How should a heterogeneous processor architecture be designed in terms of core composition, accelerators, interconnect and memory system?
>
> **Heterogeneous Software:** Given a heterogeneous processor architecture, how should application and system software be designed to obtain maximum energy efficiency?

The *SHMAC* project was initiated by the EECS to help discover the answers to these questions. The goal of the project is to develop a heterogeneous architecture which in turn will be utilized for heterogeneous software research. Important aspects of the architecture are covered in Section 2.3. A prototype for the SHMAC using reconfigurable logic has already been developed by Rusten and Sortland [54]. However, while the FPGA prototype remains a valuable tool for evaluation, the heterogeneous design space is too large to be quickly explorable by such an approach, since it is necessary to model each desired component in an Hardware Description Language (HDL). This is why SHMAC Simulator (SHMACsim) is also necessary for the SHMAC project; simulators are much more suitable for evaluating a large number of coarse-grained policies, which then can be evaluated in detail by the FPGA implementation.

1.4. Assignment Interpretation

Based on the assignment description text, the following main tasks were identified:

Task 1: ***(mandatory)*** Create a cycle-accurate simulation infrastructure that reflects the SHMAC architecture

Task 2: ***(mandatory)*** Implement a set of micro-benchmarks that stress the important aspects of the architecture, and interpret the results

Task 3: ***(optional)*** Suggest and evaluate improvements to the SHMAC architecture

Fulfilling these tasks will in turn help answer the following research questions:

RQ1 How should the SHMAC architecture be modelled in a software simulator?

RQ2 What sort of benchmark programs can be utilized to identify major bottlenecks in the SHMAC architecture?

RQ3 How can the shortcomings of the SHMAC architecture and the current implementation be remedied?

As the infrastructure to be provided as part of Task 1 is intended to serve as a research tool for continued investigations of the EECS into heterogeneous multi-core architectures, requirements and design guidelines have been further elaborated via discussions with the research group. These are summarized below.

- **Cycle Accuracy:** SHMACsim must be able to accurately reflect the complex interactions between multiple cores. The difference in cycle-based performance statistics should be less than 10 % between the simulator and the FPGA prototype. High simulation performance is not a primary goal at this point and sacrificing simulated performance for higher accuracy is deemed acceptable.
- **Configurability:** SHMACsim should be structured in a way that allows easy creation and evaluation of desired heterogeneous models via a flexible configuration system.
- **Modelling capabilities on multiple abstraction levels:** EECS aims to improve the energy efficiency of computing systems at all abstraction layers of a computing system. Therefore, flexible tools that are not bound to a particular layer of abstraction are desirable.
- **Maintainability and extensibility:** As the beginnings of a research tool, it is important that SHMACsim is easy to maintain and extend for future research. Clean interfaces, modular structure and well-commented code are important towards this.

It is worth mentioning here that the SHMAC is a dynamic research project, and development continued in parallel with SHMACsim. *SHMACsim as presented in this report is not intended to and does not reflect the state-of-the-art SHMAC architecture, but rather the FPGA prototype.* For instance,

the cores in SHMAC have been changed from MIPS to ARM Instruction Set Architecture (ISA) while SHMACsim was still in development, which could not be reflected in the simulator due to time constraints.

1.5. Contributions

This work makes contributions to the SHMAC project in three main areas, in connection with the three tasks defined in the assignment text. A complete list of contributions is provided in Table 1.1.

1.6. Report Organization

The organization of the report into individual chapters is briefly described for the reader's convenience:

- **Chapter 1: Introduction** gives a brief introduction to the trends in processor design and describes the motivations behind the SHMAC and SHMACsim.
- **Chapter 2: Background** describes the main concepts in heterogeneous processor and simulator design, as well as related work on these subjects by other authors.
- **Chapter 3: The SHMAC Simulator Infrastructure** goes into the details of SHMACsim's design and the influencing factors on design decisions, and how it was tested and verified against the FPGA prototype.
- **Chapter 4: Evaluating the SHMAC: Micro-benchmarks** introduces a description of implemented benchmarks and the results obtained from their execution.
- **Chapter 5: Improving the SHMAC** contains a set of proposed improvements based on this discussion, and their performance impact.
- **Chapter 6: Conclusion and Future Work** provides the concluding remarks and propositions for future work on carrying SHMACsim further.

Task	Section(s)	Description
T1	3.2	A SystemC/C++ model of the SHMAC, cycle accurate to within 5% of the FPGA prototype
T1	3.2.1	An ArchC-generated MIPS core model extended with LL/SC instructions
T1	3.2.1.1	A TLM adapter for ArchC core integration into SHMAC tiles
T1	3.4	A set of utilities to compile C applications for a given tile coordinate for ArchC MIPS cores
T1	3.3.1	A centralized system for defining and constructing tile types
T1	3.3	A configuration system for tile layout and runtime parameters
T2	4.3	A micro-benchmark to measure start-up delays due to the jump tile
T2	4.2	A micro-benchmark to measure effects of traffic on clock tile-based timing
T2	4.4.2	A micro-benchmark to measure the effects of sharing a RAM tile between multiple masters
T2	4.5	A micro-benchmark to measure performance impact of remote memory accesses on local cores
T2	4.6	A micro-benchmark to test synchronization of large grids with the LL/SC tile
T3	5.1	A dual-port RAM implementation in software to increase local memory performance
T3	5.2	A system register file implementation to provide per-tile bootstrapping and clock counting
T3	5.3	A packet tracker implementation to instrument memory packet route information in time

Table 1.1.: A list of contributions provided by this work, in relation to the corresponding tasks from 1.4.

2. Background

The design of HMPs is a broad field encompassing a number of different sub-fields including the design of processing cores, interconnect and the memory system. In order to have a better understanding of the heterogeneous nature of the SHMAC, it is beneficial to examine these sub-fields to some extent.

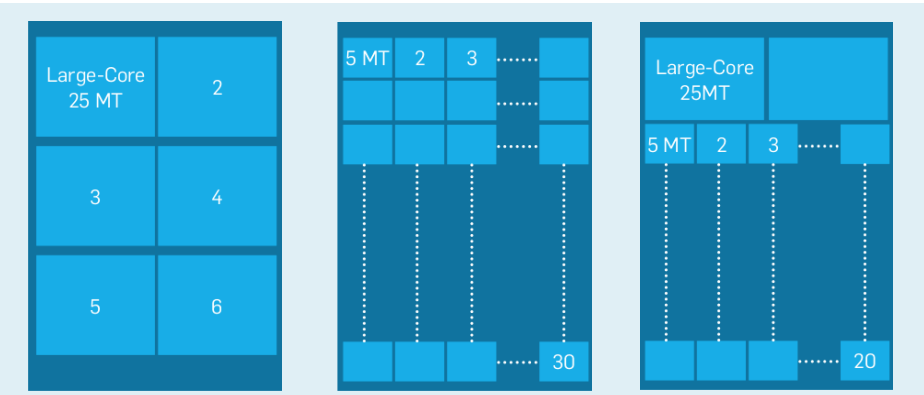
In this chapter, the reader is provided with an overview of the background concepts and related work surrounding HMP design. A summary of the SHMAC architecture is also presented. This is followed by a section on computer architecture simulators, introducing the simulation technology background SHMACsim draws on.

2.1. The Motivation for Heterogeneous Multi-Cores

As explained in Chapter 1, the primary motivation for the move to HMPs is their potential for higher energy efficiency in the face of dark silicon challenges, while still offering improved performance. This section aims to provide some insight into the analytical models and empirical examples justifying these claims regarding heterogeneous multi-cores.

As HMPs can be thought of as asymmetric multi-cores with U-cores, one can start by looking at the analytical foundations for AMPs. These can be based on three so-called "laws":

- *Amdahl's Law* describes the maximum theoretical possible speed-up limit obtainable for an application via parallel execution given its parallelizable portion [3]. It draws the primary conclusion that massively parallel execution of applications is of limited benefit, since inherent sequential portions in the application limit the performance improvement.
- *Gustafson's Law* is essentially a reinterpretation of the same relationship Amdahl's Law describes. However, Gustafson [27] pointed out that despite the limitations imposed by serial portions, the speed-ups



Large-Core Homogeneous	
Large-Core Throughput	1
Small-Core Throughput	
Total Throughput	6

Small-Core Homogeneous	
Large-Core Throughput	
Small-Core Throughput	Pollack's Rule $(5/25)^{0.5} = 0.45$
Total Throughput	13

Asymmetric Multi-Core	
Large-Core Throughput	1
Small-Core Throughput	Pollack's Rule $(5/25)^{0.5} = 0.45$
Total Throughput	11

Figure 2.1.: Comparison of 150 million transistors integrated into three different multi-core layouts. Adapted from [13]

brought by massively parallel execution can also be utilized to attack larger problem sizes, which otherwise would be impractical due to long execution times.

- *Pollack's Rule* theorizes that there is a direct proportion between performance improvements from a microprocessor and the square root of the amount of transistors used for improving that processor's micro-architecture [48]. Thus, one can make a rough estimate the total available computing power given a number of transistors and how they are to be allocated into different cores.

Figure 2.1 presents an comparison of three possible multi-core configurations given 150 million transistors, with the throughput of each core estimated using Pollack's Rule. Although the small-core homogeneous layout has greater total throughput than the others, the limited degree of parallelism available in most applications (as suggested by Amdahl) will limit the possible speedup. The AMP is able to offer a better compromise thanks to its mix of core types. Fedorova et al. [23] suggest that an AMP system can offer better performance and minimize power usage by executing the sequential and parallel phases of the application on large and small cores, respectively.

In addition to the benefits of AMPs, HMPs have the advantage of containing specialized U-cores. These specialized cores and accelerators can have up to two orders of magnitude higher energy efficiency [16, 57] and can be selectively powered on or off depending on the task at hand. This selective powering allows an HMP to trade dark silicon area for increased performance and energy efficiency. Borkar and Chien [13] refer to this as the "Rise of 10x10 Optimization": operating the chip with 90 % of the transistors inactive, and a different 10 % active at each point in time.

2.2. Hardware for Heterogeneous Multi-core Processors

2.2.1. Core Types and Accelerators

The types and features of processor cores and accelerators to be included on an HMP is a major determinant in the system's final capabilities. Many different processor cores and accelerators have been designed over the history of computing, and an HMP's ability to combine several of them into a single system increases the size of the design space to vast proportions. The question of which cores and accelerators to use is to some extent answerable within a given application area, but is much tougher for general purpose HMPs.

In terms of asymmetric general-purpose core mixes, both Kumar et al. [36] and Van Craeynest & Eeckhout [63] argue that two core sizes provide most benefits from heterogeneity. This mix of cores (usually referring to core types as "big/powerful" and "small/simple") can also be found in the industry; both ARM's big.LITTLE [26] (depicted in 2.3) and NVIDIA's Kal-El [45] are such examples.

Besides general-purpose cores, two rising trends in HMP design in the industry are the inclusion of media processing elements and GPGPUs for parallel processing. The Texas Instruments OMAP4470 System on a Chip (SoC) whose the range of computing elements is displayed in Table 2.1 is such an example. This range of core types can offer the benefits of asymmetric multi-cores in terms of general purpose computing, as well as energy efficient multimedia processing with the DSP and accelerator hardware. Similar HMPs are widely used in power sensitive computing devices such as smartphones and tablets [50, 55].

Name	Core Type	Core Count
ARM Cortex A9	general purpose (high performance)	2
ARM Cortex M3	general purpose (energy efficient)	2
PowerVR SGX544	graphics processing unit	4
IVA-HD	still image and video accelerator	1
mini-C64x+	digital signal processor	1

Table 2.1.: List of cores in a Texas Instruments OMAP4470 SoC [30]

Beyond these core types, there has also been research on including less conventional cores and hardware accelerators onto HMP dies. These include energy-efficient *conservation cores* [64] auto-synthesized from application source code, database indexing accelerators [32] and hardware web browsing [7], among others. Last but not least, there is substantial work on *reconfigurable computing* where FPGAs or similar reconfigurable hardware is utilized to dynamically create datapaths suited for the task at hand [28]. Chung et al. [16] suggest that such hardware is broadly useful if energy efficiency is a primary goal and is likely to be part of future HMP designs.

2.2.2. Interconnect

Interconnect has a critical role in HMP system design. For instance, many modern personal computers also contain heterogeneous processing elements such as GPUs, but the external nature of the GPU and the bus that connects it to the rest of the system gives rise to large delays while copying data from/to these units, creating bottlenecks. In contrast, an HMP contains the computing units and the interconnect on the same chip, allowing higher-speed communication between them.

The traditional bus is based around the idea of connecting multiple devices to a global shared medium. However, semiconductor technology scaling has brought severe limitations to this approach: global synchronization problems, deep submicron effects and power/thermal management [6]. Additionally, the increasingly larger number of devices connected to a global bus introduces additional verification difficulties. The proposed solution to shared-medium bus problems is one inspired from network engineering: a *Network on Chip*. A traditional bus-based interconnect and a NoC are illustrated in Figure

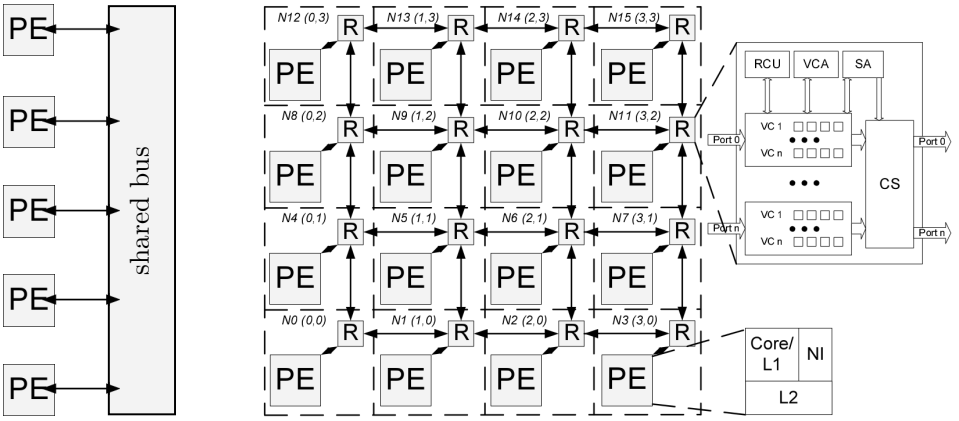


Figure 2.2.: A shared bus based multi-core on the left, and one organized as a mesh-based NoC on the right. Adapted from [67].

2.2, where processing elements (PE) in the NoC are connected by routers and network links, and data is transferred in the form of network packets. By exposing a common high-level interface to all connected units, the NoC can offer a well-structured system with increased productivity and scalability, while still being able to maintain a high-performance communication infrastructure. The TILE64 processor [1] utilizes such networks to connect modular tiles and scale up to 64 cores.

The design and implementation of such a network is an active research area [5, 19] and a complex topic involving design decisions on many levels. Bjerregaard and Shankar [10] investigate these issues with a layered approach in their survey of NoC research and practices, which are briefly summarized below:

- The **Application Level** is concerned with the decoupling of nodes from the network, which allows a modular design approach by abstracting away the details of communication.
- The **Network Level** covers issues dealing with the structure of the network and the movement of data packets inside, such as:
  - Topology, the layout of the network with routers and nodes. 2D grid topologies are by far the most popular.
  - Protocol, the strategy of moving data through the NoC. Store and forward, virtual cut-through and wormhole routing are widely known, with wormhole routing being the most popular [10].

  - Flow control, the mechanism determining packet movement along a network path. Virtual channels (VCs) are widely utilized to avoid deadlock.

  - Other features. Quality of Service (QoS) may be offered for real-time guarantees, broadcast/multicast for mass communication, or more complex operations such as test-and-set for synchronization.

- The **Link Level** regards to node-to-node links and covers a wide range of issues including cross-clock-domain synchronization, pipelining long channels, low-swing drivers to optimize power consumption, and reliable encoding schemes.

2.2.3. Memory

The location of data with respect to a given processing unit that desires to access this data is a major performance determinant – without sufficiently fast data access, a set of heterogeneous computing elements will be of little benefit. Two principal approaches to the design of a multi-core memory system are *shared memory* (coupling all cores to a shared/central memory) and *distributed memory* (coupling each core with its own memory unit), although hybrid approaches are common as well. A body of research on memory system designs for multiple computing units was produced in the late 1980s – albeit for multi-chip (as opposed to multi-core on a single chip) systems, some of the results also apply to multi-core designs since the underlying ideas are similar.

Nitzberg and Lo [44] mention that the primary advantage of a shared memory system is the relative straightforwardness of its design and programmability. However, it has a serious scalability drawback due to the serialization point brought by the common access bus. Distributed memory can offer much higher total bandwidth and does not suffer from this bottleneck, which make it better suited to multi-core systems. In turn, distributed memory is not as straightforward to program as a shared memory architecture.

While using a NoC as described in Subsection 2.2.2 helps address the design challenges for distributed memory, the programming difficulties require additional undertakings. This is commonly addressed by building a shared memory abstraction on top of distributed memory. Referred to as *distributed shared memory*, it entails constructing a coherence mechanism in either

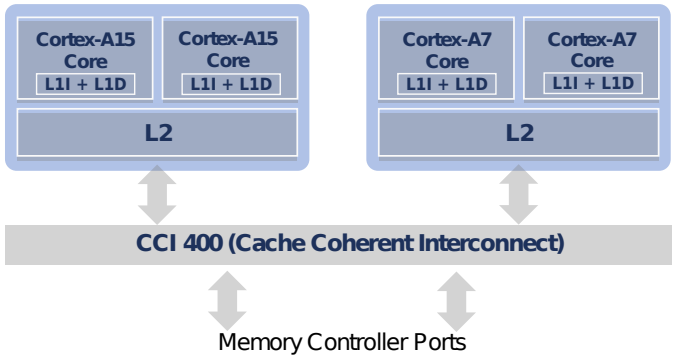


Figure 2.3.: Memory hierarchy in ARM's big.LITTLE AMP solution, with per-cluster L2 cache and per-core L1 cache. Adapted from [26].

hardware or software, and exposing a consistency model to the programmer [44].

Memory systems in many multi-cores today, both in the academy [25, 61] and the industry [1, 26], utilize a hybrid model where distributed memory is mostly found in the form of private caches. Such caches deliver bandwidth with higher scalability to multiple cores, although their performance is bound by memory reference locality. With the developments in 3D integration and Through-Silicon Via (TSV) techniques, efficient and large on-chip distributed memory implementations [38] are becoming possible. This may be important for keeping up with the bandwidth and latency demands of memory-hungry many-core chips.

2.3. The SHMAC Architecture

As the SHMAC architecture constitutes the base for the construction of SHMACsim, it is important to give an overview of the architecture in order to have a better understanding of SHMACsim. A one-sentence description of the architecture would be *heterogeneous computing elements and distributed memory organized into tiles with a mesh NoC interconnect*. This high-level description has been elaborated by Rusten and Sortland [54] to create the initial FPGA-based prototype, and the rest of this section will be a summary of their work, on which SHMACsim is also based.

Tile internal organization: Each tile contains a router, and can contain master and/or slave units, as illustrated in Figure 2.4. A master unit is

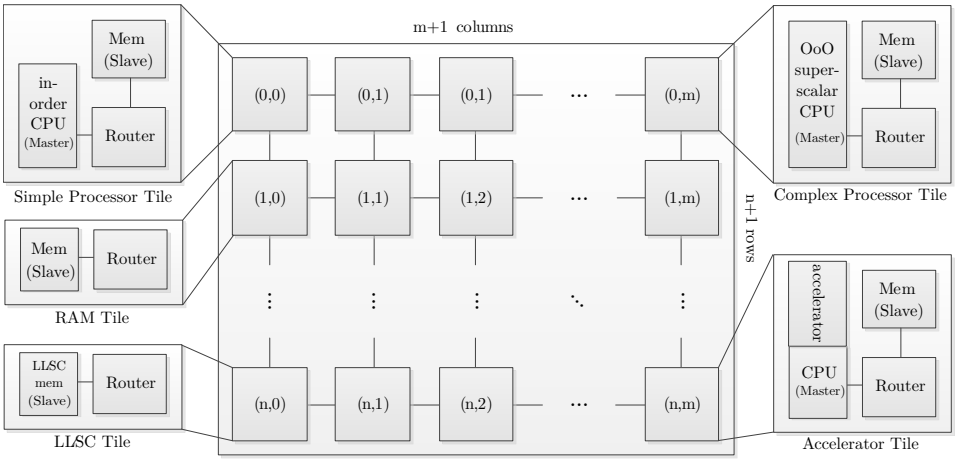


Figure 2.4.: An overview of a SHMAC mesh with a variety of tile types. Tile types and placement are solely for purposes of illustration and do not necessarily reflect a realistic scenario.

