NTNU – Trondheim
Norwegian University of
Science and Technology

Implementing a Bare-Metal Threading Library for SHMAC

Håkon Opsvik Wikene

Master of Science in Computer Science

Submission date:	June 2014
Supervisor:	Magnus Jahre, IDI
Co-supervisor:	Donn Morrison, IDI

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

Problem Description

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 evaluate the performance of this prototype. Computers are normally evaluated with test programs called benchmarks.

The task in this assignment is to implement a functional threading library with the intention to support multi-threaded applications. If the student has sufficient time, a set of multi-threaded benchmarks (such as PARSEC) that meet the hardware requirements of SHMAC can be ported. Furthermore, as the process of setting up a working build environment for SHMAC is currently both tedious and error-prone, the student should investigate possible solutions to this problem in order to simplify the process and decrease overhead in new setups. This may additionally include an automatic test suite framework for SHMAC to be used to automatically test newly checked in code.

Abstract

For decades, Moore's Law has stood as a symbol of the continued performance increases achieved through technology scaling. While Moore's observation has remained true for far longer than Moore himself predicted, it now seems to be coming to an end. The move to multi-core processors was motivated by thermal challenges, but is by no means a final solution. Heterogeneous systems could potentially achieve far greater energy efficiency than symmetric, multi-core architectures. SHMAC is an FPGA-based architecture that intends to serve as a platform for exploring the inherent challenges of heterogeneous, single-ISA systems.

Since working with parallel applications on bare-metal platforms can be difficult, a higher level of abstraction is often preferable. One such abstraction that is familiar to most developers is the POSIX Threads API. This thesis covers the implementation of a bare-metal Pthreads library that provides both threading and several other synchronization primitives, such as condition variables, mutexes and barriers. The implemented library is used to port several benchmarks from the PARSEC suite; the results from these are presented both to showcase the current performance of the platform and the capabilities of the Pthreads library.

As a result of the need to verify the correctness and performance of the implemented library, the need for another tool arose: a profiler. Since none currently existed for SHMAC, one had to be written from scratch. A fully functional and *gprof*-compatible profiler was developed. The features, implementation details and usage of this profiler is described here as well.

Two other contributions to the SHMAC project were made as well. These were intended to simplify the task of setting up a working development environment and the day-to-day work with SHMAC, respectively. These are presented in separate reports.

Sammdrag

I mange tiår har Moore's Lov stått som et symbol på den kontinuerlige ytelsesøkningen som har vært oppnådd gjennom skalering av eksisterende teknologi. Til tross for at Moore's Lov har holdt lenger enn Moore selv hadde forutsett, er det nå grunn til å tro at denne utviklingen går mot slutten. Overgangen til flerkjerneprosessorer var i hovedsak motivert av termiske utfordringer, men er ingen endelig løsning på problemet. Heterogene systemer har potensialet til å oppnå langt høyere energieffektivitet enn symmetriske flerkjerneprosessorer kan.

SHMAC er en FPGA-basert arkitektur som har til hensikt å være en plattform for utforsking av de mulige utfordringene som følger med heterogene, enkelt-instruksjonssett arkitekturer.

Siden det kan være utfordrende å jobbe med parallelle applikasjoner på operativsystemløse plattformer, kan et høyere abstraksjonsnivå ofte være å foretrekke. Et eksempel på et slikt abstraksjonsnivå er POSIX' applikasjonsprogrammeringsgrensesnitt for tråder. Denne masteroppgaven omhandler implementasjonen av et operativsystemløst, POSIX-kompatibelt trådbibliotek som tilbyr både tråding og flere andre velkjente synkroniseringsprimitiver. Det implementerte biblioteket brukes deretter til å tilpasse flere eksisterende ytelsestester fra PARSEC til SHMAC-plattformen. Resultatet av disse presenteres deretter, både for å demonstrere ytelsen til plattformen og for å vise trådbibliotekets kapabiliteter.

Behovet for å verifisere korrektheten og vurdere ytelsen til det implementerte biblioteket gjorde det klart at et verktøy til trengtes: en profilerer. Siden det per dags dato ikke eksisterer for SHMAC måtte en utvikles fra bunnen av. En fullt kapabel og *gprof*-kompatibel profilerer ble utviklet. Implementasjonen og bruken av denne vil også beskrives.

To ytterligere bidrag til SHMAC-prosjektet ble også gjort. Hensikten bak disse var henholdsvis å forenkle oppsettet av nye utviklingsmiljø og automatisere en del av det daglige utviklingsarbeidet med SHMAC. Disse bidragene er presentert i to separate rapporter.