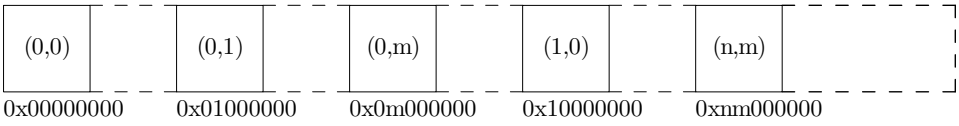


Figure 2.5.: The SHMAC global memory layout, showing the address space partitions for each tile. The most significant 16 bits represent the tile coordinate, whereas the remaining 24 bits act as the local address. Reproduced from [54].

defined as a unit initiating memory requests, and a slave unit is similarly defined as a unit handling memory requests. CPU cores and memory would be typical examples for master and slave units, respectively. The router is responsible for communication between the master and slave units, as well as other tiles. An overview of implemented tile types for the prototype can be found in Table 2.2.

Memory: The global address space is partitioned according to tile coordinates, and each tile's slave unit (if any) is mapped to one, as shown in Figure 2.5. No memory protection is implemented and tiles have free access to each other's address spaces. For tiles containing processors, the program to be executed is loaded into the local tile memory. The prototype does not include any cache memory, making the performance highly sensitive to access patterns.

Tile Type	Master	Slave	Purpose
Integer Processor Tile	mlite	RAM	general purpose, integer computations
Floating Point Processor Tile	mlite +FPU	RAM	general purpose, floating point computations
RAM Tile	none	RAM	random access memory
LL/SC Tile	none	LL/SC unit	synchronization
Jump Tile	none	jump unit	bootstrapping processors
Clock Tile	none	clock unit	measuring execution times
LED Tile	none	LED control unit	LED output
UART Tile	none	UART unit	communicating with the external world

Table 2.2.: Tile types implemented in [54]

Interconnect: The interconnect is responsible for carrying memory requests and responses across units and tiles. A packet-switched NoC is used, with each memory response or request encapsulated into a network packet. The packet format contains information about the nature of the memory operation (request or response), read/write and interrupt/sync flags, the initiator and target of the memory request, and any associated data. The utilized router model is non-pipelined with per-output-port round robin arbitration, and uses dimension order routing. The reader is referred to sections 4.1 and 4.2.3 of [54] for further details.

Processor cores and ISA: The SHMAC FPGA prototype used the mlite core [52] configured with a two-stage pipeline, which uses the MIPS-I ISA. In order to support synchronization, the ISA was extended with the Load Linked (LL) and Store Conditional (SC) instructions. Additionally, a version of the mlite core extended with a floating point unit was provided. The cores were wrapped into a state machine for integration with the SHMAC network interface. This state machine allows a single outstanding memory request; the core is stalled for read requests or LL/SC instructions, but not for write requests.

2.4. Computer Architecture Simulators

Having introduced some of the background concepts in heterogeneous multi-core architectures, it is also interesting to examine the role simulators in exploring HMP design. Computer architecture simulators allow the exploration of the architectural design space and evaluation of design decisions at a relatively low cost. Their primary advantage comes from the ability to create abstract models of the desired architecture and execute this model in a controlled environment. This modelling and execution environment is usually completely in software, although FPGA-accelerated simulation (discussed in 2.4.3) is also gaining momentum.

A brief overview of the underlying concepts in computer architecture simulation and related work will be presented here. Throughout the rest of this section the term *target* refers to the machine being simulated, while *host* refers to the machine running the simulation, which is common terminology for computer architecture simulators [43, 60].

2.4.1. Categorization of Simulators

A number of simulators are used in computer architecture research and related fields, and it can be useful to generalize some aspects of these simulators into categories in order to have a better understanding.

Scope: The simulation targets can vary widely in scope, from the microarchitecture of a single core to multiple interconnected cores and memory units. Full-system simulators capable of booting an Operating System (OS) also exist [8, 39].

Workload type: Simulators can accept different types of input, the two main types being *trace-driven simulators* which run on pre-recorded or generated streams of events, and *execution-driven simulators* which allow execution of software. Trace-driving has the advantage of offering a greater degree of control but the inputs first have to be generated and may be very large in size. Execution-driven simulation has a size advantage, and generating new workloads is faster since the target programs may already exist. Additionally, execution-driven simulation can capture dynamic interactions between instruction streams in multiple processors [18].

Level of detail: All simulators use a model of the simulation target. The level of detail of this model is the most defining characteristic of a simulator, with great impact on the performance and accuracy of the simulation. While it is difficult to precisely specify the level of detail/abstraction for a model, the degree to which elapsed simulation time is modelled is a helpful classification. The IEEE 1666 standard [29] defines the following:

- **Untimed (UT)**: The notion of simulation time is unused. UT models are used for the construction of *purely functional* models.
- **Loosely Timed (LT)**: Each transaction has two timing points; the start and the end. Simulated processes may be temporally decoupled from simulation time.
- **Approximately Timed (AT)**: Each transaction has multiple timing points, simulated processes typically run in lockstep with simulation time.
- **Cycle Accurate (CA)**, also called Pin and Cycle Accurate (PCA): Simulation time is modelled with cycle-level accuracy.

Going down the list, these abstraction levels offer increasing level of detail while sacrificing performance. This trade-off is inevitable since the amount

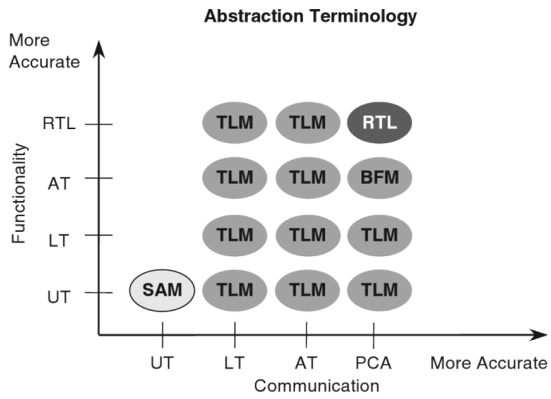


Figure 2.6.: A classification of abstraction terminology for system models. Reproduced from [11].

of necessary computation increases significantly with increasing level of detail. To find an appropriate fit for the simulation requirements, it may be necessary to mix multiple abstraction levels.

For complex system-level designs with a substantial amount of data exchange between components, it is beneficial to separate the communication domain from the functionality domain. Different abstraction levels can be mixed and matched for the two different domains, which is known as *Transaction-Level Modelling (TLM)*. Figure 2.6 presents an overview of abstraction terminology including TLM, System Architectural Model (SAM), Bus Functional Model (BFM) and Register Transfer Level (RTL) model. As the complexity of processors increase, simulators created with a mix of well-defined levels of abstraction will become important to productivity [11].

2.4.2. Multi-core Simulation

The processor development trends mentioned in Chapter 1 are reflected in simulators. Before the advent of the multi-core era, most simulators were focused on the microarchitecture of a single core to develop faster processors. However, the move to multi-cores required tools to evaluate and validate design decisions and implementations. Simulators today are required to have a wider scope, covering multiple cores, complex memory hierarchies and interconnects.

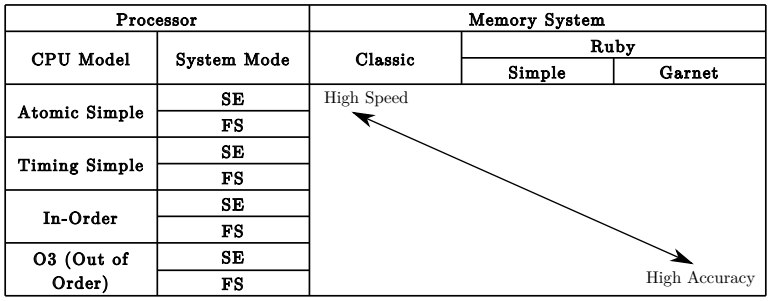


Figure 2.7.: Various processor and memory system models in gem5 with the speed versus accuracy spectrum. Reproduced from [8].

A wide range of computer architecture simulators, both from the industry and the academia, exist today [9, 15, 39, 41, 46]. To give a better understanding of the field in general, two multi-core simulators are briefly discussed below.

2.4.2.1. The gem5 Simulator

A merger of the M5 [9] and GEMS [41] simulators, the gem5 is a simulator widely used for computer architecture research [8]. Since it is not bound to a particular architecture and can instantiate simulations based on different configurations, it is more appropriately called a *simulation framework*. The unmodified Linux kernel can be booted on full-system mode using the ARM, ALPHA and x86 ISAs.

gem5 has a flexible, modular system capable of combining different CPUs, system modes and memory system models to create an appropriate configuration in terms of accuracy and speed, as illustrated in Figure 2.7. The object-oriented nature of the system makes it easy to instantiate multiple cores and there are no inherent limitations on the simulated core count except simulation speed.

- **CPU Model:** Refers to the level of detail for the core model, which can range from *AtomicSimple* always executing an instruction per cycle to *O3*, a detailed out-of-order model.
- **System Mode** can be either *System-call Emulation (SE)*, which avoids the need for peripheral device models, or *Full System (FS)* to model an entire system which can run an OS.

- **Memory System:** refers to the model of memory request-response system, including the fast and easily configurable classic M5 model as well as completely customizable models. For instance, the Garnet [2] models a NoC including router microarchitecture.
- **ISA:** Most commercial ISAs (ARM, ALPHA, MIPS, Power, SPARC, and x86) are supported.

Although gem5 does not inherently target HMPs, there are gem5-based projects [42, 66] targeting heterogeneous systems. These projects make some constraining assumptions regarding the target architecture and seem to be mostly tailored towards CPU+GPGPU systems.

2.4.2.2. Graphite: Distributed Parallel Multi-core Simulation

The origin of the Graphite simulation infrastructure [43] lies in the multi-core simulator scalability problem. The simulation of future processor architectures containing hundreds or thousands of cores will require an enormous amount of computational resources. However, most simulators today are not capable of tackling this task since they have sequential implementations. To address this issue, Graphite provides a parallel, distributed simulation infrastructure that allows functional and performance modelling for cores, on-chip networks and memory systems include coherent cache hierarchies. It is not cycle accurate, but provides good estimates through the use of a collection of models and techniques. A high-level overview of the simulator architecture is presented in Figure 2.8.

As with any kind of parallel application, synchronization is a major problem in multi-core simulation. Even though simulating the multiple cores themselves may appear trivial to parallelise, the need for synchronization arises when these cores interact with each other or with shared resources. Graphite addresses this problem by offering a *lax synchronization* scheme, where synchronization between tiles happens only on explicit events such as application-level locks/barriers, inter-tile message passing and thread spawns/joins. Together with other techniques, this allows almost linear speedup as more host resources are added.

Graphite uses a tile-based target architecture model, where a tile consists of a compute core, a network switch and a part of the memory system (cache and/or DRAM controller). This definition allows for heterogeneity in tiles. Threads on the target application are mapped to tiles and distributed to

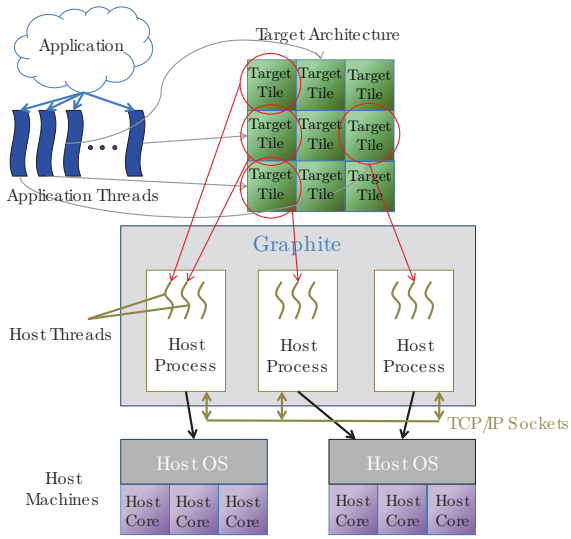


Figure 2.8.: High-level architecture of Graphite. Reproduced from [43].

host machine threads. The host threads themselves can be running on a multi-core machine and/or distributed to several machines across a network.

2.4.3. FPGA Accelerated Simulation

Although software-based simulation has carried computer architecture research a long way, their ability to quickly evaluate points in design space may be deteriorating. Tan et al. [60] argue the move to multi-cores has hampered the performance of pure software-based simulators, since multi-core simulation targets exhibit complex timing-dependent non-deterministic behaviour. Such behaviour needs detailed cycle-level simulation, which in turn requires cycle-by-cycle synchronization. Unfortunately this is notoriously difficult to parallelize (to take advantage of multi-core host machines), and as a result simulator performance has suffered.

This "simulation gap" led to *FPGA Accelerated Model Execution (FAME)* techniques, where the desired target architecture is mapped to an FPGA for evaluation [60]. The FPGA's highly parallel, programmable execution substrate is suitable for the nature of multi-core simulation, and can provide an average speedup of 263x over pure software simulators for detailed models. It is important to note here that FAME techniques do not necessarily run the

target hardware RTL description on an FPGA directly; rather, the FPGA executes a model of the target system, so it may take multiple FPGA clock cycles to execute a single target clock cycle.

3. The SHMAC Simulator Infrastructure

SHMACsim intends to provide a simulation infrastructure for the SHMAC architecture described in Section 2.3, and reflect the FPGA prototype created by Sorten and Rustland [54] with cycle-accuracy. Its construction is the primary task in this assignment, and forms a basis for evaluating micro-benchmarks in Chapter 4 and potential improvements in Chapter 5.

This chapter will provide information about the design of SHMACsim, describing the methodology and reasoning behind development, followed by the structural and behavioural details of the design.

3.1. Methodology

3.1.1. Development Basis

The first step in SHMACsim's development was to decide whether it would be built using an existing simulator as a basis or written from scratch. Three alternatives were considered, summarized below with an overview of advantages and disadvantages.

1. **Customizing an existing simulator** would involve taking an existing simulator or simulation framework, such as gem5 [8], and tailoring it to meet SHMACsim's requirements by modifications and additions.
   - \+ A solid, proven basis for work immediately available
   - \+ Reuse of features, modules
   - – Requires a good understanding of existing simulator structure
   - – Potentially large amount of unnecessary features/overheads
2. **Creating a simulator from scratch** implies designing SHMACsim completely from the ground-up, without using any external modules.

+ Complete control over simulator structure and features
+ The entire development timeframe can be used on implementation instead of studying existing modules and functionality
− Can take an extensive amount of time to get results
− "Reinventing the wheel", existing components may be already perfectly suitable for SHMACsim's needs

3. **Limited use of existing components** is a "best-of-both-worlds" approach, where some existing modules or libraries would be utilized, and the remaining functionality implemented from scratch.
   + A reasonable degree of control over structure and features
   + The most time-consuming or difficult implementations can be imported as pre-made modules
   − Suitable modules must be found
   − May need adapter logic to make everything work together

In light of the simulation infrastructure requirements and the advantages/disadvantages of each approach, the third was deemed most suitable. SHMACsim is built mostly from scratch to reflect the nature of the SHMAC accurately, while making use of suitable existing components and libraries.

3.1.2. Choice of Abstraction Levels

As described in Chapter 2, an HMP consists of a number of interacting subsystems, and SHMACsim aims to model these subsystems for the SHMAC architecture. Each subsystem has a different inherent complexity and impact on the overall system performance, which must be taken into account while modelling.

Table 3.1 lists the level of abstraction each subsystem is modelled in, while the rest of this section provides reasoning on the abstraction level choices.

> **The interconnect** is a resource utilized by the entire system and highly parallel in nature, which can give rise to complicated traffic interleaving and resource contention situations. Without an extensive and time-consuming mathematical analysis of the router, simplified models are likely to miss potentially important details. Additionally,

Subsystem	Abstraction level
Processor cores	Loosely Timed
Memory	Approximately Timed
Interconnect	Cycle-Accurate

Table 3.1.: Chosen abstraction levels for individual SHMACsim subsystems. Please refer to 2.4.1 details on each abstraction level.

Rusten and Sortland [54] mention the interconnect as the major bottleneck in the SHMAC prototype design. Therefore, a cycle-accurate model of the interconnect is used.

The memory units does not exhibit parallel access characteristics, since each slave unit is only directly accessible by its router. However, the wrapper state machines (which stall the processor cores and convert between SHMAC network packets and memory operations) cause delays that can influence the access patterns in the system. Thus, the memory units are modelled on an approximately timed abstraction level.

Processor cores in SHMAC are relatively simple with only a single outstanding memory request allowed, thus a loosely timed model is used for them. This is also appropriate, since as stated in Section 1.4 future SHMAC cores will not be MIPS cores; creating complex models can be a waste of resources.

Considering the cycle-accurate communication model and the loosely timed computation model utilized, SHMACsim could be considered a TLM model bordering on BFM according to the categorization presented in Section 2.4.1.

3.1.3. Choice of External Tools and Modules

Taking into account the decisions on development methodology and abstraction levels, the next step in the design of SHMACsim was to select the set of external tools and modules to be utilized.

The following subsections will describe the selections with regard to their properties and how they correspond to SHMACsim's requirements.

3.1.3.1. Simulation Framework

A software-based simulation kernel or framework that provides the necessary capabilities to model generic hardware is a powerful tool, and a prime candidate for accelerating SHMACsim's development. Indeed, Skadron et al. [58] mention that the manual mapping from the concurrent, structural nature of a computer architecture to the sequential, procedural nature of a programming language such as C or C++ is a complex, ad-hoc and error-prone process.

SystemC, the IEEE standard for system-level modelling [11], was chosen as the SHMACsim simulation infrastructure for the following reasons:

Language of implementation: The Open SystemC Initiative provides an open source reference implementation in C++, meaning that the regular language constructs in C++ are readily available alongside those provided by SystemC. This means that external C++ models can also be integrated with ease. An example would be components from gem5 [8].

Concurrency and Time: Hardware is inherently concurrent and operates at a well-defined rate, thus the ability to model concurrency and time is important for cycle accurate hardware simulation. SystemC offers an event-driven simulation kernel with explicit concepts of passage of time and concurrency.

Abstraction Level of Modelling: SystemC and the associated TLM standard do not enforce a particular level of abstraction for modelling system components, allowing the modelling of different parts of the system with varying levels of detail. The developer is then free to choose the level of modelling for each component in accordance with speed or accuracy requirements of the simulation. On the low-end of the modelling scale, SystemC offers RTL-like language constructs inspired by hardware design languages. One particular advantage of using this level of modelling is the possibility to translate code back and forth between SystemC and HDLs with minimal effort. On the other end of the modelling scale, it is possible to use any abstraction C++ is capable of, thus an instruction decoder could be implemented as a *switch()* statement, or random-access memory modelled with an array. These capabilities correspond to the mixed abstraction level decisions in Section 3.1.2.

3.1.3.2. Processor Core Generation

The construction and verification of processor core models is a potentially lengthy process, which marks it as a strong candidate for utilizing an external module instead of writing from scratch. Since the standard MIPS ISA is utilized in the FPGA prototype, it is relatively easy to find an appropriate pre-made model.

SHMACsim uses an ArchC-generated SystemC model for processor cores. ArchC is a Processor Description Language which can generate SystemC processor models from an ISA behavioural description. Mature models capable of running the SPEC2000 benchmark suite are provided for MIPS-I, PowerPC, SPARC V8, ARMv5 and the Intel 8051 ISAs [4]. Other advantages of using ArchC-generated models are syscall emulation support, TLM memory port support for connecting external memory models, and GNU bintools generation for the models.

This set of features, especially the SystemC model generation, the availability of a mature loosely-timed MIPS-I model and an external memory port, were the reasons for the choice of ArchC. The multiple ISAs supported is also beneficial for future work where a different ISA is desired for the SHMAC.

3.2. Design

Having described the development basis, choice of abstraction levels and external modules, the design details of SHMACsim will be provided in this section. The constructed infrastructure is capable of instantiating desired SHMAC grids and running applications on them. An illustration of the high-level workflow in the system can be found in 3.1. The simulated SHMAC system itself comprises three core subsystems: the processor cores, the memory system, and the interconnect. These are organized into `SHMACTile`s which form a `SHMACGrid`. There is an additional configuration subsystem which is responsible for instantiating and configuring the desired grid, and a toolchain for compiling applications for the architecture is also provided.

The following sections describe these subsystems in some detail. The notation used to refer to design elements throughout the rest of the chapter is given in Table 3.2.

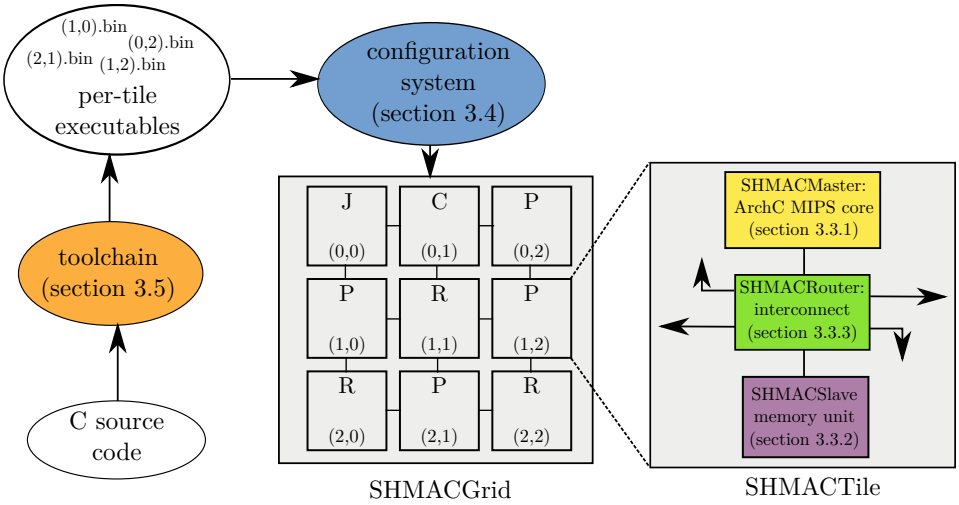


Figure 3.1.: An overview of the simulator core subsystems and helper subsystems in SHMACsim

3.2.1. Processor Cores

SHMACsim utilizes loosely-timed MIPS-I core models generated by ArchC, similar to the Plasma MIPS-I cores used in the FPGA prototype. The model introduces a one clock cycle delay between instruction executions, and although it lacks the two-stage pipeline found in the prototype, the overall system performance was found to be very similar within SHMACsim's scope as indicated by the results in Section 3.5.

The generated cores are configured without any built-in cache or memory except the register file. Instead, memory requests are sent to the on-tile router via the TLM port as described in Subsection 3.2.1.1. Additionally, two extra instructions are implemented from the MIPS-II ISA to support multi-core synchronization, which is described in Subsection 3.2.1.2.

It is worth noting here that only one processor core model is supplied with SHMACsim. The "complex core" enhanced with a Floating Point Unit (FPU) in the FPGA prototype is not implemented since floating point computations can be trivially performed by the host computer in simulation. If desired, individual cores can be made faster or slower by changing the inter-instruction delay in the current loosely timed model, thus providing core diversity. This has not been explored in this report since the assignment does not focus

Type	Notation Example(s)
Class names	`SHMACTLMAdapter`, `shmacsim::packet`
Function names	*receiveMessage*
Shell scripts	***shmacsim-archc-allcompile***
Active objects[1]	
Input-Output Ports[1]	

Table 3.2.: Notation and symbols used in the simulator design description

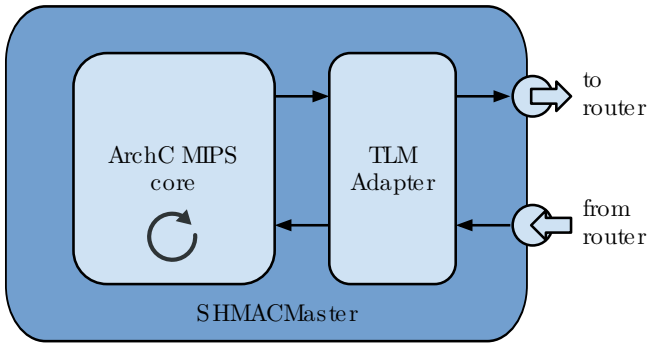


Figure 3.2.: The internal structure of a `SHMACMaster`, showing the ArchC MIPS core and the TLM adapter.

on effects of core diversity, but rather the construction of an infrastructure capable of supporting this diversity.

3.2.1.1. Integration with SHMAC Memory Interface

ArchC-generated cores provide an optional TLM port for accessing memory devices. This port is utilized in SHMACsim to integrate the ArchC-generated cores with the network-based memory interface. An adapter is necessary for this integration, since ArchC uses a bidirectional blocking interface and SHMACsim uses double unidirectional blocking interfaces (described in Section 3.2.2). The `SHMACTLMAdapter` class provides this adaptation by

[1] Active objects contain SC_THREAD or SC_METHODs, and all ports are implemented as SystemC ports. Please refer to [29] or [11] for details on SystemC processes and ports.

translating back and forth between ArchC TLM transport calls and SHMAC network packets. This process works as follows:

1. The ArchC-generated core calls the *transport* function on the `SHMACTLMAdapter` with a memory request
2. The adapter creates a SHMAC network packet corresponding to the ArchC memory request, and sends out this packet to the local router port
3. The core is stalled by a SystemC *wait* call by the adapter while the adapter waits for the reply
4. Upon receiving the corresponding reply packet for this request, the adapter converts the reply into the response format expected by ArchC and returns from the *transport* function call
5. The core's memory request has been served and it can continue execution

It can be observed that `SHMACTLMAdapter` has a similar purpose to the state machine wrapper for the processor cores in the SHMAC prototype - both stall the processor core while waiting on memory requests. The delays introduced by the state machine wrapper are also annotated on `SHMACTLMAdapter` to have a more accurate model of the prototype. Finally, the adapter is also a key part of SHMACsim's LL/SC instruction implementation, as described in 3.2.1.2.

Reflecting the master-slave terminology in the SHMAC architecture, the ArchC-generated cores and the TLM adapter are packed into a `SHMACMaster` class with inbound and outbound memory ports as shown in Figure 3.2.

3.2.1.2. Implementation of LL/SC Instructions

As mentioned in Section 2.3, the SHMAC prototype uses MIPS-I cores enhanced with the capability to execute LL/SC instructions from the MIPS-II instruction set. Together with the LL/SC Tile, these instructions provide a way of lock-free atomic read-modify-write operation. An overview of the process from a LL/SC tile perspective is presented in Section 3.2.2.2. This section focuses on the extension of the MIPS-I ISA with the LL/SC instructions for the ArchC cores.

In the SHMAC network packet format, the LL/SC instructions are required to generate special memory requests (with the SYNC flag set) in order to separate them from regular load and store operations. However, the ArchC TLM port does not support sending out memory requests with attached extra information. One way of addressing this issue would have been to extend the ArchC TLM interface to support sync bits in requests, but a workaround was preferred in order not to make further changes to the ArchC libraries.

The workaround makes use of the explicit LOCK and UNLOCK operations defined by the ArchC TLM interface, which are normally unused in the MIPS ISA implementation. The presence of these operations causes `SHMACTLMAdapter` to behave differently as follows:

> **The LL instruction** causes a sequence of LOCK, READ, UNLOCK memory operations. The `SHMACTLMAdapter` generates a SHMAC read request packet with the SYNC flag set when it detects this sequence.
>
> **The SC instruction** causes a sequence of LOCK, WRITE, READ, UNLOCK memory operations. The WRITE after LOCK causes the adapter to generate a SC network packet, whose reply will contain the success of the SC operation. The subsequent READ does not generate a new network packet, but rather returns the SC success value.

Although the workaround involves apparent multiple memory accesses per LL/SC instruction, only a single network packet is generated and no additional delays are introduced.

3.2.2. Memory Units

The SHMAC model of request-response based memory operations defines any device capable of responding to memory requests as a *slave unit*. These units may implement general-purpose memory or different types of memory-mapped devices.

More details on slave units modelled as part of SHMACsim are given below.

3.2.2.1. Base Slave Unit

The `SHMACSlave` class provides a foundation for SHMAC slave units, as well as modelling a general-purpose RAM. This base implementation provides

the capability to generate response packets for requests and customizable getter-setter functions for accessing memory contents. The general-purpose RAM functionality is modelled as an array being accessed by the getter-setter functions.

In the SHMAC FPGA prototype, the actual memory accesses are invoked via a state machine driven by the network interface signals. To reflect this, SHMACsim uses an approximately timed model where delays are annotated in appropriate locations while memory requests are being served.

It is also desirable to import or export the contents of storage devices as files for examination, implemented by the *importContents* and *exportContents* functions. By utilizing the runtime configuration system described in Section 3.3, the memory array can be loaded with the contents of a binary file before runtime, thus customizing the program to be run on the local tile.

3.2.2.2. The LL/SC Slave Unit

The Load Linked/Store Conditional instructions utilized for synchronization in SHMAC require support from hardware. The hardware constitutes the single point of synchronization by determining the success of SC operations. This is provided by the special slave unit on the LL/SC Tile in the SHMAC prototype, and SHMACsim follows suit by modelling the same slave unit in software. The `SHMACLockSlave` class extends the functionality of `SHMACSlave` with support for LL/SC instructions. This slave unit implements the LL/SC scheme in the following manner:

1. An entry is kept for each processor core in the system, consisting of an address entry and a validity bit.
2. Upon receiving a LL request[1] for an address, the entry for the processor which sent the request is updated with this address, and the validity bit is set.
3. The success of a following SC request[1] from a processor is determined by the validity flag:
    - If the flag is not set, no store operation happens and a response indicating failure is returned to the requester.

[1] The processor cores are responsible for generating messages with LL/SC requests upon LL/SC instruction execution. This is described in Section 3.2.1.2.

- If the flag is set, the store operation succeeds, and all entries containing this address are invalidated.

3.2.2.3. Other Slave Units

The jump and clock tiles in the SHMAC provide bootstrapping and tick counting functionality, which are important for benchmarking purposes. Although their implementation was not necessary in SHMACsim since the functionality can be easily implemented in simulation, simplified versions were implemented in order to evaluate their efficiency and alternative approaches.

The UART slave tile was not implemented since program loading and examining memory contents can be simply done via functions calls on the slave units.

3.2.3. Interconnect

The modelling of the SHMAC NoC is central to SHMACsim. As explained in Section 3.1.2, a cycle-accurate interconnect model is vital due to the complex interactions of multiple cores and distributed memory. Meanwhile, it is also desirable to be able to easily replace the current router model with alternative implementations, thus clear interfaces and abstractions are important. This section describes the design of SHMACsim's network-based memory interface and interconnect model.

3.2.3.1. Network Packets and Memory Interface

Each memory operation in SHMAC is represented as a network packet. The network packets in SHMACsim are represented as C structures of type `shmacsim::packet` and are identical to the packets in the FPGA prototype. Function and operator implementations that operate on this packet type are provided for convenience in debugging and packet tracing.

The concept of memory access is represented by the `SHMACMemoryInterface` interface class, which defines a single pure virtual method *receiveMessage* taking a single `shmacsim::packet` as an argument. Any class that needs to handle network traffic (including masters, slaves and routers) must implement this interface. The interface can be viewed as a TLM unidirectional blocking interface; the object receiving the message is expected to delay returning from

the *receiveMessage* call until the packet has been processed. The semantics of "processed" here are intentionally left vague to allow flexible implementations, for instance buffering locally or simply stalling until the packet reaches its final destination.

3.2.3.2. Router Interface

The routers form the backbone of the SHMAC NoC. The `SHMACRouter` class, intended to serve as a basis for SHMAC router models, contains functions to achieve functionality common to any router model. These were identified as the following three points:

- Given another router, establish appropriate neighbour router port connections.
- Given a SHMAC master or slave unit, establish connections to appropriate router ports
- Given a SHMAC network message, route the message to appropriate destination

Each router model can override these functions to replace them with their own specific implementations while keeping the same interface to other components. For instance, for a functional model, the concept of connecting router ports is simply be binding SystemC ports to an object implementing `SHMACMemoryInterface`. The cycle-accurate model port connections, on the other hand, require connecting three ports (request, data and acknowledge) with appropriate signal types.

In the base `SHMACRouter` implementation, a simple message routing scheme with zero delays/infinite bandwidth is modelled to verify the functional system model. The message routing here consists of calling the *receiveMessage* function on the next router along the message path.

3.2.3.3. Cycle-Accurate Router Implementation

The SHMAC prototype routers are composed of six input and output ports (four cardinal directions plus two for on-tile master and slave), a round-robin arbiter per output port, and a crossbar switch. Figure 3.3 illustrates the interplay between these components during routing. The strict handshakes between router components and lack of pipelining are notable in this design.

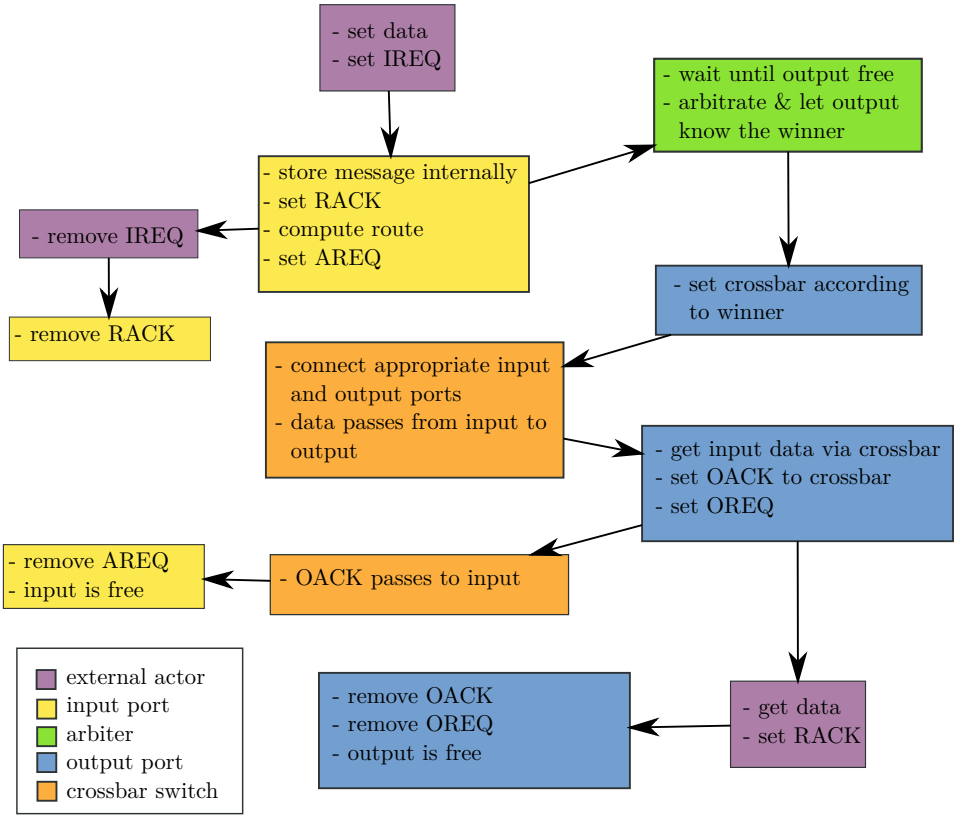


Figure 3.3.: An overview of the routing process in the cycle-accurate router model, showing the actions taken by different router components. Arrows indicate causality, while the colors indicate the actor executing each action.

Since the cycle-accurate model is a replica of the router model from the FPGA prototype, the reader is referred to [54] for further design details on the router.

The approach for creating the cycle-accurate interconnect for SHMACsim was direct, manual translation of all router components from the original VHDL descriptions into SystemC. Thanks to the HDL-like language constructs in SystemC, this process is relatively straightforward and less prone to errors compared to creating a model from scratch. Each router component (namely input port, arbiter, output port and crossbar switch) was verified with a testbench as described in Section 3.5.

After the CA model was constructed, adapters were introduced to bridge the signal-level router interface and the high-level `SHMACMemoryInterface` definitions. These adapters do not introduce any delay of their own and thus do not impact the model accuracy.

3.2.3.4. Network Construction

Each tile is responsible for connecting its master and slave units to the on-tile router, performed by the constructors in the `SHMACTile`-derived classes. Connecting the tiles into a mesh network is performed by the `SHMACGrid` class, which calls the router connection methods for neighbouring routers. Since all actual connections are established by function calls on the router, the nature of the connections is abstracted away from the upper-level organizational units such as tile and grid.

3.3. Configuration System

In order to constitute a practical tool for design space exploration, it is necessary to be able to configure SHMACsim with different HMP configurations. The SHMAC architecture is structured around the idea of forming a grid from available heterogeneous tiles, and the SHMACsim configuration system reflects the same paradigm: the types of available tiles are specified, the grid layout is composed from the available types, and tile-specific configuration is applied on each tile. To easily create different experiment setups and avoid re-compilation, the major part of configuration takes place at runtime via passed command line arguments to the SHMACsim executable.

Figure 3.4 presents a graphical overview of how the configuration system works, with more details on the aspects of configuration in the following subsections.

3.3.1. Tile Types

The singleton `SHMACLibrary` class is responsible for tile type configuration. It contains set of "recipes" for building each tile type. Each recipe is described as a callback function and registered with `SHMACLibrary`, with a single character identifying this tile type. This recipe describes the creation of appropriate tile, master and slave subclasses, and can also contain a custom-defined

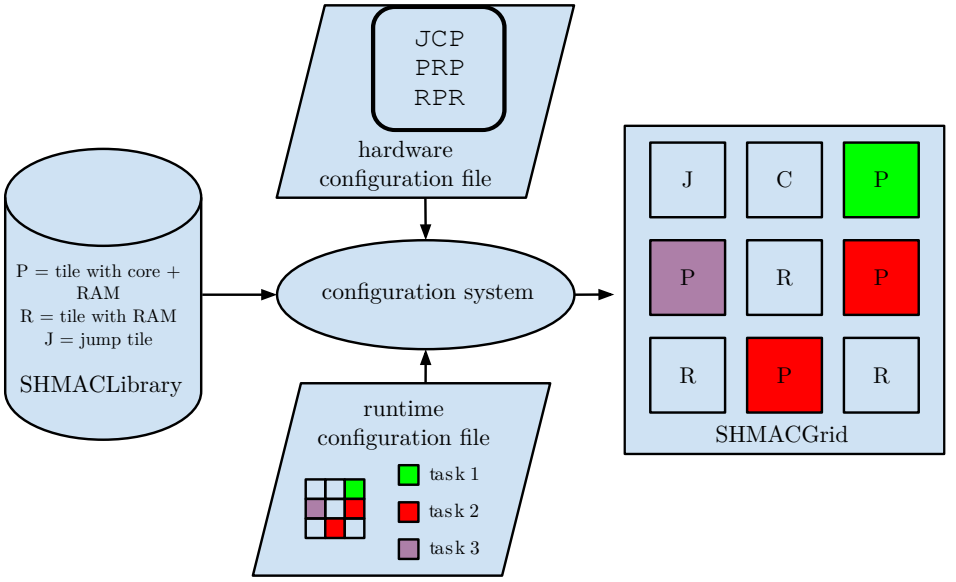


Figure 3.4.: A graphical overview of the SHMACsim configuration process.

initialization step which will be executed right before runtime, which allows the configuration of per-tile runtime behaviour. Tile type configuration is done in code and recompilation is necessary for new tile types or modifications to existing ones.

A list of tile types implemented in SHMACsim and the corresponding identifiers are presented in Table 3.3.

3.3.2. Tile Layout

A hardware configuration file is passed to SHMACsim via the -t command line option. Formatted as a grid of characters, it describes the desired tile layout for a simulation instance. Each character in the grid must correspond to a registered tile type in `SHMACLibrary`. At startup, SHMACsim parses this file and uses `SHMACLibrary` to instantiate each tile, which are contained in a `SHMACGrid`. Since the hardware configuration file is a command line parameter, no recompilation is necessary to change the tile layout.

ID	Master Unit	Slave Unit	Description
R	none	RAM slave (16 MB)	general purpose memory tile
C	none	clock slave	Clock tile for timing benchmarks
J	none	jump slave	Jump tile for bootstrapping
L	none	LL/SC slave	For multi-core synchronization
P	ArchC MIPS core	RAM slave (16 MB)	ArchC MIPS core and memory

Table 3.3.: An overview of tile types implemented in SHMACsim. A list of tile types in the FPGA prototype is provided in Table 2.2 for comparison.

3.3.3. Runtime Configuration

The runtime configuration file contains zero or more key-value pairs for each tile coordinate. Each tile instance contains a metadata storage, into which these key-value pairs are parsed and loaded. The tile recipe can then make use of the tile instance metadata to configure the tile as desired. The most obvious use of runtime configuration is the association of executable files with each tile. Another usage is the adjustment of heap pointers for the ArchC MIPS cores utilized in SHMACsim. This metadata storage can be used to implement any scheme that requires per-tile configuration data before the simulation starts.

3.4. Toolchain and Utilities

SHMACsim is an execution-driven simulator, which takes compiled programs as input. The ArchC processor cores utilized in the current version support Executable and Linkable Format (ELF) executables, which can be produced from C code using the ArchC-provided MIPS toolchain. SHMACsim intro-

duces some changes to and some extra utilities on top of this toolchain, which will be described here.

The ArchC MIPS toolchain is a standard ***mips-elf-gcc*** toolchain with a custom linker script, some assembly code for runtime initialization and a statically linked library mapping the C runtime functions to ArchC syscalls. Using this toolchain as a basis, SHMACsim provides its modified toolchain, utilized by calling the ***shmacsim-archc-compile*** shell script. Compilation via this script is identical to compilation via gcc, except the tile coordinates for which the program is desired to be compiled must be specified. This is mostly necessary due to the cache-free nature of the current SHMACsim; performance is highly sensitive to data location. To maximize performance, the stack, heap and code sections for each program should be placed in the local memory of the tile executing that program, which is ensured by the linker script generated by the ***shmacsim-archc-compile*** script. Finally, the script also calls ***mips-elf-objcopy*** to generate raw binary files from the produced executables, since SHMACsim does not currently support ELF parsing.