Preface

This thesis concludes my Master of Science education in Computer Science at the Norwegian University of Science and Technology (NTNU). The work was carried out in my 10th semester, spring 2014, at the Department of Computer and Information Science under the supervision of associate professors Magnus Jahre and Donn Morrison.

Acknowledgments

I would like to thank my main supervisor Magnus Jahre for giving me the opportunity to participate in the SHMAC project, which has been both challenging and highly rewarding. For his continuous support and feedback, co-supervisor Donn Morrison also deserves a special thanks.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

APB	Advanced Peripheral Bus, page 13.
API	Application Programming Interface, page 6.
ARM	ARM Holdings plc, page 11.
CABAC	Context-Adaptive Binary Arithmetic Coding, page 74.
CFS	Completely Fair Scheduler, page 16.
CMP	Chip Multiprocessor, page i.
EECS	Energy Efficient Computing Systems, page 3.
EPS	Encapsulated PostScript, page 52.
FP	Floating Point, page 13.
FPGA	Field-Programmable Gate Array, page i.
GOP	Group of Pictures, page 74.
MMU	Memory Management Unit, page 78.
NFS	Network File System, page 11.
NTNU	Norwegian University of Science and Technology, page 3.
PARSEC	Princeton Application Repository for Shared-Memory Computers, page i.
PC	Program Counter, page 48.
PID	Process Identifier, page 28.
POSIX	Portable Operating System Interface, page 4.
PTE	POSIX Threads for Embedded systems, page 19.
RAMP	Research Accelerator for Multiple Processors, page 10.
RISC	Reduced Instruction Set Computer, page 12.

SHMAC	Single-ISA Heterogeneous Many-Core Computer, page 3.
SRAM	Static Random Access Memory, page 11.
TCG	Task Control Groups, page 17.

Chapter 1

Introduction

Since the birth of the modern computing era, there has been a race towards increased performance. For decades, this continued trend of exponentially increased performance has been synonymous with Moore's Law: the observation that the number of transistors on a chip roughly doubles every two years [13, 14]. In the early 2000s, several challenges arose[1].

To address these challenges, multicore processors were introduced. These offer great performance improvements to applications that can leverage multiple threads of execution. However, they still suffer from certain problems, chief of which is high power consumption.

One suggested solution to this challenge is to incorporate heterogeneous processing cores on a single chip. Heterogeneous computing can potentially offer superior performance and power efficiency to that of homogeneous processors. This is the main motivation behind the SHMAC project.

However, SHMAC is currently missing one feature that has been ubiquitous for around two decades: software support for multitasking. Applications expect to be able to spawn as many processes/threads as they want, and they expect those to run concurrently. If the number of running threads exceed the number of available processing units, the operating system should provide the illusion of concurrency.

As SHMAC currently does not have an operating system[2], writing parallel applications is a challenge. Porting existing ones is completely unfeasible for all but the simplest applications.

This is the motivation behind developing a bare-metal threading library for SHMAC: to both simplify application development and the porting of existing applications. This master's thesis describes such a threading library.

[1]See section 2.1 for details.

[2]Although porting existing operating systems – specifically Linux and Barrelfish [2] – to SHMAC is ongoing.

1.1 The SHMAC Project

EECS (Energy Efficient Computing Systems) is a research initiative started at NTNU in 2012 by the Faculty for Information Technology, Mathematics and Electrical Engineering. Its main focus is on achieving energy efficiency of computing systems by employing a vertically integrated approach across abstraction layers, from low-level electronics to high-level software.

SHMAC (Single-ISA Heterogeneous Many-Core Computer) [17] is one of EECS's research projects, initiated to investigate the challenges posed by heterogeneous architectures. Heterogeneity is one solution proposed to tackle the challenge known as Dark Silicon, a limiting factor in current CMP technology scaling. It is as such an important area of current research.

More specifically, SHMAC is an FPGA-based platform that allows for quick instantiation of tile-based, heterogeneous architectures. The architectural details of SHMAC are further detailed in Section 2.3.

1.2 Assignment Interpretation

The problem description presented on page i presents a clear task: implement a threading library so that multithreaded benchmarks can more easily be ported to the SHMAC platform. Specifically, two main tasks can be extracted from the description:

Task 1 (mandatory): Implement a functional threading library.

Task 2 (optional): Port one or more existing benchmarks that rely on threading to SHMAC.