Two more utilities delivered with SHMACsim are worth noting here. The first one, ***shmacsim-archc-allcompile***, is a simple iterator script which calls ***shmacsim-archc-compile*** with the same gcc arguments for each tile in a given interval. This is useful to compile the same application for a large mesh. The other notable script is ***shmacsim-run-benchmarks***, which executes a given set of SHMACsim benchmark configurations in parallel, allowing higher simulation performance and faster design space exploration on multi-core host systems.

Code listings for the utility scripts are provided in Section A.1 in the Appendices.

3.5. Testing and Verification

As for any complex system, it is necessary to ensure that SHMACsim operates as desired by employing a number of testing and verification strategies. Due to time constraints, only the processor cores and the interconnect could be independently verified; the behaviour of the remaining components is tested as part of system-wide testing. More details on the tested/verified aspects of the system are provided below.

Processor cores: The utilized ArchC-generated MIPS cores are already verified using SPEC2000, Mediabench and MiBench [4]. However, since the cores have been extended with LL/SC instruction implementations, selected benchmarks from Mediabench were re-run as regression testing. The LL/SC instruction implementations themselves were tested using the micro-benchmark in Section 4.6.

Routers and Interconnect: The VHDL to SystemC translation approach taken for the CA router model should make it correct by construction, but additional verification was also performed to ensure proper cycle level behaviour. Towards this, each router component testbench was also translated from VHDL to SystemC and the behaviour verified by comparing VHDL and SystemC signal traces. Finally, the assembled router model was tested with a translated testbench and verified to reflect the routers in the FPGA prototype with cycle accuracy.

Assembled system: Exhaustively verifying a configurable simulator system like SHMACsim is very difficult given the size of the state space, and quite infeasible within the time-frame of a Master's Thesis. Therefore, a pragmatic testing strategy is applied, where correctness is determined by correct end results from execution. This strategy is applied in the form of micro-benchmarks and is covered in Chapter 4. The correctness is asserted by comparison the benchmark results against the expected end results, as well as by comparing against the micro-benchmark results from the SHMAC prototype for the micro-benchmarks in Section 4.4.1.

Toolchain: The ArchC-provided MIPS toolchain is based on GNU Compiler Collection (GCC) and does not require extra verification. However, SHMACsim makes modifications to the initialization assembly code and the linker script to place programs' heap, stack and code sections on the local tile memory. To verify that all instruction fetches are read from the local memory range, assertions have been placed inside the ArchC MIPS cores. It should be noted that these checks are only to verify the expected linker script behaviour and cores are able to perform instruction fetches from any location if desired.

4. Evaluating the SHMAC: Micro-benchmarks

Benchmarking is a powerful tool for evaluating a computer architecture. Owing to the mature ArchC core model and syscall emulation, SHMACsim is already able to execute larger benchmark suites and complex programs [4]. However, given the relatively early stage the SHMAC architecture is in, micro-benchmarks [1] and detailed analysis of their results can be considered more helpful for assessment of architectural features and identifying possible improvements.

In order to carry out an evaluation of the current state of SHMAC architecture and highlight its strengths and weaknesses, a set of micro-benchmarks were implemented and executed using SHMACsim, and the results were analyzed. Some of these benchmarks serve the additional purpose of comparing SHMACsim's accuracy to the prototype, thus validating the simulator implementation.

Section 4.1 contains a description of the methodology for benchmark implementation and measurements. The rest of the chapter describes each implemented micro-benchmark, the results obtained from execution, and a discussion of the results. An overview of implemented benchmarks can be found in Table 4.1.

4.1. Methodology, Metrics and Notation

The benchmarks described in this chapter consist of one or more tasks executing on a SHMAC grid, which can consist of different tile types. Each benchmark was implemented in C, compiled using the utilities described in Section 3.4, the hardware and runtime parameters configured via the SHMACsim configuration system described in Section 3.3 and executed.

[1] A micro-benchmark here is defined as measuring the performance of a very small, specific piece of code

Benchmark Name	Section	Primary Assessment
Clock Tile Access Time	4.2	Impact of traffic on execution time measurements
Tile Startup Delays	4.3	Impact on grid size on tile boot latency
Single Master Memory Access	4.4.1	Performance of local and neighbour memory accesses by one master
Multi-master Memory Access	4.4.2	Performance of remote accesses by multiple masters
Remote Impact on Local Fetch	4.5	Impact of remote accesses on local instruction fetch performance
Lock Acquisition Time	4.6	Performance of LL/SC tile for synchronization in large grids

Table 4.1.: An overview of implemented micro-benchmarks

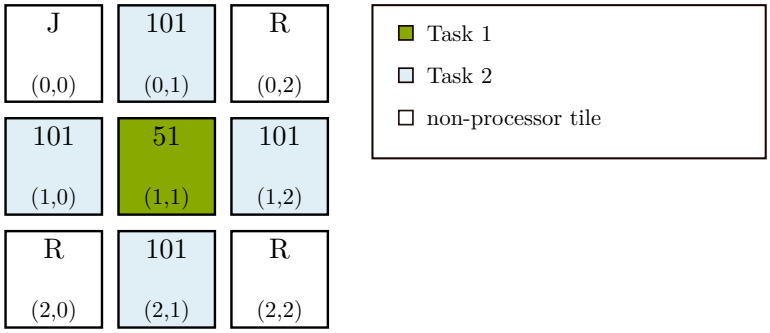


Figure 4.1.: Benchmark notation

The implementations consist of a *common part* which performs setup and reporting functions, and a *kernel part* which contains the actual code desired to be benchmarked. The kernels mostly consist of a single repeated MIPS instruction, with sufficiently high number of repeats to obtain stable results. Appendix A.2 contains code listings for each benchmark.

The clock counter value (elaborated in Section 4.2) is read right before and right after the benchmark's kernel is executed. The number of clock cycles elapsed (referred further on simply as *cycles*) and cycles divided per instruction count (referred to as *Cycles Per Instruction (CPI)*) are used as the primary metrics for benchmark results.

In Section 4.4.1, cycle count is used as a basis for accuracy comparison against the SHMAC FPGA prototype. The metric for accuracy comparison is the relative error of simulated cycles with respect to FPGA cycles, whose calculation is shown in Equation 4.1.

$$\Delta\% = \frac{Cycles_{sim} - Cycles_{FPGA}}{Cycles_{FPGA}} \cdot 100 \tag{4.1}$$

The notation used to express the benchmark configurations and results is as specified in Figure 4.1. All processor tiles are uniform in terms of hardware and execute a single task indicated by their background colour. Non-processor tiles have a white background and can contain different slave units. The performance of each task to be benchmarked is specified on each tile in terms of CPI.

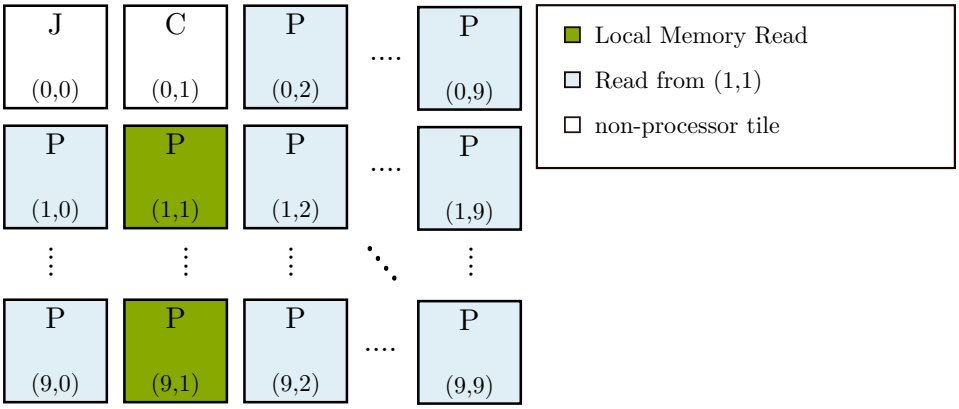


Figure 4.2.: Configuration for the clock tile access time benchmark. CPI values are omitted, see text and Figure 4.3 for results.

4.2. Clock Tile Access Time

A reliable measurement methodology is vital to the process of benchmarking. Ideally, the observed benchmark should not alter the correctness of the measurement, and neither should the measurement process alter the results. This micro-benchmark intends to test the clock tile-based measurement approach in this context, where the clock tile is read before and after a micro-benchmark kernel and subtracted to get the cycle count.

Like all tiles, the clock tile is subject the Non-Uniform Memory Access (NUMA). Reading its clock counters takes a varying amount of time proportional to reader-clock distance, and the traffic along the path. Delay due to distance is constant, and does not influence start-to-end cycle count measurements. However, traffic delay is a dynamic parameter with a potentially different impact on the start and end readings, which this benchmark aims to model.

Figure 4.2 illustrates the benchmark setup. In the 10×10 grid, the clock tile is located at (0,1) and all tiles located "right below" ($j = 1$) execute the Local Memory Read kernel. All other tiles are processor tiles and read from the memory of the tile at (1,1). This creates a dense traffic throughout the mesh. Finally, a clock counting system register is built into each processor, as described in Section 5.2. The tiles performing Local Memory Read use two different methods to get clock count values: reading from the clock tile at (0,1) and reading from the system register.

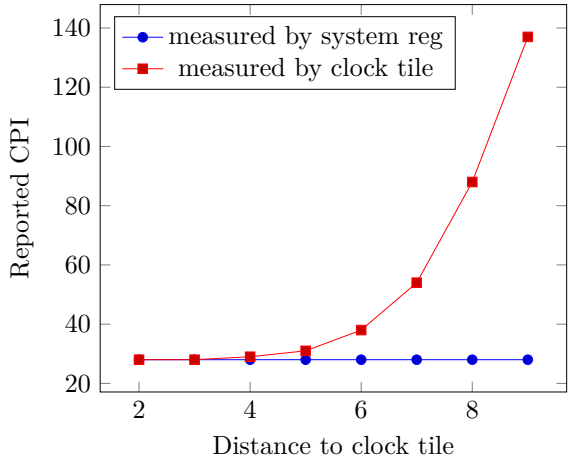


Figure 4.3.: Plot of CPI measurement skew versus distance to clock tile, under heavy traffic.

Figure 4.3 presents a comparison of CPI measurements from the clock tile versus those from the internal clock count register. The register-based measurements remain constant at 28 CPI, consistent with the findings in Section 4.3. As the distance to the clock tile increases, the measurements from the clock tile are skewed dramatically. This is due to the heavy traffic on and around the (1,1) tile, which the clock tile read requests have to pass through.

Although the situation created here is extreme and rather unlikely to happen in real life-scenarios, skewed cycle measurements under traffic in a large mesh remains a possibility. This does not constitute a problem in simulation where cycle counts from the simulation kernel are readily available, but hardware implementations using the clock tile do not have this advantage. To avoid false time measurements in hardware, the system register file-based clock counter implementation described in Section 5.2 is proposed.

The benchmarks in this chapter are timed using the system register method. Section 4.4.1 is an exception and uses a clock tile, since it is desirable to have a thorough comparison with the SHMAC FPGA prototype and since this benchmark does not have dynamic traffic.

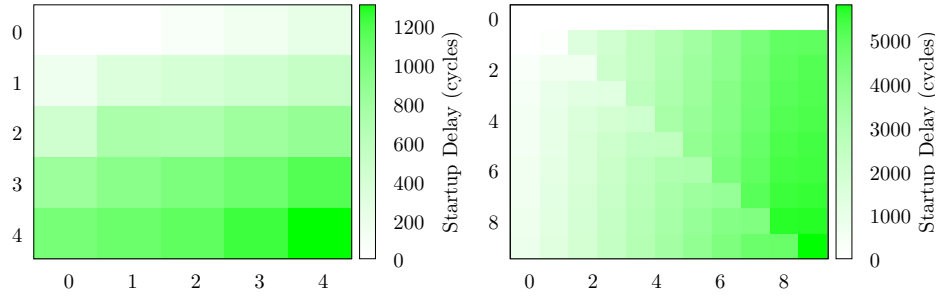


Figure 4.4.: Startup delays due to jump tile at (0,0) in 5×5 and 10×10 grids. Cycles given relative to earliest starting processor.

4.3. Tile Start-Up Delays

The current bootstrapping method in SHMAC is the so-called jump tile approach. The jump tile is placed at (0,0) corresponding to global memory address 0, where all cores start loading instructions from. The jump tile generates a series of MIPS instructions which causes a core to start reading from the beginning of its own local memory. While this method enables booting multiple cores without any changes to the cores themselves, it can generate a lot of memory accesses targeting a single address at boot. This benchmark aims to measure the behaviour of large SHMAC grids booting in this manner.

The benchmark configuration is very simple. It includes a single jump tile at (0,0) and processor tiles everywhere else in the grid. The processors run a kernel which simply reports the cycle count at which it starts running and exits.

Figure 4.4 contains heat-map representations of the results from 5×5 and 10×10 grids. A general trend can be observed where processors located further away from the jump tile take longer time to boot, up to 5780 cycles for (9,9) compared to the earliest booting processor at (0,1). The delay pattern, however, is not completely dependent on Manhattan distance of the processor tiles – for instance, the processors under the diagonal have lower boot times. This is likely due to an interaction of the timing of serving memory requests and dimension order routing. It should be kept in mind that this pattern is not specific to the jump tile, but applies to all cases where multiple cores do simultaneous memory access on a single tile.

To conclude, the complications are manageable in terms of boot delay. Some start-up delay is always expected in large systems, and processors can be made to wait until the whole system is up and running. Placing the jump tile in the middle of the grid can give better load balancing and results. Alternatively, the per-tile system register approach described in Section 5.2 can provide integrated, constant-time bootstrapping.

The benchmarks in this chapter use the integrated bootstrapping method. Section 4.4.1 is an exception and uses a clock tile, since it is desirable to have a thorough comparison with the SHMAC FPGA prototype.

4.4. Pure Memory Access Performance

Multi-core processors are often limited by the memory bandwidth visible to the cores, which was discussed in Section 2.2.3. An evaluation of basic memory read-write performance can thus yield useful insights into a multi-core architecture. The following subsections will cover the evaluation of memory access performance first for a single master, and then for multiple masters sharing a memory unit.

4.4.1. Single Master

In the SHMAC prototype report, three micro-benchmarks, namely *Local Memory Read*, *Neigbouring Tile Read* and *Neighbouring Tile Write*, were introduced [54]. They measure the performance of a single master performing memory operations on local and neighbour memory units. The extracted benchmark kernels have been executed on SHMACsim, both as a basis for comparison in terms of accurately reflecting the prototype's performance, and for performing a more in-depth analysis of the results.

The types of kernels used are also used as building blocks for other benchmarks. The descriptions of these kernels and the corresponding MIPS assembly code can be found in Table 4.2.

The benchmark configuration is as depicted in Figure 4.5. The three micro-benchmarks have no interaction with each other, and are executed on the same grid only for convenience. Each kernel is run two thousand iterations to be consistent with the methodology in the FPGA prototype's evaluation [54].

Kernel Name	MIPS Assembly	Description
Local Memory Read	`addiu $4, $4,1`	A register-to-register instruction, essentially measuring local instruction fetch performance
Neighbour Read	`lw $5,`*`neighbour_addr`*	Reads a word from a neighbouring tile's memory
Neighbour Write	`sw $5,`*`neighbour_addr`*	Writes a word to a neighbouring tile's memory

Table 4.2.: Descriptions and the repeated MIPS assembly instruction for the micro-benchmark kernels utilized in this chapter

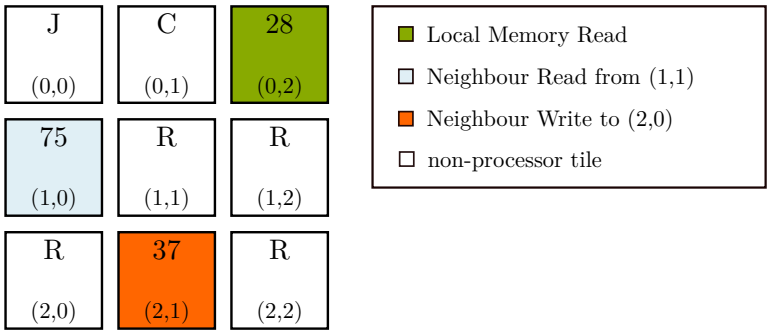


Figure 4.5.: The Pure Memory Read-Write Performance benchmark

Benchmark	Cycles			CPI	
	SHMAC	SHMACsim	Δ%	SHMAC	SHMACsim
Local Mem Read	58038	56270	-3.0	29	28
Neighbour Read	156038	150270	-3.7	78	75
Neighbour Write	80038	76270	-4.7	40	38

Table 4.3.: Comparison of pure memory read-write benchmarks between SHMACsim and the SHMAC FPGA prototype.

Benchmark	CPI	
	Instruction Fetch	Neighbour Operation
Local Memory Read	27	0
Neighbour Read	27	47
Neighbour Write	31	22

Table 4.4.: Breakdown of memory packet lifetime for pure memory read-write micro-benchmarks. Time spent inside the processor core itself (1 cycle) is not included.

The results can be found in Table 4.3. Presented alongside are the reported results from the FPGA prototype. It can be observed that the simulation results are within 5 % of the prototype.

As can be expected, both neighbour read and neighbour write operations take significantly longer than local memory reads. This is due to the packets in neighbour read/write operations travelling a longer distance (in terms of router hops) compared to the local memory. The neighbour write operation is much faster than the neighbour read, since the write operations do not cause the processor core to be stalled.

It is also interesting to see a breakdown of the overall CPI for these benchmarks. Using the packet tracking capabilities introduced in Section 5.3, memory packets corresponding to instruction fetches (both requests and responses) and neighbouring tile memory operations in the micro-benchmark were tracked. The results are presented in Table 4.4.

The neighbour operation for writing takes approximately half as long as reading, since write requests continue without waiting for a reply packet.

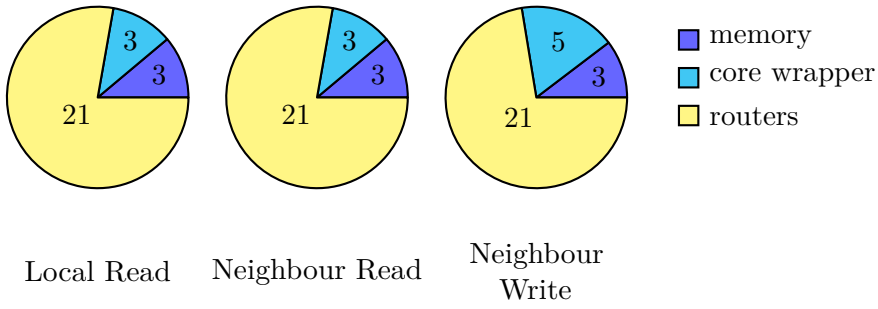


Figure 4.6.: Breakdown of instruction fetch time for single-master memory performance

One might expect that the instruction fetches always take the same amount of cycles in Local Memory Read, but this is not the case. While equal for Local Memory Read and Neighbour Read, instruction fetches for Neighbour Write are actually slower.

A deeper breakdown of instruction fetch packet lifetime, presented in Figure 4.6, reveals a longer stalling time in the processor-network interface. This can be explained by the non-stalling nature of the write operations. The processor immediately issues another instruction fetch after the write operation, but the router input port is not yet ready for receiving a new message due to the lengthy handshakes in the router.

The results from this benchmark are in line with those from Rusten and Sortland [54], and the breakdown of packet lifetime clearly reveals the router as the single greatest bottleneck. For the simple memory access operations tested here, up to 75 % of the time is spent in the router, which marks the router an obvious candidate for further optimization. The introduction of a faster, pipelined router is very likely have a great impact on performance.

Another conclusion that can be drawn here is that the memory bandwidth usable by a single master is much lower than the memory's own saturation bandwidth. The 3-4 cycle delay per memory operation is far exceeded by the 20+ cycle router delays. Adding several masters using the same memory could improve resource utilization for the slave units. The dual-port slave and router bypass improvement proposed in Section 5.1 aims to solve this problem by introducing low-latency local memory access.

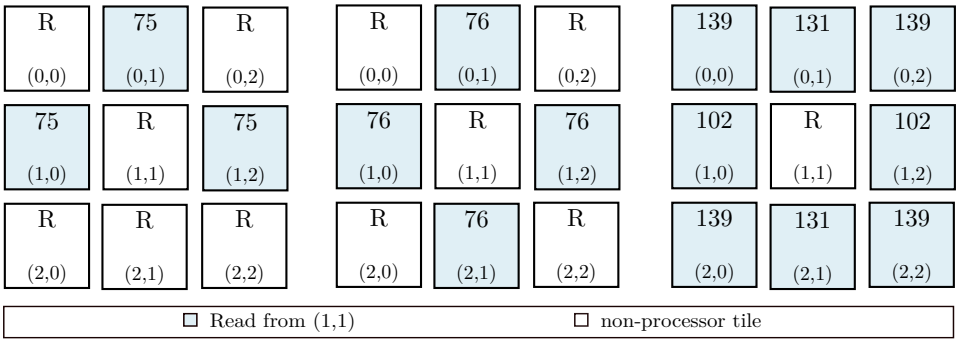


Figure 4.7.: Configurations and results for three, four and eight masters reading from the central RAM tile.