The choice of threading library and benchmarks to be ported are left unspecified. As such, evaluating available threading libraries and making a decision is considered a natural part of the assignment. This, in turn, will have consequences for the possible benchmarks that can be ported. Some existing threading libraries are discussed in Section 2.5. Benchmarks are evaluated in Chapter 6.

Two additional tasks are given in the assignment not related to benchmarking:

Task 3 (mandatory): Investigate ways of simplifying the task of setting up a complete toolchain and C standard library for development for the SHMAC platform.

Task 4 (mandatory): Implement a testing framework for running automated tests on SHMAC.

These two tasks are addressed in separate technical reports [21, 22] to keep this thesis focused on the main task. They are also included as appendixes: Appendix A and Appendix B, respectively.

1.3 Contributions

The contributions of this work are as follows:

1. A fully working, POSIX-compliant threading library has been developed. This allows a large number of benchmarks (and other parallel applications) to more easily be ported to SHMAC.

2. The toolchain setup process has been automated by a setup script. This simplifies the previously error-prone task of setting up a complete build environment for SHMAC.

3. A simple test framework has been developed for SHMAC. Previous applications and benchmarks developed for SHMAC have all been written independently of one another without regard for a unified build process.

   This has led to every application having their own set of idiosyncrasies. For instance, it was not certain that the default *Make* [3] rule would build a working application suitable for testing.

   This test framework should encourage users to adhere to certain standards of development that makes it easy to integrate new benchmarks into the ever-growing benchmark suite.

4. A low-overhead profiler was also written to validate the performance of the scheduler. While not a part of the assignment, it became a necessity to reason about the performance of the scheduler and present quantitative results from it.

   The profiler combines function instrumentation and link register sampling to provide a complete[4] function-level profile. The profiler output is self-explanatory enough that anyone can write analysis tools for it. An analyzer capable of printing function-level statistics, critical paths and call trees was also written. For those more comfortable working with the familiar *gprof* [8], a conversion tool was made to convert the output to *gprof*'s binary format.

5. Finally, a set of parallel benchmarks were ported to SHMAC and evaluated using the newly developed threading library. These can both provide a reference point for evaluating future accelerators and serve as starting point for extending benchmarks with accelerator support.

[3] `http://www.gnu.org/software/make/`

[4] Meaning it captures the call tree precisely without statistical inaccuracies and it does not need to make assumptions about program execution the way GNU Prof does. This is explained further in Chapter 4.

1.4 Thesis Organization

The rest of the thesis is organized as follows:

Chapter 2 presents a brief history of traditional CPU architectures to motivate to need for heterogeneity. The SHMAC project (and prototype) is then introduced as a research platform for heterogeneous systems.

Since the main task is to implement a functional threading library, an understanding of schedulers is essential. Chapter 2 introduces some of the latest schedulers employed by the Linux kernel, as they provide useful insight into possible scheduler designs and their strengths and weaknesses.

The chapter is rounded off by a brief description of existing threading libraries intended for embedded platforms.

Chapter 3 presents the implemented scheduler (and its threading API) in detail. Every component is described in detail, design decisions are explained and possible alternatives are evaluated. The API implementation coverage, unit testing coverage and source code structure are presented as well.

Chapter 4 describes an instrumentation-based profiler that was developed in order to reason about the performance of the threading library. This was not a part of the assignment, but might nevertheless be useful in future work on SHMAC and is thus included here. An understanding of how the performance results were collected is also useful background information to have when the scheduler's performance evaluation results are presented, which is done in Chapter 5.

In chapter 6, some well-known benchmarks that rely on Pthreads are run on top of the SHMAC threading library. The results of these are presented and discussed.

Finally, chapter 7 concludes the paper and reiterates important results. Future work is also suggested.

Chapter 2

Background

This chapter presents some necessary background information, both on the motivation for heterogeneity and schedulers in general. Recognizing the need for heterogeneous hardware is essential in understanding the motivation behind the SHMAC project. A short historical recap of processor designs is given, after which the SHMAC architecture is presented.

Finally, some background information is presented on the topic of task scheduling. The design decisions behind the scheduler that was implemented are motivated by both the limitations of the SHMAC platform, as well as existing schedulers and their goals. While the assignment was to implement a threading library, the core component is a full-fledged scheduler – the POSIX Threads standard just defines the API. Hence, a basic understanding of existing schedulers is necessary.