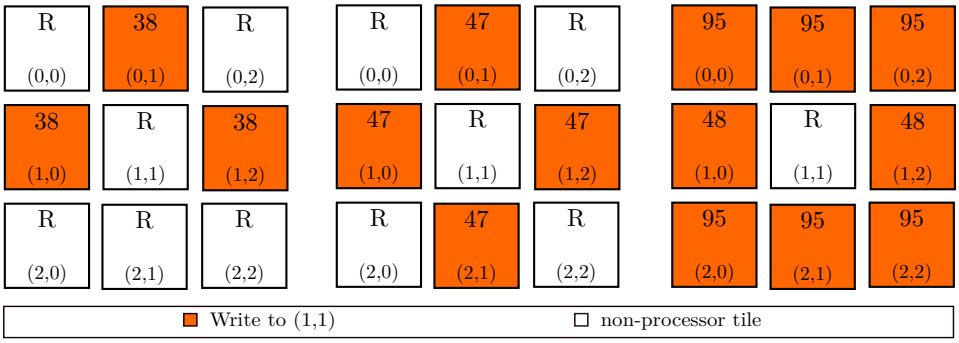


Figure 4.8.: Configurations and results for three, four and eight masters writing to the central RAM tile.

4.4.2. Multiple Masters

Following the conclusion in Section 4.4.1 that a single master cannot utilize the full bandwidth available from a memory unit, it is desirable how sharing by multiple masters affects memory performance under the current conditions. In order to observe this, configurations where multiple masters are reading or writing to the same tile were created. The kernels used are identical to Neighbour Read and Neighbour Write from Section 4.4.1. The configurations and results for three, four and eight masters are shown in Figure 4.7 for reads and Figure 4.8 for writes.

Figure 4.9 presents a plot of average read and write CPI performance for an increasing number of masters. It can be observed that the performance for both reading and writing remains constant, for up to and including three

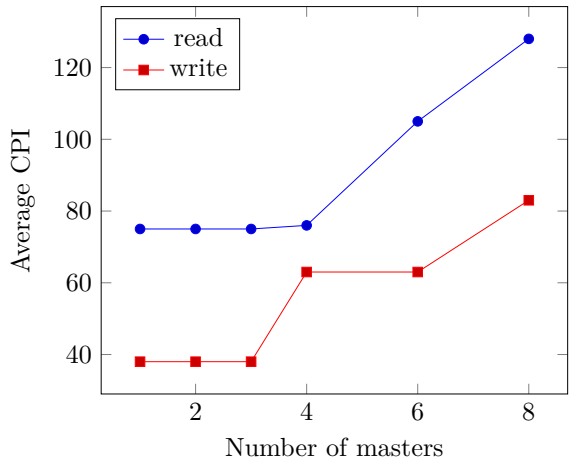


Figure 4.9.: Plot of average read and write CPI with an increasing number of masters accessing the same RAM tile.

masters. Significant deterioration in performance is apparent for four masters and more, which may be an indication of resource contention on the memory unit.

The results indicate that a memory tile can be shared by at most three masters under sustained load, without any performance penalties. Lower load and different timing characteristics for the memory requests are likely to allow even more masters. However, care must be taken not to assign too many masters to operate on a single RAM tile, since significant slowdowns will occur when the memory unit is saturated.

4.5. Remote Impact on Local Fetch

The current SHMAC architecture has a memory system where all tiles are able to directly access memory units from all other tiles, without any restrictions or memory protection. This can be convenient for accessing and sharing data, and the results from Section 4.4.2 encourage sharing a memory unit between multiple masters. However, sharing the on-tile scratchpad RAM for a processor tile may have additional consequences on the performance of the local core, which the Remote Impact on Local Fetch (RILF) benchmark attempts to measure.

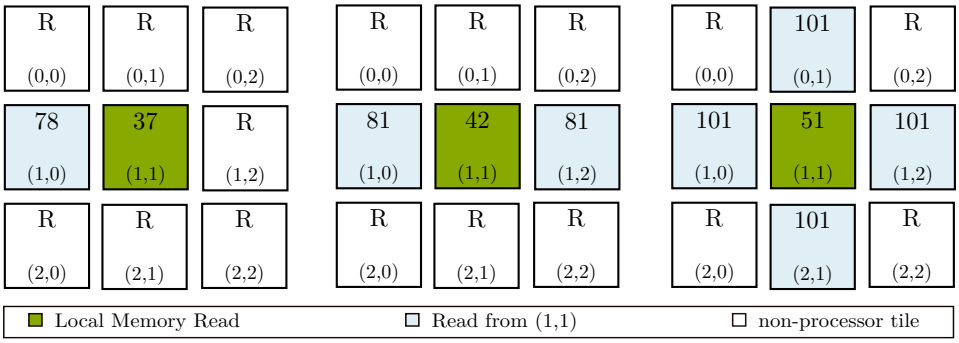


Figure 4.10.: Remote Impact on Local Fetch with one, two and four remote reading tiles

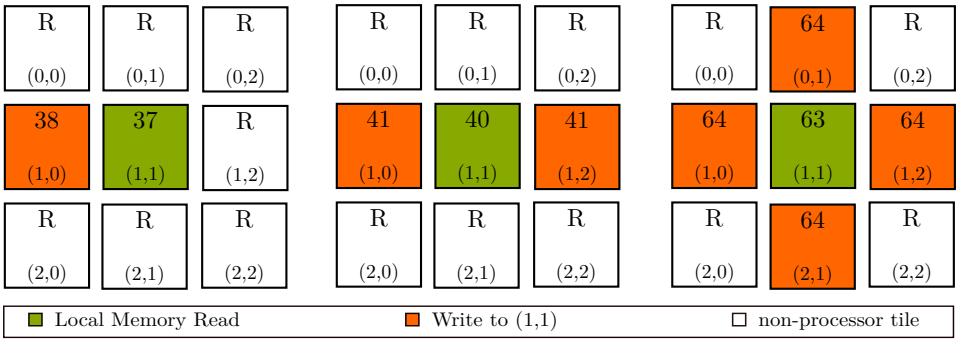


Figure 4.11.: Remote Impact on Local Fetch with one, two and four remote writing tiles

This benchmark combines the Local Memory Read and Neighbour Read-/Write kernels from Section 4.4.1. While the center tile is running the Local Memory Read kernel, one or more neighbouring tiles simultaneously read from or write to the central tile's memory. It is thus possible to observe the impact of remote processor accesses on local instruction fetch performance.

Figures 4.10 and 4.11 depict three RILF configurations with an increasing number of neighbours accessing the central tile. The negative impact of remote memory accesses on local instruction fetch performance is evident. The performance of remote accesses also suffers, though to a lesser degree and in a similar pattern to findings in Section 4.4.2.

Figure 4.12 contains a plot comparing the base instruction fetch performance of 28 CPI (from Section 4.4.1) with the instruction fetch performance of tile

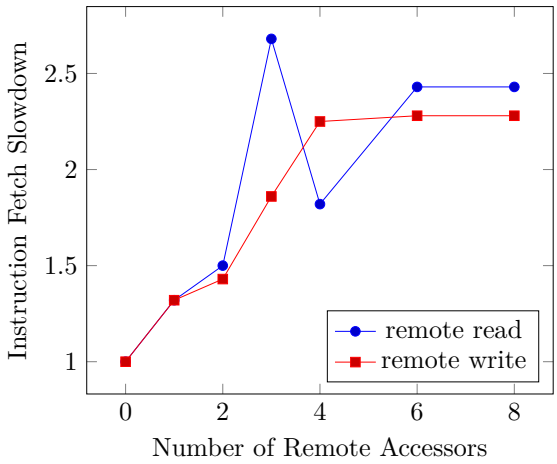


Figure 4.12.: Plot of instruction fetch slowdown in central core with an increasing number of masters accessing its on-tile memory. Baseline instruction fetch performance taken as 28 CPI, from the results in Section 4.4.1.

(1,1) in different RILF scenarios. The slowdown is almost linearly related to the number of remote accessors for up to 4 writers or 6 readers, after which it reaches a plateau. Also notable is the sharp spike with three remote read accessors, causing a dramatic drop in instruction fetch performance. This is assumed to be due to a combination of arbitration and ordering of requests resulting in the instruction fetches always being served last, though deeper analysis was not possible due to time constraints.

To conclude, sustained data reads or writes to on-tile memory have a significant effect on the local instruction fetch performance and should be avoided for tiles running performance-sensitive applications. The impact should be remediable by giving higher priority to on-tile master requests during arbitration, or by introducing instruction caches.

4.6. Lock Acquisition Time

The LL/SC tile provides multi-core synchronization for SHMAC systems as described in Section 3.2.2.2. The current hardware implementation uses a single LL/SC tile as the point of synchronization. On the software side, the SHMAC utilizes a locking function whose behaviour is illustrated in Figure

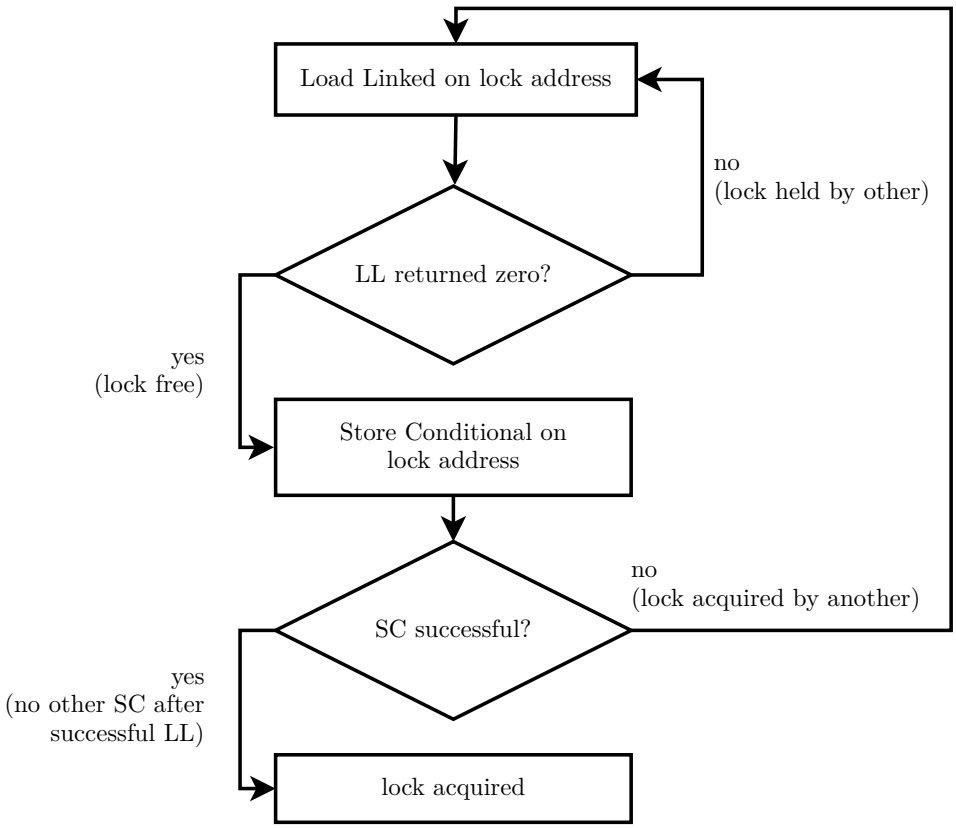


Figure 4.13.: Flowchart illustrating the current SHMAC lock acquisition routine in software

4.13. While the current approach has been verified to provide the desired synchronization capabilities both by Rusten and Sortland in [54] and in this work, it is interesting to see how this approach scales for larger grids. This benchmark aims to observe how long it takes for processor tiles to acquire an LL/SC-based lock while the rest of the grid is also doing the same.

The benchmark configuration is described on a 3×3 grid [2] in Figure 4.14. An LL/SC tile is located in the center of the grid, all other tiles execute a kernel which attempts to lock a memory location on the LL/SC tile using the scheme in Figure 4.13. The lock is released once successfully acquired, and the benchmark kernel terminates.

[2] The scenario is also applicable to larger grids where the entire grid is attempting to lock a central LL/SC tile

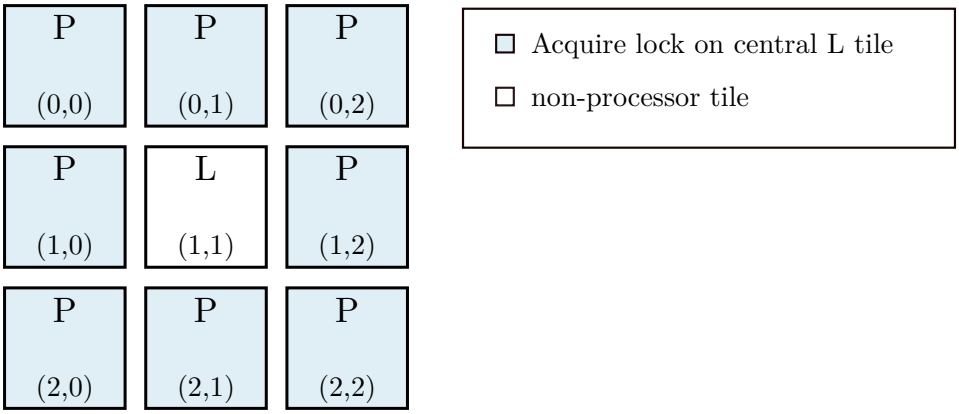


Figure 4.14.: Configuration for a 3×3 grid evaluating the lock acquisition time. CPI values are omitted since another metric is utilized.

The CPI values are not utilized in this benchmark; instead, the number of total LL operations plus the number of failed SC operations from each tile is used as a metric. This allows comparison of lock acquisition times in a more coordinate-independent way, since memory access delays themselves are not included. The looping nature of the lock acquisition scheme should be kept in mind here; a tile does not stop trying to acquire the lock after a number of retries. Instead, it restarts the lock acquisition operations, and thus keeps generating memory traffic.

Figure 4.15 illustrates the distribution of lock acquisition failures for relatively large SHMAC grids. The lock acquisition times observed are not proportional to the tile distance to the center, but instead seem somewhat random. This suggests that the lock acquisition times are dependent on the dynamic traffic generated in the process. It is also visible that increasing the grid size by a factor of $81/49 \approx 1.6$ increases the worst-case lock acquisition delay by $122/82 \approx 1.5$ for these two cases. The total time for all processors to have acquired the lock once, in turn, goes up by $5068/1798 \approx 2.8$.

The primary conclusion from the micro-benchmark results is that the current approach is not very suitable for barrier-style synchronization of large SHMAC grids. While the current results are likely to improve with a better router model, the LL/SC tile will still be the single point of synchronization, which will limit the scalability. One solution to this would be utilizing multiple, distributed LL/SC tiles and implementing a nested scheme for barrier synchronization.

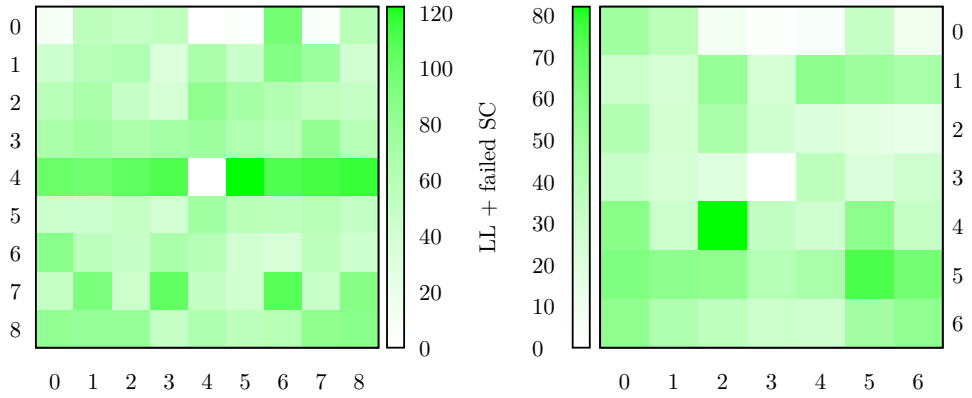


Figure 4.15.: Heat map plot showing total number of failures in acquiring an LL/SC lock relative to coordinate in 9×9 and 7×7 grids.

5. Improving the SHMAC

From the micro-benchmark results in Chapter 4, some conclusions regarding the current state of the SHMAC architecture have been drawn. In light of these results and their analysis, three proposals will be made in this chapter, and their impact measured by the same benchmarks where possible.

The first two of these proposals cover improvements to the SHMAC architecture. The dual-port RAM and router bypass implementation aims to provide lower latency access to on-tile memories, which increases instruction fetch performance. The system register file proposes integrating per-tile clock counting and bootstrapping capabilities to make these features independent of grid traffic. The final proposal, packet tracking, is not an architectural improvement but rather a new feature to give better instrumentation capabilities to the SHMAC.

5.1. Dual-Port RAM and Router Bypass

A common result from almost all the micro-benchmarks described in Chapter 4 is that the current model constitutes a large bottleneck for SHMAC's performance. This is perhaps most visible for the 28 CPI base instruction fetch performance observed in Section 4.4.1. Even though the current core model has single-cycle instruction execution and the memory takes around three cycles to respond to the request, the round-trip through the router introduces a 20+ cycle delay. This results in unacceptable performance levels for the entire system, since instruction fetches contribute significant delay to each operation. While a fast router model would be the most obvious improvement, this was unfortunately not possible in the assignment timeframe. Instead, a simpler solution that was also suggested by Rusten and Sortland [54] was implemented and evaluated: the *dual-port RAM with router bypass*.

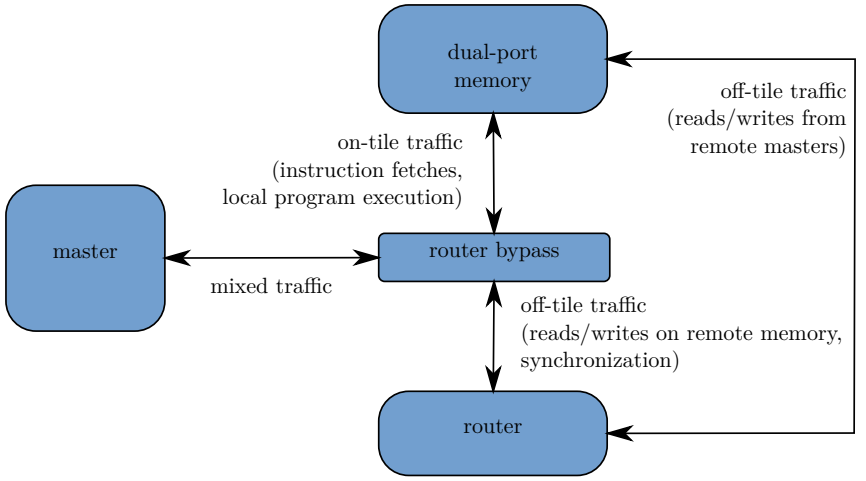


Figure 5.1.: Overview of on-tile memory system with dual-port RAM and router bypass unit integrated

5.1.1. Description

A dual-port RAM unit exposes two memory access ports, each capable of executing read or write operations independent of the other. The second memory port can be exploited to achieve higher on-tile memory access bandwidth for SHMAC's case. In the current implementation, this is done by introducing a *router bypass* unit between the master unit and the router, as illustrated in Figure 5.1.

The router bypass unit redirects the memory requests corresponding to on-tile addresses directly to one of the memory ports on the dual-port slave units. All other traffic passes through unmodified to the router, and the other memory port is connected to the router as usual. Assuming a low-latency router bypass unit, this allows a much shorter path between the on-tile slave and master units.

Two `SHMACMemoryInterface` ports on the dual-port slave and a zero-delay router bypass are utilized in the current implementation. While the zero-delay is unrealistic for hardware, a minimal implementation need only introduce a minimal delay on the critical path. Additionally, local memory accesses are still subject to network adapter delays in the current implementation. These delays can be removed by making a direct memory interface connection. Figure 5.2 exemplifies such a hardware implementation scenario with a direct interface between the master and the slave unit.