2.1 Processor Design: A Historical Perspective

The single core processor has come a long way since the very first single-chip processor, the Intel 4004, released in 1971. From the mid-80s until the early 2000s, performance increased with more than 50% per year. This performance was achieved through a combination of die shrinking and microarchitectural breakthroughs. Figure 2.1 shows the number of transistors in a processor since 1971.

As we can see, the number of transistors on chip has stayed in line Moore's prediction since the seventies. However, sub-threshold leakage and slowed supply voltage scaling prevents transistors from achieving similar improvements in the future [4].

As we keep scaling down transistors, leakage power becomes a growing problem. When leakage power dominates the active power, power consumption is roughly proportional to the transistor count. However, Pollack's Rule [5] tells us that performance increase is proportional to only the square root of the die area. These problems of diminishing returns and high

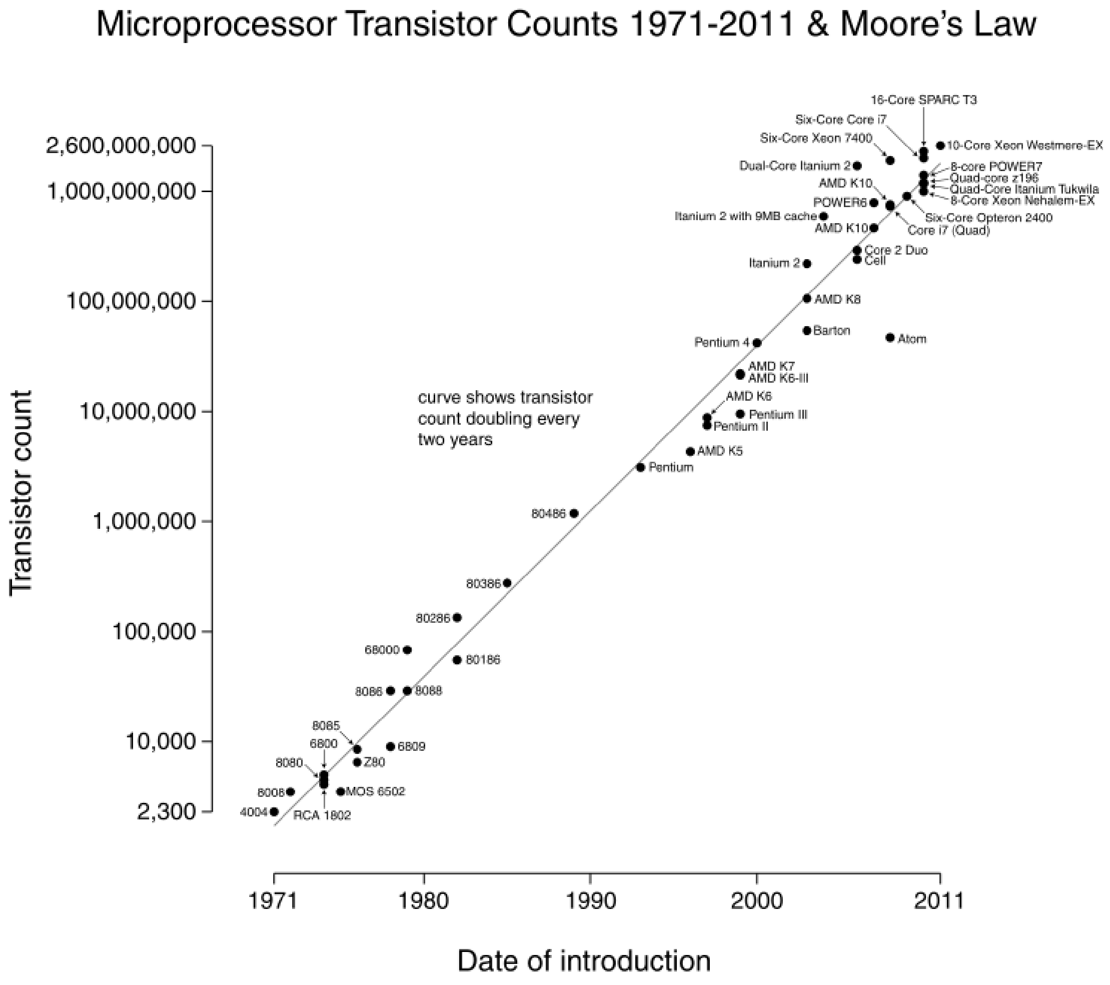


Figure 2.1: Transistor count in typical processors throughout the last few decades, in line with the predictions of Moore's Law.

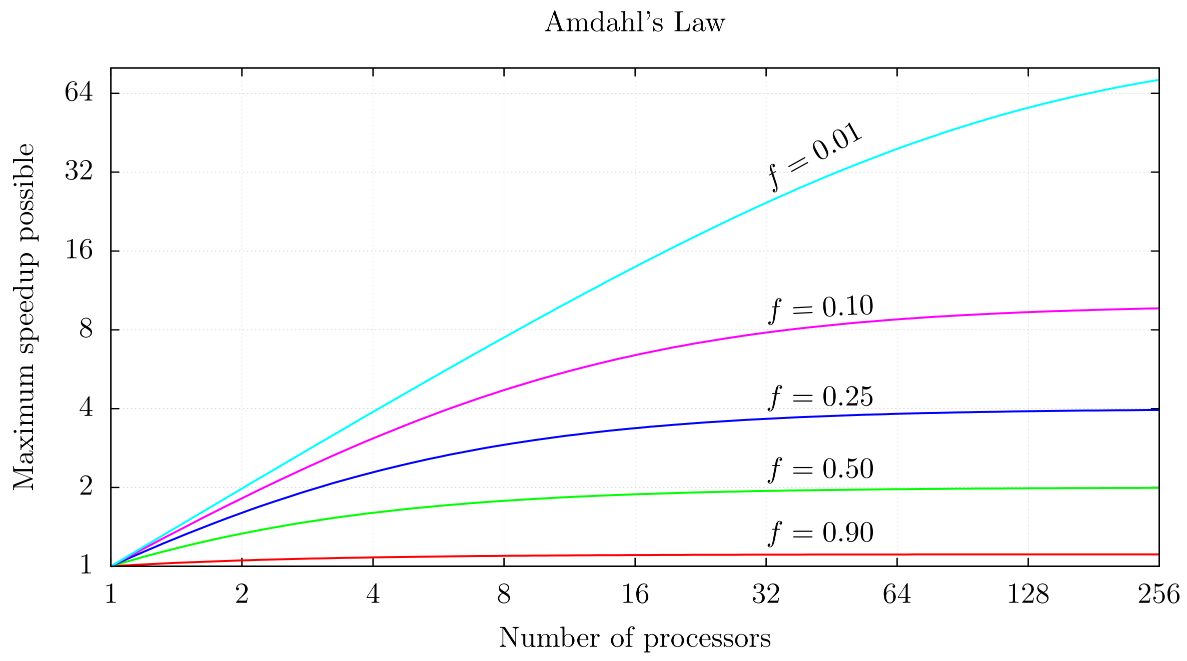


Figure 2.2: Amdahl's Law illustrates the maximum attainable speedup for a given serial fraction f.

power consumption are in part what motivated an industry-wide shift towards multicore architectures.

The switch to homogeneous multicore processors partly solves the problem of Pollack's Rule: we can now scale performance linearly[1] as a function of power. In addition, each core can generally be made simpler if we can compensate by using more than one. This works well for a few cores, as evident by the abundance of dual- and quad-core processor readily available on the consumer market. Unfortunately, numerous problems remain.

Amdahl's Law, for one, presents a bleak view of multi-core architectures: for programs with an inherently serial fraction f, the speedup will never exceed $1/f$. This is illustrated for different values of f in Figure 2.2. For most problems, the serial fraction of an application is significant, maybe even dominant. No number of homogeneous cores can improve on this upper bound.

Another challenge is that most software is not written for parallel execution. Taking advantage of the possible performance benefits from multicore processors require software to be rewritten, a task that is generally not trivial.

Even for problems that can be split among hundreds of cores, the power consumption would be too high to dissipate. This problem would suggest the need for lower per-core power consumption, which in turn reduces per-core performance. With this solution, non-parallelizable applications would suffer.

[1] As long as the problem to be solved is easily parallelizable.

Desktop- and server systems are not the only ones that suffer. The emergence of mobile computing systems has also led to an increased focus on energy efficiency as a goal to achieve longer battery lives, less heat dissipation and smaller units [7].

From a wider perspective, the last 5-7 years can be seen as a paradigm shift from area-constrained to energy-constrained computing [16]. Simultaneously increasing performance and decreasing energy consumption is a significant challenge for the computer industry.