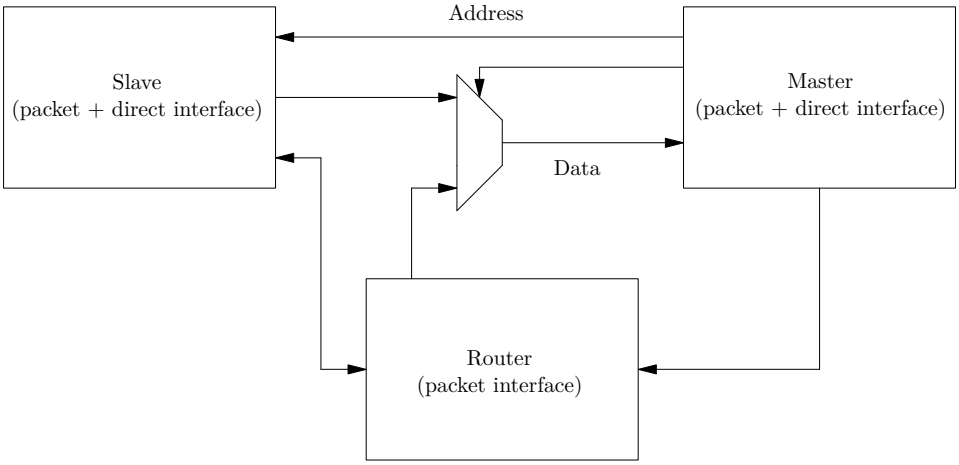


Figure 5.2.: A sample implementation for the router bypass unit using a direct memory interface and a multiplexer in conjunction with a dual-port slave tile. Reproduced from [54].

5.1.2. Evaluation and Results

As can be expected, the most evident effect of introducing dual-port slaves with router bypasses is on instruction fetch performance. Figure 5.3 shows the results from the Single Master Memory Access benchmark with the dual-port RAM and router bypass enhancements. Comparing with the results from Section 4.4.1, the instruction fetch performance has a significant 3.5× speedup, taking 8 CPI instead of 28. The Neighbour Read and Neighbour Write performance also increases to a lesser degree, with respective speedups of 1.36× and 2.71×.

The Multi-master Memory Access benchmark was also tested to measure the impact of improved instruction fetch latency. Although the latency to access the remote tile remains the same, each master is now able to send out requests faster. The results plotted in Figure 5.4 reveal some interesting changes compared to those in Section 4.4.2. The write performance goes down almost linearly with an increasing number of masters, although there is still a general improvement for four masters or less. The read performance is also improved but exhibits a similar behaviour to the single port slave case; up to three masters can read from the same tile without impacting each other's read performance. This suggests that reducing the instruction fetch latency allows a single master to saturate a memory unit's bandwidth with write operations, while multi-master read operations are still router-bound.

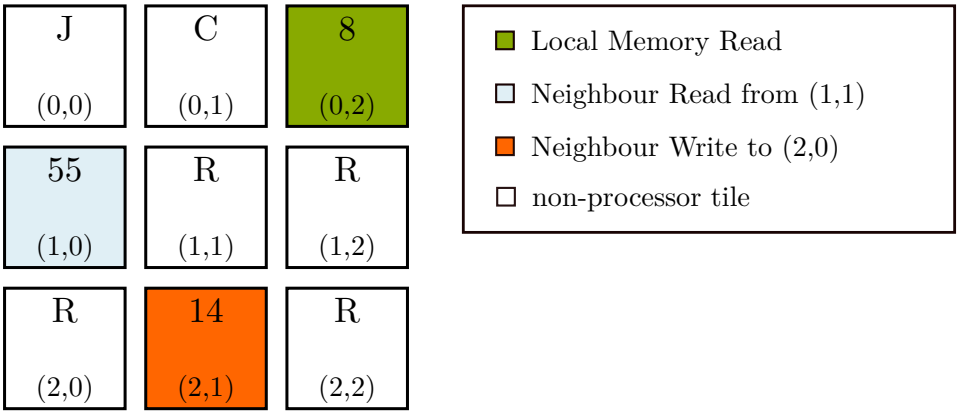


Figure 5.3.: The single master Pure Memory Read-Write benchmark, with all processor tiles utilizing dual-port slaves.

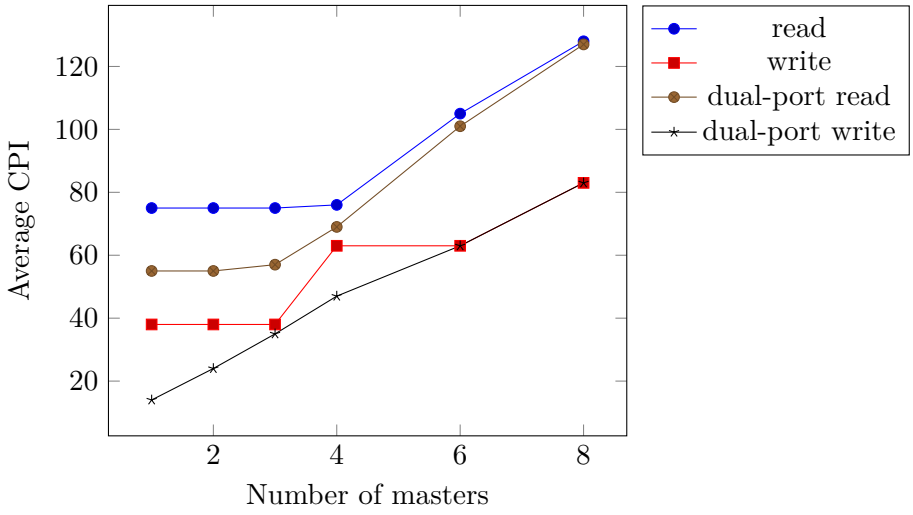


Figure 5.4.: Comparison of an increasing number of processors with single and dual-port slaves accessing the same RAM tile.

Separating the on-tile memory accesses from the remote requests also has important consequences for the RILF benchmark. Re-running the tests described in Section 4.5 after replacing all standard processor tiles dual-port slave versions, it was observed that remote accesses no longer hamper local instruction fetch performance.

To conclude, introducing a dual-port RAM slave unit with a router bypass primarily results in large instruction fetch performance improvements. This has the indirect consequence of greater bandwidth demand since each core can now quickly execute instructions and generate more memory requests per unit time. As memory unit bandwidth was found to be under-utilized in Section 4.4.1, the expected result is improved performance for the entire system, which is verified by the results here.

5.2. System Register File

In sections 4.2 and 4.3, the downsides of using the clock and jump tiles were discussed. To remedy the deterioration in timing measurements and start-up time, as well as to introduce useful new features, a *system register file* addition is proposed. This register file will provide on-tile bootstrapping and clock counting functionality, and can be enhanced with other per-tile information.

5.2.1. Description

The system register file is intended to be tightly coupled to the core, but in order to remain portable with future core models it was modelled separately and composed as part of the master unit. Figure 5.5 illustrates how an ArchC MIPS core, the TLM adapter and the system register file are combined into a master unit.

In this implementation, the system register file unit implements the standard SHMAC memory interface, and sits between the master unit (i.e. processor core) and the router. One or more regions of the global address space are reserved for the system registers. Any memory requests corresponding to addresses in the reserved regions are trapped by the system register file unit and responded to directly. All other traffic passes through untouched.

Table 5.1 describes a list of the registers implemented inside the current system register file implementation. This is only a small subset of possible

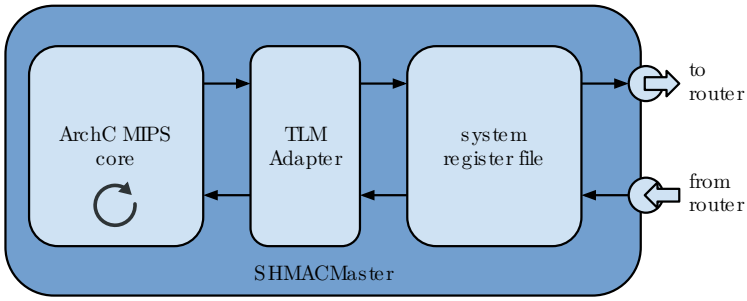


Figure 5.5.: The system register file shown in a SHMACMaster context.

Register	Address	Description
Bootstrap	`0x00000000` to `0x0000000C`	Core-specific instructions to jump to beginning of local tile memory
Clock counter	`0xFFFFFDFF`	Elapsed clock count since system start
Tile identifier	`0xFFFFFE03`	The local tile coordinate

Table 5.1.: A list of registers in the current system register file implementation.

system registers; any type of per-tile data, metadata or configuration could be stored here. For instance, configuration for a hardware packet tracking implementation could be placed in its own system register.

5.2.2. Evaluation and Results

The system register file was developed and evaluated in response to the jump tile delay and clock skew problems, and the Clock Tile Access Time benchmark (Section 4.2) already uses it to provide the baseline local instruction fetch values and demonstrate the skewing in clock tile readings. Coordinate-independent boot delay has been verified by re-running the Tile Startup Delays (Section 4.3) benchmark, and the processor ID register has been used across all benchmarks to customize program flow.

A summary of benefits from the system register file approach are listed below.

- The part of the memory map used by jump and clock tiles are freed
- Hardware resources used for the jump and clock tiles (e.g routers) are freed
- Time measurement skews described in 4.2 are no longer an issue
- Tiles boot-up in constant time, independent of grid coordinates and traffic
- The local tile coordinate can be exposed to the processor at runtime to customize program behaviour

As a downside, extra hardware resources will be required per processor tile; but taking the results and the big picture into account, it is the author's belief that a system register file implementation would be of significant benefit to the SHMAC.

It is worthy to note here that the current implementation is not a realistic one for hardware. There are no delays introduced by the system register file itself, neither for register accesses nor for messages passing through to the router and back to the core. A realistic implementation of the current scheme would introduce additional delays on the core-to-memory path for every memory access. This could be remedied by removing the system register from the critical path and accessing it in a different manner. Accessing both the router and the register file in parallel, connecting it to a dedicated router

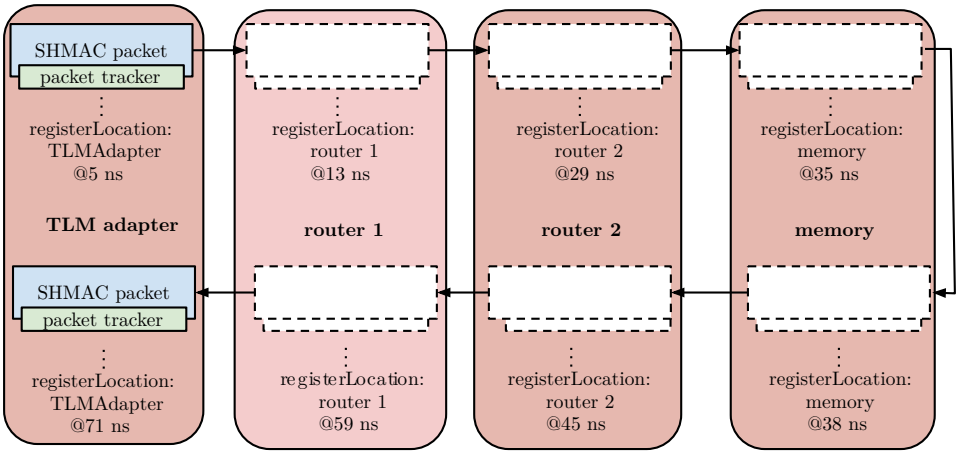


Figure 5.6.: Illustration of tracking a request packet and its reply across network hops.

port or integrating it into the processor core's register file are three of the possible options for better performance, though further evaluation was not possible due to time constraints.

5.3. Packet Tracking

5.3.1. Description

While analysing benchmark results on an architecture, it is useful to know the sources and lengths of delays that influence performance. Due to its packet and NoC-based memory system, packets are a good candidate for instrumentation in SHMAC, and a packet tracking system is proposed for the SHMAC. Inspired by the ***traceroute*** command used for Internet Protocol (IP) diagnostics, this system allows tracking the path of SHMAC network packets with timestamps.

Figure 5.6 illustrates how a packet could be tracked across hops in the SHMAC network. This system has been implemented with the following workflow in SHMACsim:

1. The characteristics [1] of packets desired to be tracked is configured.
2. A `SHMACPacketTracker` is attached to the packets matching the configured characteristics just before entering the network
3. If a packet has an attached tracker, each hop on the network annotates its name and the current simulation time by calling the ***registerLocation*** function on the tracker
4. The annotated hop names and simulation times are printed when the packet reaches its terminus and the `SHMACPacketTracker` object is destroyed

5.3.2. Results

While not an architectural improvement on its own, the packet tracking system can be a useful tool for debugging and instrumentation by identifying how much time the packets spend in different parts of the memory system. For instance, the analysis of packet tracking data in Section 4.4.1 helps identify the reason behind the lower instruction fetch performance in the Neighbour Write micro-benchmark. An example packet trace showing the lifetime of a local instruction fetch request and response is shown in Figure 5.7.

The current implementation was made on a rather high abstraction level due to time constraints, though it should be possible to implement such a structure in hardware. A brief literature search on the subject yielded no previous implementations, although Quality of Service capabilities for NoCs may be of some relevance in terms of a packet tracking implementation.

[1] The current way of selecting packets to be tracked is done directly inside SHMACsim source code. While this approach is not practical, it is very flexible since C++ code can be used to specify the packet characteristics.

```
Tracking entries for SHMAC packet:
    req = 1 addr = 11000000, data = 0,
    r/w = 0, flags = 0, sender = (1, 1)

****************************************
@TLMAdapter    1 ns(20901 ns...20902 ns)
@MemToRTL      (at 20902 ns)
@routerIn      5 ns(20902 ns...20907 ns)
@routerOut     5 ns(20907 ns...20912 ns)
@RTLToMem      1 ns(20912 ns...20913 ns)
@memory        3 ns(20913 ns...20916 ns)
@MemToRTL      (at 20916 ns)
@routerIn      5 ns(20916 ns...20921 ns)
@routerOut     5 ns(20921 ns...20926 ns)
@RTLToMem      (at 20926 ns)
@TLMAdapter    2 ns(20926 ns...20928 ns)
****************************************

End-to-end: 27 ns (20901 ns...20928 ns)
```

Figure 5.7.: A packet trace obtained from tracking a local instruction fetch request and response on tile (1,1).

6. Conclusion and Future Work

In this thesis, the topics of the construction of a cycle-accurate simulation framework for the SHMAC prototype, how it was used to evaluate the current state of the architecture via micro-benchmarks, and how the SHMAC can be improved have been covered. The following sections will relate the obtained results to the initial goals for the assignment, and provide a list of further improvements on both the simulation infrastructure and the SHMAC architecture via future work.

6.1. Conclusion

Three research questions were presented in Section 1.4 which this assignment aims to answer. The results from the report are related to each research question as presented below:

RQ1. How should the SHMAC architecture be modelled in a software simulator?

A mixed-level modelling approach using SystemC was deemed appropriate in SHMACsim, as explained in Section 3.1.2. The simulation infrastructure uses the object-oriented paradigm to provide a modular, extensible structure with clean interfaces and includes a configuration system to instantiate and configure desired heterogeneous tile grids. Most of the tiles provided in the SHMAC prototype have been implemented using this infrastructure, as indicated by a comparison of Tables 2.2 and 3.3. The simulator-constructed SHMAC grids have been verified to reflect the performance of the prototype with less than 5 % relative error.

RQ2. What sort of benchmark programs can be utilized to identify major bottlenecks in the SHMAC architecture?

As the SHMAC is at a relatively early architectural stage, micro-benchmarks instead of large benchmark suites were applied to evaluate the current state.

These micro-benchmarks mostly target the memory system, a vital component for high performance. The results obtained suggest that the NoC routers, which are depended on for every single memory access, cause the single largest performance hit in the system. It was demonstrated that the router latencies cause an under-utilization of bandwidth from memory units, which can be remedied by sharing a memory tile between several masters, though sustained access to on-tile scratchpad memories of other masters was shown to cause deteriorated performance. Additionally, the traffic-dependent behaviour of the clock tiles and its potential impact on benchmark measurements were demonstrated. Finally, system-wide synchronization of large SHMAC grids using a single LL/SC tile was shown to exhibit low scalability.

RQ3. How can the shortcomings of the SHMAC architecture and the current implementation be remedied?

In the light of the micro-benchmark results and analysis, the greatest shortcoming of the current SHMAC was identified as the slow router, which in turn causes low memory access performance. To improve on-tile memory access latency for processor tiles, a dual-port RAM slave connected to the processor core via a router bypass unit was proposed and implemented. The resulting 3.5x speedup in instruction fetch performance has positive effects for all types of workloads. Another proposed enhancement is upgrading processor tiles with a system register file. This provides integrated clock counting and bootstrapping functionality, independent of the mesh traffic. Finally, a packet tracking system for instrumentation of network packet lifetime was also proposed, allowing easier analysis of architectural bottlenecks.

6.2. Future Work

While the work on done in this assignment provides the foundations of a software-based evaluation tool for the SHMAC architecture, there is room for many improvements and useful new features. These are described in the following sections.

6.2.1. Power Modelling

SHMACsim currently does not provide any information on power consumption of the simulated models. Since power consumption and efficiency is the

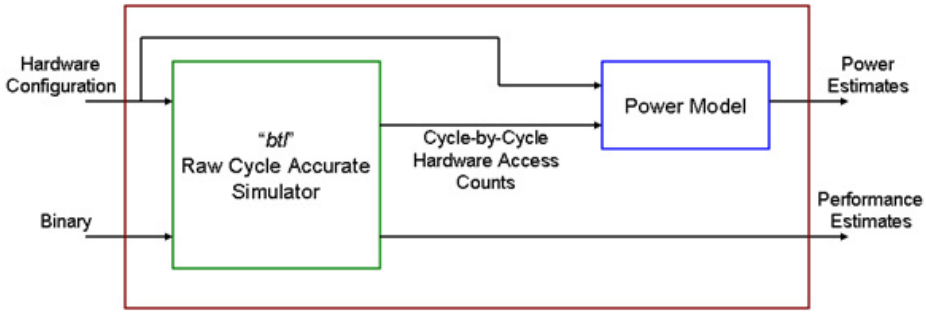


Figure 6.1.: Power modelling overview in TileWattch. Reproduced from [33].

primary reason for the shift to HMP systems, these can be greatly useful metrics for an HMP simulation infrastructure to provide.

The power model is usually provided by a framework, which maps fine- or coarse-grained operations to their respective power costs depending on technology parameters. Frameworks Wattch [14] and Cacti [51] are two such examples, and Konstantakopoulos et al. [33] mention a Wattch-based model for tiled processors called TileWattch. While it is possible to add a power model manually to relevant subsystems in SHMACsim, another approach that is potentially easier is to use a solution that integrates with SystemC. A body of existing work [12, 21, 31] covers the construction of SystemC-based power estimation frameworks.

Once such a framework has been constructed, additional schemes such as power and clock gating are relatively easy to implement in SystemC. Especially power gating is expected to be particularly useful to create a realistic "heterogeneous dark silicon" environment.

6.2.2. ELF Parsing

The slave units in SHMACsim only accept raw binary executable code as input in the current version, and the ***shmacsim-archc-compile*** script has the responsibility to convert from the ELF-executables generated by the toolchain into raw binaries as described in Section 3.4. However, the objcopy command called by the script only copies the code and data segments of the executable, which results in the stripping of all other segments, including debug information. ELF parsing can also be implemented as part of an

operating system running on the SHMAC, though this requires significant development effort.

Implementing a simulation infrastructure-level ELF parser, for instance with the help of libelf [53], would allow a more flexible approach without the extra objcopy step or segment stripping. This would enable debugging and the usage of dynamic libraries. Additionally, the flexible structure of the ELF format could be then utilized to add SHMAC-specific features such as per-tile segment relocation specifications or code segment characteristics for deciding on which tile is best suited to execute that segment of code.

6.2.3. GDB Support

Even with verified processor core models, software debugging remains a powerful tool to make deeper observations into a program's behaviour. While ArchC itself supports the popular GNU Debugger (GDB), the processor cores shipped with the current SHMACsim lack the necessary protocol wrapper that allows GDB to connect to the program running on the core.

The initial problem with enabling the protocol wrappers is assigning them to different TCP ports to allow individual GDB connections to the cores, which is not particularly difficult. In order for GDB to be able to access memory contents, a wrapper for the SHMAC memory system will also have to be added.

Finally, the debug segments are currently stripped from executables before loading them on the cores. To enable source code line matching, ELF parsing will have to be implemented, as discussed in Section 6.2.2.

6.2.4. Dynamic Model Switching

The initialization phase for the micro-benchmarks tested within the scope of this work is very small. Combined with the small micro-benchmark kernels, running to completion takes a reasonably short time, typically ending within minutes. However, if SHMACsim is to be used for evaluating larger applications with heavy initialization procedures or running on top of an operating system, the current simulation performance will be prohibitively low.

The current simulation performance is low due to the cycle-accurate interconnect model. While optimization can increase performance to some degree,

cycle-accurate models in software will always suffer from low performance due to cycle-by-cycle synchronization. A way to get around the performance problem for booting operating systems or performing heavy initialization is *dynamic model switching*. In this scheme, a fast non-CA model is utilized to quickly boot the operating system or perform the initialization phase. Afterwards, the system switches to an accurate model to perform the desired evaluation.