2.2 Recent Steps Toward Heterogeneity

The aforementioned challenges has led to the proposal of heterogeneous computing, in which processor cores admit some sort of asymmetry. Asymmetric systems are often further classified as either performance- or functionally asymmetric. Performance-asymmetric cores can again be partitioned into two groups: cores with identical microarchitectures and cores with different microarchitectures. Functional asymmetry can be either overlapping or non-overlapping. This gives us the following gradation of asymmetry:

1. Single-ISA, identical microarchitectures, different power and frequency characteristics.
2. Single-ISA, different microarchitectures. These can be, for instance, have different pipelines or differ in being out-of-order or supporting speculative execution.
3. Overlapping ISA, but not identical. These often differ in a small subset of instructions.
4. Different ISAs. Examples include integrated CPU/GPU on a single chip, or accelerator-based architectures such as the IBM Cell.

In general, the more homogeneous the processor the easier it is to program. In contrast, the more heterogeneous processors can potentially offer more efficiency and speed at the cost of implementation complexity.

Over the last decade, the questions of how to design and program heterogeneous hardware have been thoroughly researched. However, not much heterogeneous hardware exist, especially in the consumer segment.

Previous work has showed that only performance asymmetry can yield a significant increase in both power efficiency and performance [11, 12]. In fact, most of the benefit of performance asymmetry can be reaped with as little as two cores [19].

2.2.1 Research Initiatives Into Heterogeneous Hardware

Since Kumar, et. al. first proposed single-ISA heterogeneous multicore processors in 2003, several projects have been initiated to research new architectures. This subsection provides

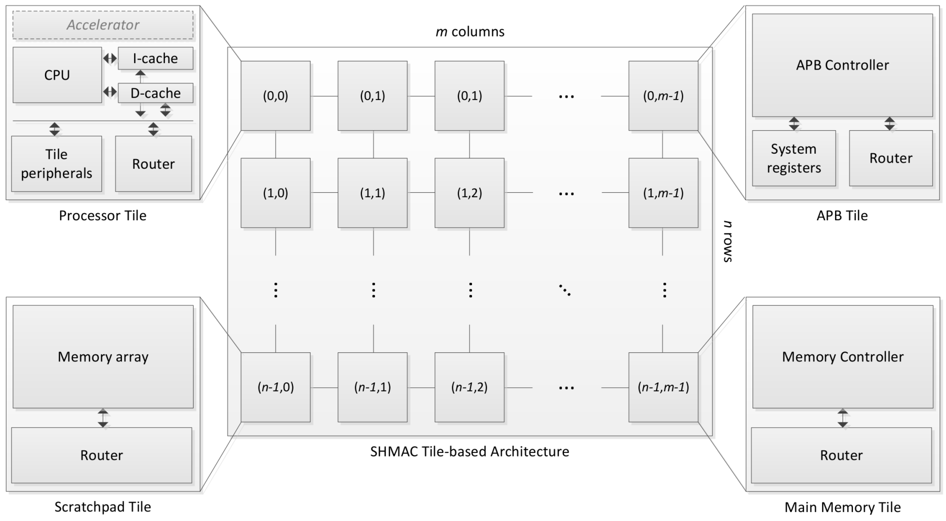


Figure 2.3: An overview of the SHMAC architecture.

a quick overview of some of these.

ATLAS [20] is a FPGA-based platform for CMP research developed at Stanford University. Using a multi-FPGA board, they provide a system with 8 PowerPC cores running at 100MHz, capable of running Linux. The *ATLAS* prototype provides support for transactional memory, a technique intended to simplify parallel programming. In likeness to SHMAC, they argue that an FPGA-based research platform provides a tremendous performance advantage over software simulators.

HAsim [15] intends to accelerate performance modeling of multicore processors by using FPGAs. It provides a cycle-accurate framework for modeling shared-memory CMPs, among other contributions. *RAMP Gold* [18] is another FPGA-based, cycle-accurate architecture simulator intended for early design-space exploration.

Heracles [10] is a complete, open-source multicore system toolkit. It is intended as a tool for fast exploration of future multicore processors as well as being a teaching tool. It is also FPGA-based and written in Verilog. It is modular, in the sense that processing elements, memory configuration and routing settings can easily be reconfigured. A toolchain is provided to map applications written in C/C++ onto core units. They provide a graphical user interface to simplify configuration and the launching of new architectures.

2.3 The SHMAC Project

The SHMAC architecture can succinctly be summed up as *a tile-based architecture with processing- and memory elements laid out in a rectangular grid with a shared memory space and, optionally, tile-local peripherals*. The architecture is displayed in Figure 2.3.

The current FPGA prototype was developed as a master's thesis project in 2012 [17]. Since then, it has undergone some architectural changes. This section will describe the current design to the level of detail necessary for reasoning about the platform from a software perspective.

2.3.1 Hardware Platform

The SHMAC prototype is is built on the ARM RealView Versatile Platform. This is a development platform consisting of a processor (ARM11 MPCore) and I/O devices. For operating system, it runs Rasbian Linux, kernel version 3.8.8. Since the Versatile Platform does not have any persistent storage, the root filesystem is mounted over NFS.

The Platform baseboards can also be expanded by adding Logic Tiles[2] or Core Tiles[3]. The SHMAC prototype runs on a connected Logic Tile: a Xilinx Virtex 5 XC5VLX330 FPGA. This FPGA is controlled by software on the host system.

A kernel module (`shmac.ko`) sets up two character devices used to communicate with SHMAC: `/dev/shmac` and `/dev/ttySHMAC0`. These are used to send commands to SHMAC (reset, load program and read/write memory space) and read/write standard input/output, respectively. This is explained in more detail in Section **??**.

2.3.2 Memory Space

SHMAC's memory layout has been redesigned since Rusten and Sortland's [17] work. This section will describe the current layout.

SHMAC has two types of memory: one 32MB SRAM chip ("Z-tile") that serves as main memory and memory tiles that each can hold 16kB of data. These memory tiles ("R-tiles") use the FPGA's block RAM. By default, these R-tiles are not used for anything.

The main memory is mapped to the start of the address space, which means that the SRAM will always occupy the range [0x0, 0x2000000). All the required memory segments should (most of the time, at least) be mapped into this region. Since each core starts by executing address 0x0, the startup vector must be placed here. However, other than that, the developer is free to chose any segment mapping he/she wishes.

Figure 2.4 shows the SHMAC memory space in detail. Every core shares all memory segments, such as `.text`, `.bss` and `.data`. There is also a shared memory heap. libc protects the heap from corruption caused by simultaneous access by locking it whenever necessary.

There is no virtual memory or memory protection in place. This means that if core 1 overflows

[2] `arm.com/products/tools/development-boards/versatile/logictiles.php`.
[3] `arm.com/products/tools/development-boards/versatile/coretiles.php`.

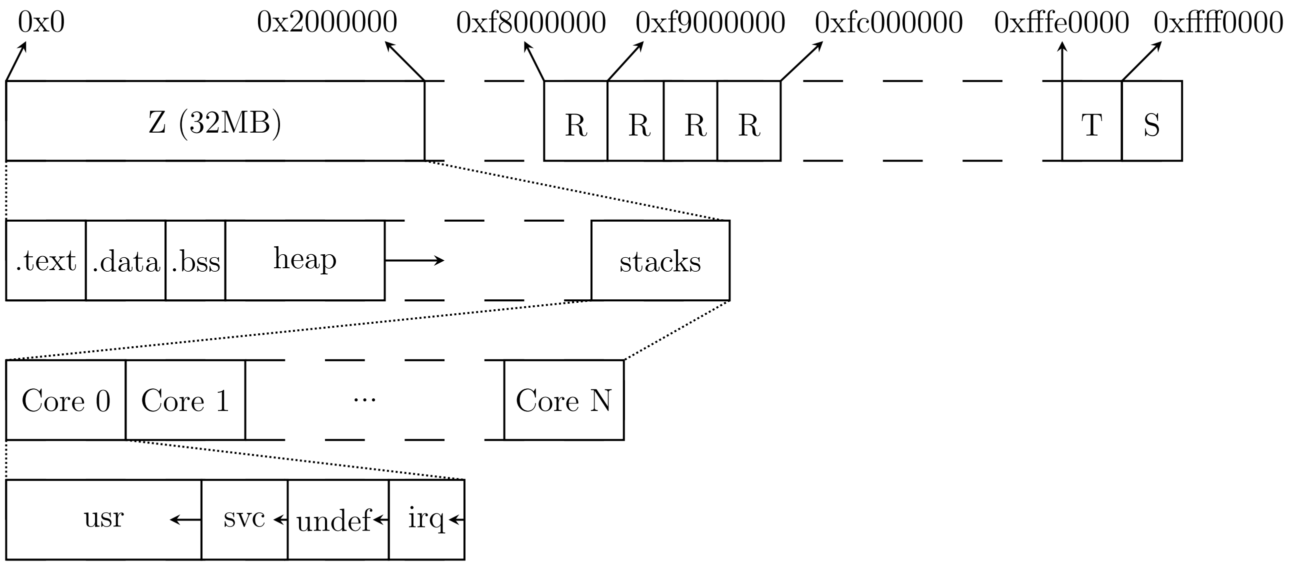


Figure 2.4: The SHMAC memory space. Z is the external memory space, while R are memory tiles, each holding 16kB. T and S are memory-mapped components and/or registers that are either tile-local og system-wide. Each core has four different stacks set up, one for each of the processor modes that are used

its stack in user mode, core 0 will immediately get a corrupted interrupt-stack[4].