Both functional and cycle-accurate models already exist in SHMACsim, so it would be sufficient to add the necessary functionality to change the router models in a configurable manner. This could be done by including a simulator configuration register in the system register file.

6.2.5. Packet Tracking

Packet tracking can be a powerful tool for identifying bottlenecks and debugging, though more development is needed to unleash its full potential. Several proposals to make the packet tracking system more useful are listed below.

Configuring the Packet Tracker: The amount of traffic created on the SHMAC network by even the simplest of micro-benchmarks is very large. Because of this, it is undesirable to track every single packet, but rather a specific configurable subset that would yield the most interesting information. Currently, identifying the packets to be tracked is done via a C++ function call examining the packet and returning *true* or *false* accordingly. While this is a powerful and flexible method, it is desirable to be able to change the tracking configuration without recompilation. To provide this, a simpler method involving setting desired packet characteristics in the runtime configuration file could also be implemented.

Aggregating Statistics: The current packet tracking implementation prints every single trace to the standard error stream. If a large amount of packets are to be tracked, it is desirable to see a statistical summary instead of sifting through large volumes of data. Towards this end, a module aggregating packet traces into statistics is can be implemented. Average lifetime of a packet type for each hop and histograms are examples of potentially useful packet trace statistics.

Automatic Location Registration: In Section 5.3 it was described how location information is annotated onto the packet as it moves through

the network. Currently, this is done by manually adding function calls to *registerLocation* at each step to be monitored. Since all packet traffic passes through SystemC ports, a better solution would be creating a subclass of `sc_port` which automatically adds the tracking information as the packet goes through the port.

6.2.6. Improvements to the SHMAC Architecture

Considering that the goal of the SHMAC is to provide an infrastructure for heterogeneous software as well as evaluating heterogeneous hardware, the continuous evolution of the architecture is actually part of the goal itself. Beyond the improvements discussed in Chapter 5, many potential architectural enhancements and new features remain unexplored. In light of the body of knowledge acquired while preparing this thesis, a brief discussion of some of these is provided below.

6.2.6.1. A Faster, Pipelined Router Implementation

A number of proposals were covered by Rusten and Sortland [54] in their Future Work section, including router improvements. While these will not be repeated here, the results from this thesis strongly support the need for a faster router, which remains the greatest bottleneck for the current SHMAC. NoC router implementations are well-studied in existing literature, and an implementation based on those surveyed in [10] could provide a useful starting point.

6.2.6.2. Integrating a System-Wide Bus

Despite the difficulties discussed in Section 2.2.2 about using a global shared bus in a multi-core environment, there may still be use-cases where a global bus is useful. Namely, while a bus has worse throughput and scalability characteristics, it does not suffer from the multi-hop latencies introduced by a NoC. By introducing a global bus into the system, low-latency multicast and broadcast operations are made possible. To get most benefits from such a bus, it becomes important to keep the traffic volume low. Multi-core synchronization messages, which are small in size and sensitive to latency, become likely candidates to be carried over the bus. Manevich et al. [40]

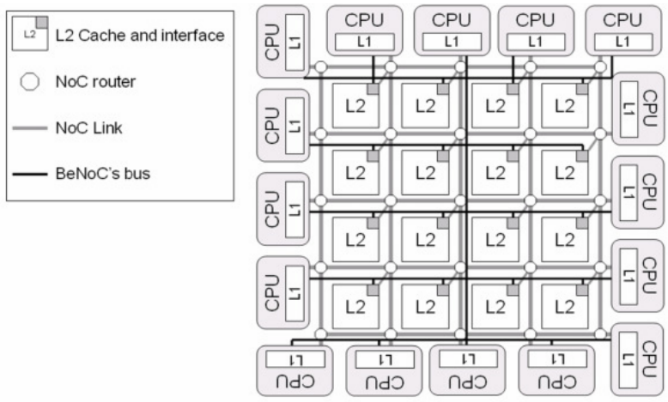


Figure 6.2.: Overview of a BENoC-based CMP system with 16 processors and 4x4 L2 caches. Reproduced from [40].

discuss such a Bus Enhanced NoC (BENoC) scheme and report promising results for carrying cache coherence messages using the bus.

Instead of being coupled only a specific piece of hardware such as the cache, the bus can also be exposed to software via memory-mapped access to allow a wider range of architectural experiments. Support for this could be implemented in the SHMAC by expanding the routers to add one more input/output port connected to the global bus.

6.2.6.3. Core Diversity and Accelerators

As the SHMAC architecture matures, it will be interesting to experiment with heterogeneous layouts containing different cores and diverse accelerators. The topics of core and accelerator diversity, and how accelerators should be integrated into an HMP system remain open research questions. A brief discussion on these is provided below.

Core Types: Different cores can be created by customizing various micro-architectural features, including but not limited to pipeline stages, instruction reordering, additional datapaths and cache. A framework which allows easy customization of these micro-architectural features would be very useful for evaluating different cores within the SHMAC, similar to how ArchC is used in SHMACsim. Even though existing work [36, 63] suggests most heterogeneity benefits are achieved by as little as two core types, these two

core types will still need to be identified, and such a framework will help in the process.

Accelerator Integration: The accelerators may be exposed to the system directly as individual tiles, but the single-ISA nature of the system will require a mapping from the ISA to the accelerator functionality. Towards this, the work from Clark et al. [17] describing a way of expressing the computation to be accelerated using a baseline ISA can be useful. Another approach is to control the accelerators by a tightly-coupled general-purpose core on the same tile.

Accelerator Types: Accelerator access and control aside, the types of accelerators to be included is another important HMP aspect. Several were mentioned in Section 2.2.1, but the possibilities are many more. Single Instruction Multiple Data (SIMD) and digital signal processor (DSP)-style accelerators which are common in industry should be evaluated in a SHMAC context to identify their usefulness in a variety of target scenarios. More unconventional heterogeneity by adding dataflow execution or artificial neural network accelerators may also be worth exploring.

Bibliography

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

[2] Niket Agarwal et al. "GARNET: A detailed on-chip network model inside a full-system simulator". In: *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE. 2009, pp. 33–42.

[3] Gene M Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM. 1967, pp. 483–485.

[4] Rodolfo Azevedo and Sandro Rigo. "ArchC Model Design Handbook". In: *Electronic System Level Design*. Springer, 2011, pp. 39–70.

[5] James Balfour and William J Dally. "Design tradeoffs for tiled CMP on-chip networks". In: *Proceedings of the 20th annual international conference on Supercomputing*. ACM. 2006, pp. 187–198.

[6] Luca Benini and Giovanni De Micheli. "Networks on chips: A new SoC paradigm". In: *Computer* 35.1 (2002), pp. 70–78.

[7] V Bhatt et al. "Sichrome: Mobile web browsing in hardware to save energy". In: *Dark Silicon Workshop, ISCA*. 2012.

[8] Nathan Binkert et al. "The gem5 simulator". In: *SIGARCH Comput. Archit. News* 39.2 (Aug. 2011), pp. 1–7. ISSN: 0163-5964. DOI: 10.1145/2024716.2024718. URL: http://doi.acm.org/10.1145/2024716.2024718.

[9] Nathan L Binkert et al. "The M5 simulator: Modeling networked systems". In: *Micro, IEEE* 26.4 (2006), pp. 52–60.

[10] Tobias Bjerregaard and Shankar Mahadevan. "A survey of research and practices of network-on-chip". In: *ACM Computing Surveys (CSUR)* 38.1 (2006), p. 1.

[11] David C Black and Jack Donovan. *SystemC: From the ground up*. Vol. 71. Springer, 2010.

[12] A. Bona, V. Zaccaria, and R. Zafalon. "System level power modeling and simulation of high-end industrial network-on-chip". In: *Design, Automation and Test in Europe Conference and Exhibition, 2004.*

Proceedings. Vol. 3. 2004, 318–323 Vol.3. DOI: 10.1109/DATE.2004.1269258.

[13] Shekhar Borkar and Andrew A Chien. "The future of microprocessors". In: *Communications of the ACM* 54.5 (2011), pp. 67–77.

[14] David Brooks, Vivek Tiwari, and Margaret Martonosi. "Wattch: a framework for architectural-level power analysis and optimizations". In: *Proceedings of the 27th annual international symposium on Computer architecture*. ISCA '00. Vancouver, British Columbia, Canada: ACM, 2000, pp. 83–94. ISBN: 1-58113-232-8. DOI: 10.1145/339647.339657. URL: http://doi.acm.org/10.1145/339647.339657.

[15] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. "Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation". In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '11. Seattle, Washington: ACM, 2011, 52:1–52:12. ISBN: 978-1-4503-0771-0. DOI: 10.1145/2063384.2063454. URL: http://doi.acm.org/10.1145/2063384.2063454.

[16] Eric S Chung et al. "Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus?" In: *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society. 2010, pp. 225–236.

[17] N. Clark, A. Hormati, and S. Mahlke. "VEAL: Virtualized Execution Accelerator for Loops". In: *Computer Architecture, 2008. ISCA '08. 35th International Symposium on*. 2008, pp. 389–400. DOI: 10.1109/ISCA.2008.33.

[18] R. C. Covington et al. "The rice parallel processing testbed". In: *SIGMETRICS Perform. Eval. Rev.* 16.1 (May 1988), pp. 4–11. ISSN: 0163-5999. DOI: 10.1145/1007771.55596. URL: http://doi.acm.org/10.1145/1007771.55596.

[19] William J Dally and Brian Towles. "Route packets, not wires: On-chip interconnection networks". In: *Design Automation Conference, 2001. Proceedings*. IEEE. 2001, pp. 684–689.

[20] Robert H Dennard et al. "Design of ion-implanted MOSFET's with very small physical dimensions". In: *Solid-State Circuits, IEEE Journal of* 9.5 (1974), pp. 256–268.

[21] Nagu Dhanwada, Ing-Chao Lin, and Vijay Narayanan. "A power estimation methodology for systemC transaction level models". In: *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. CODES+ISSS '05. Jersey City, NJ, USA: ACM, 2005, pp. 142–147. ISBN: 1-59593-161-9.

DOI: 10.1145/1084834.1084874. URL: http://doi.acm.org/10.1145/1084834.1084874.

[22] Hadi Esmaeilzadeh et al. “Dark silicon and the end of multicore scaling”. In: *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*. IEEE. 2011, pp. 365–376.

[23] Alexandra Fedorova et al. “Maximizing power efficiency with asymmetric multicore systems”. In: *Communications of the ACM* 52.12 (2009), pp. 48–57.

[24] D Frommer. “Smartphone sales to beat PC sales by 2011”. In: *Business Insider* (2009).

[25] Nathan Goulding-Hotta et al. “The GreenDroid mobile application processor: An architecture for silicon’s dark future”. In: *Micro, IEEE* 31.2 (2011), pp. 86–95.

[26] Peter Greenhalgh. *big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7*. 2011. URL: "http://www.arm.com/files/downloads/big.LITTLE_Final.pdf".

[27] John L Gustafson. “Reevaluating Amdahl’s law”. In: *Communications of the ACM* 31.5 (1988), pp. 532–533.

[28] Scott Hauck and Andre DeHon. *Reconfigurable computing: the theory and practice of FPGA-based computation*. Morgan Kaufmann, 2010.

[29] “IEEE Standard for Standard SystemC Language Reference Manual”. In: *IEEE Standard 1666-2011* (2011).

[30] Texas Instruments. *OMAP Applications Processors*. 2013. URL: "http://www.ti.com/product/OMAP4470".

[31] Felipe Klein et al. *PowerSC: A SystemC-based framework for power estimation*. Tech. rep. Technical Report IC-07-02, Institute of Computing, UNICAMP, 2007.

[32] Onur Koçberber et al. “Dark Silicon Accelerators for Database Indexing”. In: *Dark Silicon Workshop, ISCA*. 2012.

[33] T Konstantakopoulos et al. *Power Performance of Tiled Processor Architectures*. URL: "http://publications.csail.mit.edu/abstracts/abstracts05/tkonsta/tkonsta.html".

[34] Jonathan G Koomey. “Worldwide electricity used in data centers”. In: *Environmental Research Letters* 3.3 (2008), p. 034008.

[35] Kelin J Kuhn. “Moore’s Law Past 32nm: Future Challenges in Device Scaling”. In: *Computational Electronics, 2009. IWCE’09. 13th International Workshop on*. IEEE. 2009, pp. 1–6.

[36] Rakesh Kumar et al. “Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction”. In: *Microarchitec-*

ture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on. IEEE. 2003, pp. 81–92.

[37] Rasmus Christian Larsen and Oyvind Janbu. *Introduction to EFM32 Microcontrollers*. URL: http://cdn.energymicro.com/dl/pdf/efm32_introduction_white_paper.pdf.

[38] Igor Loi and Luca Benini. "An efficient distributed memory interface for many-core platform with 3D stacked DRAM". In: *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association. 2010, pp. 99–104.

[39] Peter S Magnusson et al. "Simics: A full system simulation platform". In: *Computer* 35.2 (2002), pp. 50–58.

[40] R. Manevich et al. "Best of both worlds: A bus enhanced NoC (BENoC)". In: *Networks-on-Chip, 2009. NoCS 2009. 3rd ACM/IEEE International Symposium on*. 2009, pp. 173–182. DOI: 10.1109/NOCS.2009.5071465.

[41] Milo MK Martin et al. "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset". In: *ACM SIGARCH Computer Architecture News* 33.4 (2005), pp. 92–99.

[42] Jiayuan Meng and Kevin Skadron. "A reconfigurable simulator for large-scale heterogeneous multicore architectures". In: *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*. IEEE. 2011, pp. 119–120.

[43] Jason E Miller et al. "Graphite: A distributed parallel simulator for multicores". In: *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*. IEEE. 2010, pp. 1–12.

[44] Bill Nitzberg and Virginia Lo. "Distributed shared memory: A survey of issues and algorithms". In: *Computer* 24.8 (1991), pp. 52–60.

[45] nvidia. *Variable SMP - A multi-core CPU architecture for low power and high performance*. 2011. URL: http://www.nvidia.com/content/PDF/tegra_white_papers/tegra-whitepaper-0911b.pdf.

[46] Pablo Montesinos Ortego and Paul Sack. "SESC: SuperESCalar simulator". In: *17 th Euro micro conference on real time systems (ECRTS'05)*. 2004, pp. 1–4.

[47] Mario Pickavet et al. "Worldwide energy needs for ICT: The rise of power-aware networking". In: *Advanced Networks and Telecommunication Systems, 2008. ANTS'08. 2nd International Symposium on*. IEEE. 2008, pp. 1–3.

[48] Fred J Pollack. "New microarchitecture challenges in the coming generations of CMOS process technologies (keynote address)". In: *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society. 1999, p. 2.

[49] Jeff Preshing. *A Look Back at Single-Threaded CPU Performance*. Feb. 2012. URL: "http://preshing.com/20120208/a-look-back-at-single-threaded-cpu-performance".

[50] Qualcomm. *Qualcomm Snapdragon Processors*. 2013. URL: "http://www.qualcomm.com/snapdragon".

[51] Glen Reinman and Norman P Jouppi. "CACTI 2.0: An integrated cache timing and power model". In: *Western Research Lab Research Report* 7 (2000).

[52] Steve Rhoads. *Plasma - most MIPS-I opcodes*. Apr. 2012. URL: "http://opencores.org/project,plasma".

[53] Michael Riepe. *libelf*. URL: "http://directory.fsf.org/wiki/Libelf".

[54] Leif Tore Rusten and Gunnar Inge Sortland. "Implementing a Heterogeneous Multi-Core Prototype in an FPGA". MA thesis. Norwegian University of Science and Technology, 2012.

[55] Samsung. *Samsung Exynos Processors*. 2013. URL: "http://www.samsung.com/global/business/semiconductor/minisite/Exynos/index.html".

[56] Robert R Schaller. "Moore's law: past, present and future". In: *Spectrum, IEEE* 34.6 (1997), pp. 52–59.

[57] David E Shaw et al. "Anton, a special-purpose machine for molecular dynamics simulation". In: *ACM SIGARCH Computer Architecture News*. Vol. 35. 2. ACM. 2007, pp. 1–12.

[58] Kevin Skadron et al. "Challenges in computer architecture evaluation". In: *Computer* 36.8 (2003), pp. 30–36.

[59] Herb Sutter. "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software". In: *Dr. Dobb's Journal* 30.3 (2005). URL: "http://www.gotw.ca/publications/concurrency-ddj.htm".

[60] Zhangxi Tan et al. "A case for FAME: FPGA architecture model execution". In: *ACM SIGARCH Computer Architecture News*. Vol. 38. 3. ACM. 2010, pp. 290–301.

[61] Michael Bedford Taylor et al. "The Raw microprocessor: A computational fabric for software circuits and general-purpose programs". In: *Micro, IEEE* 22.2 (2002), pp. 25–35.

[62] *The Move to Multi-Core Architecture Explained*. Intel Corporation. URL: http://software.intel.com/en-us/articles/frequently-asked-questions-intel-multi-core-processor-architecture#_The_move_to_dual/multi-coreexplain.

[63] Kenzo Van Craeynest and Lieven Eeckhout. "Understanding fundamental design choices in single-ISA heterogeneous multicore architectures".

In: *ACM Transactions on Architecture and Code Optimization (TACO)* 9.4 (2013), p. 32.

[64] Ganesh Venkatesh et al. "Conservation cores: reducing the energy of mature computations". In: *ACM SIGARCH Computer Architecture News*. Vol. 38. 1. ACM. 2010, pp. 205–218.

[65] David W Wall. *Limits of instruction-level parallelism*. Vol. 19. 2. ACM, 1991.

[66] Hao Wang et al. "Workload and power budget partitioning for single-chip heterogeneous processors". In: *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. PACT '12. Minneapolis, Minnesota, USA: ACM, 2012, pp. 401–410. ISBN: 978-1-4503-1182-3. DOI: 10.1145/2370816.2370873. URL: http://doi.acm.org/10.1145/2370816.2370873.

[67] Thomas Canhao Xu et al. "Hardware/Software Co-design for Multicore Architectures". PhD thesis. University of Turku, 2012.

A. Appendices

A.1. SHMACsim Utility Scripts

A.1.1. shmacsim-archc-compile

```
#!/bin/sh

if [ "$#" -lt "3" ]; then
  echo "mips-elf-gcc wrapper script for SHMACsim"
  echo "Usage: shmacsim-archc-compile tile_i_coord
      tile_j_coord gcc_args"
  echo "Outputs will be named (tile_i, tile_j) in .x (ELF
      executable) and .bin (raw binary) formats"
  exit 0
fi

SHMACSIM_TOOLS_DIR=$SHMACSIM_ROOT/tools
ARCHC_LDSCRIPT_GEN_NAME=create_archc_linkerscript.sh
SHMACSIM_SPEC_NAME=shmacsim_specs

MEM_LENGTH=0x1000000
MEM_ORIGIN=0x"$1$2"000000
CMDLINE_ARGS_RESERVE=1024
PROG_NAME="($1","$2)"

# remove any old linker scripts, suppress errors
rm $SHMACSIM_TOOLS_DIR/temp.ld 2> /dev/null

echo "Compiling for tile:\t($1,$2)"
echo "Tile RAM start:\t\t$MEM_ORIGIN"
echo "Tile RAM size:\t\t$MEM_LENGTH"
echo "Cmdline args resv:\t$CMDLINE_ARGS_RESERVE"

# create ArchC linker script for given configuration
# this linker script will place sections into the local tile
    RAM
$SHMACSIM_TOOLS_DIR/$ARCHC_LDSCRIPT_GEN_NAME $MEM_ORIGIN
    $MEM_LENGTH $CMDLINE_ARGS_RESERVE > $SHMACSIM_TOOLS_DIR/
    temp.ld
```

```
# shift out the first 3 arguments, we'll past the rest to
    gcc
shift 2

# call GCC with the rest of the arguments
mips-elf-gcc -specs=$SHMACSIM_TOOLS_DIR/$SHMACSIM_SPEC_NAME
    -T$SHMACSIM_TOOLS_DIR/temp.ld $@ -o $PROG_NAME.x

# call objcopy to generate binary output
# keep the .bss section in generated output
mips-elf-objcopy -K .bss --set-section-flags .bss=alloc,load
    ,contents -I elf32-bigmips -O binary $PROG_NAME.x
    $PROG_NAME.bin

# remove any old linker scripts, suppress errors
rm $SHMACSIM_TOOLS_DIR/temp.ld 2> /dev/null
```

A.1.2. shmacsim-archc-allcompile

```
#!/bin/sh

if [ "$#" -lt "3" ]; then
  echo "Compiles a given source file for all tiles in
      SHMACsim"
  echo "Usage: shmacsim-archc-allcompile i_dim j_dim
      gcc_args"
  exit 0
fi

SHMACSIM_TOOLS_DIR=$SHMACSIM_ROOT/tools
SHMACSIM_COMPILE_SCRIPT_NAME=shmacsim-archc-compile.sh
SHMACSIM_COMPILE_SCRIPT=$SHMACSIM_TOOLS_DIR/
    $SHMACSIM_COMPILE_SCRIPT_NAME

# since tile indices start from 0, subtract 1 from max vals
I_DIM=`expr $1 - 1`
J_DIM=`expr $2 - 1`

# shift out first two parameters, rest will be passed to gcc
shift 2

# call SHMACsim compile script for each tile
for i in `seq 0 $I_DIM`
do
  for j in `seq 0 $J_DIM`
  do
      $SHMACSIM_COMPILE_SCRIPT $i $j $@
  done
```

```
done

echo "\n\nshmacsim-allcompile finished. Runtime config file
    suggestion:\n"

# print out runtime config file mapping tiles to produced
    executables
for i in `seq 0 $I_DIM`
do
  for j in `seq 0 $J_DIM`
  do
      echo "($i,$j)\tsLoadBinary\t$PWD/($i,$j).bin"
  done
done
```

A.1.3. shmacsim-run-benchmarks

```
#!/bin/sh

# Location of SHMACsim executable
SHMACSIM_PATH="$SHMACSIM_ROOT/shmacsim/Debug/shmacsim"

# Benchmark folder configration, each benchmark
# $BENCHMARK is assumed to be structured as:
# hardware config file at $BENCHMARKS_ROOT/$BENCHMARK/
    $HARDWARECFG_FILENAME
# runtime config file at $BENCHMARKS_ROOT/$BENCHMARK/
    $RUNTIMECFG_FILENAME
BENCHMARKS_ROOT="$SHMACSIM_ROOT/microbenchmarks"
HARDWARECFG_FILENAME="hardwarecfg.txt"
RUNTIMECFG_FILENAME="runtimecfg.txt"

# The stdout and stderr from each benchmark will be
# placed in the BENCHMARK_LOGDIIR, whose name is
# generated using the current date-time
DATE=$(date +"%Y%m%d%H%M%S")
BENCHMARK_LOGDIR="$SHMACSIM_ROOT/benchmark_logs/$DATE"

BENCHMARKS=$@


echo "The following SHMACsim benchmarks will be run: "
echo $BENCHMARKS
mkdir -p $BENCHMARK_LOGDIR
echo "Benchmark results will be placed in $BENCHMARK_LOGDIR"

export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$SYSTEMC/lib-linux64
    :$SYSTEMC/lib-linux
```

```
PROCESS_LIST="$$"

for benchmark in $BENCHMARKS;
do
  echo "Now launching benchmark: $benchmark"
  BENCHMARK_SUBDIR=$(dirname $BENCHMARK_LOGDIR/$benchmark)
  mkdir -p $BENCHMARK_SUBDIR
  $SHMACSIM_PATH -h $BENCHMARKS_ROOT/$benchmark/
      $HARDWARECFG_FILENAME -r $BENCHMARKS_ROOT/$benchmark/
      $RUNTIMECFG_FILENAME 1>"$BENCHMARK_LOGDIR/$benchmark"
      _out.log 2>"$BENCHMARK_LOGDIR/$benchmark"_err.log &
  PROCESS_LIST="$PROCESS_LIST,$!"
done

echo "Use the following command to monitor benchmark process
     status:"
echo "top -p $PROCESS_LIST"
```

A.2. Micro-benchmark Listings

A.2.1. Clock Tile Access Time

```
#include <stdio.h>
#include "../shmacsim_microbenchmark.h"

#define REPEAT_COUNT 2000
#define REPEAT_COUNT_ASM ".rept 2000"

volatile unsigned int * clockTile = (volatile unsigned int
    *) 0x01000000;
#define SHMAC_CLKCNT_TILE    (*clockTile)

ProcessorTask __processorTasks[SHMAC_MAX_GRIDDIM*
    SHMAC_MAX_GRIDDIM];
unsigned int __processorID;

// task function definitions
void localInstrFetch();
void readFrom11();
void writeTo20();

int main(int argc, char *argv[])
{
    // initialization
    __processorID = SHMAC_PROCID;
    SHMAC_assignTasksToProcessors();
```

```
    // retrieve task for this processor
    ProcessorTask myTask = __processorTasks[__processorID];

    if(!myTask)
        printf("Processor %d was not assigned any tasks,
            exiting...\n", __processorID);
    else
        myTask();

  return 0;
}

void localInstrFetch()
{
    // variables for time measurement
    unsigned int clkStartTile = 0, clkEndTile = 0;
    unsigned int clkStart = 0, clkEnd = 0;
    printf("Processor %d: Local memory read, %d repeats...\n
        ", __processorID, REPEAT_COUNT);

    clkStartTile = SHMAC_CLKCNT_TILE;
    clkStart = SHMAC_CLKCNT;
    // ----- start of microbenchmark code -----
    __asm__("li $4,0");
    __asm__(REPEAT_COUNT_ASM);
    __asm__("addiu $4,$4,1");
    __asm__(".endr");
    // ----- end of microbenchmark code -----
    clkEnd = SHMAC_CLKCNT;
    clkEndTile = SHMAC_CLKCNT_TILE;


    // print statistics
    printf("Processor %d: Elapsed clock cycles: %d\n",
        __processorID, clkEnd-clkStart);
  printf("Processor %d: CPI: %d \n", __processorID, (clkEnd-
      clkStart) / REPEAT_COUNT);

  printf("Processor %d: Elapsed clock cycles (CT): %d\n",
      __processorID, clkEndTile-clkStartTile);
  printf("Processor %d: CPI (CT): %d \n", __processorID, (
      clkEndTile-clkStartTile) / REPEAT_COUNT);
}

void readFrom11()
{
    printf("Processor %d: Reading from (1,1) %d repeats...\n
        ", __processorID, REPEAT_COUNT);
    // ----- start of microbenchmark kernel -----
```

```
    __asm__("li $4,0x11000000");
    __asm__(REPEAT_COUNT_ASM);
    __asm__("lw $5,0($4)");
    __asm__(".endr");
    // ----- end of microbenchmark kernel -----
}

void SHMAC_assignTasksToProcessors()
{
    int i;
    // set all processor tasks to localInstrFetch
    for(i = 0; i < 16*16; i++)
    {
        if(i % 16 != 1)
            __processorTasks[i] = &readFrom11;
        else
            __processorTasks[i] = &localInstrFetch;
    }
}
```

A.2.2. Tile Start-Up Delays

```
#include <stdio.h>
#include "../shmacsim_microbenchmark.h"

#define REPEAT_COUNT 2000
#define REPEAT_COUNT_ASM ".rept 2000"

// override clock count retrieval def using clock tile
// remove these two lines to use the system register instead
//volatile unsigned int * clockTile = (volatile unsigned int
    *) 0x02000000;
//#define SHMAC_CLKCNT    (*clockTile)

ProcessorTask __processorTasks[SHMAC_MAX_GRIDDIM*
   SHMAC_MAX_GRIDDIM];
unsigned int __processorID;

// task function definitions
void printStartTime();

int main(int argc, char *argv[])
{
    // initialization
    __processorID = SHMAC_PROCID;
    SHMAC_assignTasksToProcessors();
    // retrieve task for this processor
    ProcessorTask myTask = __processorTasks[__processorID];
```

```
    if(!myTask)
        printf("Processor %d was not assigned any tasks,
            exiting...\n", __processorID);
    else
        myTask();

  return 0;
}

void printStartTime()
{
    // variables for time measurement
    unsigned int clkStart = 0;
    clkStart = SHMAC_CLKCNT;

    // print statistics
    printf("Processor %d: Start at: %d\n", __processorID,
        clkStart);
}


void SHMAC_assignTasksToProcessors()
{
    int i;
    // set all processor tasks to printStartTime
    for(i = 0; i < 16*16; i++)
        __processorTasks[i] = &printStartTime;
}
```

A.2.3. Pure Memory R/W Performance - Single Master

```
#include <stdio.h>
#include "../shmacsim_microbenchmark.h"

#define REPEAT_COUNT 2000
#define REPEAT_COUNT_ASM ".rept 2000"

// override clock count retrieval def using clock tile
// remove these two lines to use the system register instead
volatile unsigned int * clockTile = (volatile unsigned int
    *) 0x01000000;
#define SHMAC_CLKCNT    (*clockTile)

ProcessorTask __processorTasks[SHMAC_MAX_GRIDDIM*
    SHMAC_MAX_GRIDDIM];
unsigned int __processorID;
```

```
// task function definitions
void localInstrFetch();
void readFrom11();
void writeTo20();

int main(int argc, char *argv[])
{
    // initialization
    __processorID = SHMAC_PROCID;
    SHMAC_assignTasksToProcessors();
    // retrieve task for this processor
    ProcessorTask myTask = __processorTasks[__processorID];

    if(!myTask)
        printf("Processor %d was not assigned any tasks,
            exiting...\n", __processorID);
    else
        myTask();

  return 0;
}

void localInstrFetch()
{
    // variables for time measurement
    unsigned int clkStart = 0, clkEnd = 0;
    printf("Processor %d: Local memory read, %d repeats...\n
        ", __processorID, REPEAT_COUNT);
    clkStart = SHMAC_CLKCNT;
    // ----- start of microbenchmark code -----
    __asm__("li $4,0");
    __asm__(REPEAT_COUNT_ASM);
    __asm__("addiu $4,$4,1");
    __asm__(".endr");
    // ----- end of microbenchmark code -----
    clkEnd = SHMAC_CLKCNT;

    // print statistics
    printf("Processor %d: Elapsed clock cycles: %d\n",
        __processorID, clkEnd-clkStart);
  printf("Processor %d: CPI: %d \n", __processorID, (clkEnd-
      clkStart) / REPEAT_COUNT);
}

void readFrom11()
{
    // variables for time measurement
    unsigned int clkStart = 0, clkEnd = 0;
```

```
    printf("Processor %d: Reading from (1,1) %d repeats...\n
        ", __processorID, REPEAT_COUNT);
    clkStart = SHMAC_CLKCNT;
    // ----- start of microbenchmark kernel -----
    __asm__("li $4,0x11000000");
    __asm__(REPEAT_COUNT_ASM);
    __asm__("lw $5,0($4)");
    __asm__(".endr");
    // ----- end of microbenchmark kernel -----
    clkEnd = SHMAC_CLKCNT;

    // print statistics
    printf("Processor %d: Elapsed clock cycles: %d\n",
        __processorID, clkEnd-clkStart);
  printf("Processor %d: CPI: %d \n", __processorID, (clkEnd-
      clkStart) / REPEAT_COUNT);
}

void writeTo20()
{
    // variables for time measurement
    unsigned int clkStart = 0, clkEnd = 0;
    printf("Processor %d: Writing to (2,0) %d repeats...\n",
         __processorID, REPEAT_COUNT);
    clkStart = SHMAC_CLKCNT;
    // ----- start of microbenchmark kernel -----
    __asm__("li $4,0x20000000");
    __asm__(REPEAT_COUNT_ASM);
    __asm__("sw $5,0($4)");
    __asm__(".endr");
    // ----- end of microbenchmark kernel -----
    clkEnd = SHMAC_CLKCNT;

    // print statistics
    printf("Processor %d: Elapsed clock cycles: %d\n",
        __processorID, clkEnd-clkStart);
  printf("Processor %d: CPI: %d \n", __processorID, (clkEnd-
      clkStart) / REPEAT_COUNT);
}

void SHMAC_assignTasksToProcessors()
{
    int i;
    // reset all processor tasks
    for(i = 0; i < 16*16; i++)
        __processorTasks[i] = 0;
    // assign tasks
    __processorTasks[2] = &localInstrFetch;
    __processorTasks[16] = &readFrom11;
```

```
    __processorTasks[33] = &writeTo20;
}
```

A.2.4. Pure Memory R/W Performance - Multiple Masters

```
#include <stdio.h>
#include "../shmacsim_microbenchmark.h"

#define REPEAT_COUNT 2000
#define REPEAT_COUNT_ASM ".rept 2000"

ProcessorTask __processorTasks[SHMAC_MAX_GRIDDIM*
    SHMAC_MAX_GRIDDIM];
unsigned int __processorID;

// perform write on neighbour if 1, perform read otherwise
int writeNeighbour;

// task function definitions
void access11();

int main(int argc, char *argv[])
{
    // initialization
    __processorID = SHMAC_PROCID;
    SHMAC_assignTasksToProcessors();
    // retrieve task for this processor
    ProcessorTask myTask = __processorTasks[__processorID];

    // benchmark-specific cmdline setup
    writeNeighbour = 0;
    if(argc > 0 && argv[0][0] == 'w')
        writeNeighbour = 1;

    if(!myTask)
        printf("Processor %d was not assigned any tasks,
            exiting...\n", __processorID);
    else
        myTask();

  return 0;
}

void writeTo11()
{
    // variables for time measurement
    unsigned int clkStart = 0, clkEnd = 0;
```

```
    printf("Processor %d: Writing to (1,1) %d repeats...\n",
         __processorID, REPEAT_COUNT);
    clkStart = SHMAC_CLKCNT;
    // ----- start of microbenchmark kernel -----
    __asm__("li $4,0x11000000");
    __asm__(REPEAT_COUNT_ASM);
    __asm__("sw $5,0($4)");
    __asm__(".endr");
    // ----- end of microbenchmark kernel -----
    clkEnd = SHMAC_CLKCNT;

    // print statistics
    printf("Processor %d: Elapsed clock cycles: %d\n",
        __processorID, clkEnd-clkStart);
  printf("Processor %d: CPI: %d \n", __processorID, (clkEnd-
      clkStart) / REPEAT_COUNT);
}

void readFrom11()
{
    // variables for time measurement
    unsigned int clkStart = 0, clkEnd = 0;
    printf("Processor %d: Reading from (1,1) %d repeats...\n
        ", __processorID, REPEAT_COUNT);
    clkStart = SHMAC_CLKCNT;
    // ----- start of microbenchmark kernel -----
    __asm__("li $4,0x11000000");
    __asm__(REPEAT_COUNT_ASM);
    __asm__("lw $5,0($4)");
    __asm__(".endr");
    // ----- end of microbenchmark kernel -----
    clkEnd = SHMAC_CLKCNT;

    // print statistics
    printf("Processor %d: Elapsed clock cycles: %d\n",
        __processorID, clkEnd-clkStart);
  printf("Processor %d: CPI: %d \n", __processorID, (clkEnd-
      clkStart) / REPEAT_COUNT);
}

void access11()
{
    // perform read or write depending on cmdline arg
    if(writeNeighbour)
        writeTo11();
    else
        readFrom11();
}
```

```
void SHMAC_assignTasksToProcessors()
{
    int i;
    // set all processor tasks
    for(i = 0; i < 16*16; i++)
        __processorTasks[i] = &access11;
}
```

A.2.5. Remote Impact on Local Fetch

```
#include <stdio.h>
#include "../shmacsim_microbenchmark.h"

#define REPEAT_COUNT 2000
#define REPEAT_COUNT_ASM ".rept 2000"

// perform write on neighbour if 1, perform read otherwise
int writeNeighbour;

ProcessorTask __processorTasks[SHMAC_MAX_GRIDDIM*
    SHMAC_MAX_GRIDDIM];
unsigned int __processorID;

// task function definitions
void localInstrFetch();
void acess11();

int main(int argc, char *argv[])
{
    // initialization
    __processorID = SHMAC_PROCID;
    SHMAC_assignTasksToProcessors();
    // retrieve task for this processor
    ProcessorTask myTask = __processorTasks[__processorID];

    // benchmark-specific cmdline setup
    writeNeighbour = 0;
    if(argc > 0 && argv[0][0] == 'w')
        writeNeighbour = 1;

    if(!myTask)
        printf("Processor %d was not assigned any tasks,
            exiting...\n", __processorID);
    else
        myTask();

  return 0;
}
```

```
void localInstrFetch()
{
    // variables for time measurement
    unsigned int clkStart = 0, clkEnd = 0;
    printf("Processor %d: Local memory read, %d repeats...\n
        ", __processorID, REPEAT_COUNT);
    clkStart = SHMAC_CLKCNT;
    // ----- start of microbenchmark kernel -----
    __asm__("li $4,0");
    __asm__(REPEAT_COUNT_ASM);
    __asm__("addiu $4,$4,1");
    __asm__(".endr");
    // ----- end of microbenchmark kernel -----
    clkEnd = SHMAC_CLKCNT;

    // print statistics
    printf("Processor %d: Elapsed clock cycles: %d\n",
        __processorID, clkEnd-clkStart);
  printf("Processor %d: CPI: %d \n", __processorID, (clkEnd-
      clkStart) / REPEAT_COUNT);
}

void writeTo11()
{
    // variables for time measurement
    unsigned int clkStart = 0, clkEnd = 0;
    printf("Processor %d: Writing to (1,1) %d repeats...\n",
         __processorID, REPEAT_COUNT);
    clkStart = SHMAC_CLKCNT;
    // ----- start of microbenchmark kernel -----
    __asm__("li $4,0x11000000");
    __asm__(REPEAT_COUNT_ASM);
    __asm__("sw $5,0($4)");
    __asm__(".endr");
    // ----- end of microbenchmark kernel -----
    clkEnd = SHMAC_CLKCNT;

    __asm__("lw $5,0($4)");

    // print statistics
    printf("Processor %d: Elapsed clock cycles: %d\n",
        __processorID, clkEnd-clkStart);
  printf("Processor %d: CPI: %d \n", __processorID, (clkEnd-
      clkStart) / REPEAT_COUNT);
}

void readFrom11()
{
```

```
    // variables for time measurement
    unsigned int clkStart = 0, clkEnd = 0;
    printf("Processor %d: Reading from (1,1) %d repeats...\n
        ", __processorID, REPEAT_COUNT);
    clkStart = SHMAC_CLKCNT;
    // ----- start of microbenchmark kernel -----
    __asm__("li $4,0x11000000");
    __asm__(REPEAT_COUNT_ASM);
    __asm__("lw $5,0($4)");
    __asm__(".endr");
    // ----- end of microbenchmark kernel -----
    clkEnd = SHMAC_CLKCNT;

    // print statistics
    printf("Processor %d: Elapsed clock cycles: %d\n",
        __processorID, clkEnd-clkStart);
  printf("Processor %d: CPI: %d \n", __processorID, (clkEnd-
      clkStart) / REPEAT_COUNT);
}

void access11()
{
    // perform read or write depending on cmdline arg
    if(writeNeighbour)
        writeTo11();
    else
        readFrom11();
}

void SHMAC_assignTasksToProcessors()
{
    int i;
    // set all processor tasks to access11
    for(i = 0; i < 16*16; i++)
        __processorTasks[i] = &access11;
    // ...except (1,1) itself, do local instr fetch there
    __processorTasks[17] = &localInstrFetch;
}
```

A.2.6. Lock Time Acquisition

```
#include <stdio.h>
#include "../shmacsim_microbenchmark.h"

#define LOCK_ADDR       0x44000000
#define xStringify(s) stringify(s)
#define stringify(s) #s
```

```
ProcessorTask __processorTasks[SHMAC_MAX_GRIDDIM*
    SHMAC_MAX_GRIDDIM];
unsigned int __processorID;

// task function definitions
void testLockTime();

int main(int argc, char *argv[])
{
    // initialization
    __processorID = SHMAC_PROCID;
    SHMAC_assignTasksToProcessors();
    // retrieve task for this processor
    ProcessorTask myTask = __processorTasks[__processorID];

    if(!myTask)
        printf("Processor %d was not assigned any tasks,
            exiting...\n", __processorID);
    else
        myTask();

  return 0;
}

void testLockTime()
{
    // variables for time measurement
    unsigned int clkStart = 0, clkEnd = 0;
    printf("Processor %d: Test lock time...\n",
        __processorID);
    clkStart = SHMAC_CLKCNT;
    // ----- start of microbenchmark kernel -----
    // address of variable to lock
    __asm__ ("li $4," xStringify(LOCK_ADDR));
    // acquire lock
    __asm__ ( "initmutex:" );
    __asm__ ( "li $1, 1" );
    __asm__ ( "getmutex:" );
    __asm__ ( "ll $2, ($4)" );
    __asm__ ( "bne $zero, $2, getmutex" );
    __asm__ ( "sc $1, ($4)" );
    __asm__ ( "beq $zero, $1, initmutex" );
    // ----- end of microbenchmark kernel -----
    clkEnd = SHMAC_CLKCNT;

    // release lock
    volatile unsigned int * lockVar = (volatile unsigned int
        *) LOCK_ADDR;
    *lockVar = 0;
```

```
    // print statistics
    printf("Processor %d: Elapsed clock cycles: %d\n",
        __processorID, clkEnd-clkStart);
}


void SHMAC_assignTasksToProcessors()
{
    int i;
    // set all processor tasks
    for(i = 0; i < 16*16; i++)
        __processorTasks[i] = &testLockTime;
}
```