Since all tiles (including memory-tiles, both Z and R) are placed in a grid, memory requests must be routed to the appropriate tile somehow. This is done by simple XY-routing: the request first moves along the X-axis, then the Y-axis. This affects both the latency of memory operations and the congestion at different routers in the grid, which is worth keeping in mind.

2.3.3 Cores and ISA

While development of additional cores (and accelerators) is in progress, there was really only one computational tile ready for use as of writing this: the Amber 25 core. While the Amber core is originally only ARMv2a compatible, it has been modified to support version 3 of the instruction set. This is a very simple RISC instruction set. As such, it incorporates the typical RISC features:

1. Large register file
2. Fixed-length instruction words
3. A simple load/store memory architecture

[4]As one might imagine, this has been the cause of many interesting bugs.

Name	**Description**
Amber 25 (A)	A 32-bit RISC ARMv2-compatible core No division or FP support; this is implemented in software.
Turbo Amber [1] (T)	A high performance core based on Amber. Adds fast multiplication, instruction buffer, branch prediction and more.
Scratchpad Tile (R)	A pure on-chip memory tile without processing capabilities. Each scratchpad tile currently has 16kB of memory.
Dummy Tile (.)	Tile containing only a router. This is needed to fill any ”gaps” in the tile layout.
APB Tile (V)	A core to interface with the host computer.
Main Memory Tile (Z)	Gives SHMAC access to off-chip memory, currently 32MB. In this report, only main memory is used unless otherwise stated.

Table 2.1: Tiles currently supported by SHMAC. The letters in parentheses are one-letter names used in the configuration when synthesizing each layout.

Amber executes most instructions in a single cycle[5]. For more details on the instruction set, the reader is referred to the ARM Architecture Reference Manual and OpenCores' project page for the Amber core[6].

There are a few less desirable traits worth noting at this point: there is no support for integer division or any type of floating point operations. These operations are provided as software routines by *libgcc*.

The current list of supported tiles are shown in Table 2.1.

2.4 A Brief History of Linux Schedulers

A central part of any operating system is the task scheduler, which decides which task to run and for how to run it. This is a vital – and often rather complex – part of any modern operating system. Studying several of the existing schedulers served as useful inspiration for designing one from scratch.

The Linux schedulers are an interesting case study for several reasons:

1. It is open source, so its source code is readily available
2. It has been widely adopted, both on the mobile, embedded, desktop and server market. One might thus assume that some effort has been put into its design and performance.

[5]Plus instruction fetch time, which on SHMAC is roughly 17 cycles.
[6]`http://opencores.org/project,amber`

3. Much has been written about the Linux schedulers, both current and historical ones.

In this section, the recent Linux schedulers will briefly be presented.

2.4.1 The O(n) Scheduler

The O(n) scheduler was used in Linux between version 2.4 and 2.6. Its name stems from the fact that it schedules task in linear time.

The scheduling is split into *epochs*, which are periods of time in which every task is allowed to run for a certain amount of time, called a *timeslice*. The timeslice is a function of a task's priority, which is specified through the task's *nice* value: a priority between -19 and 20, where 20 is the ”nicest” (meaning lowest priority).

All tasks are kept in a global runqueue, which is implemented as a linked list. The linear runtime stems from the scan of the global runqueue that is performed each time a task needs to be scheduled. The runnable task with the highest `goodness()` value will be scheduled next. This function basically adds up the remaining timeslice and the task priority. In addition, boosts are given to realtime tasks and tasks sharing address space; very large and fairly small boosts, respectively. A task with `p->counter == 0` will return a goodness of 0 and thus not be schedulable.

When no tasks are schedulable any more (used up their timeslice or blocked for some reason, presumably on I/O), it is recalculated for every task:

```
for_each_task(p)
    p->counter = (p->counter >> 1) + NICE_TO_TICKS(p->nice);
```

Listing 2.1: Recalculating timeslices in the O(n) scheduler

That is, the timeslice is calculated from the task priority plus half of what might have been remaining from the current epoch. This is done to give a slight bonus to I/O bound tasks, which are likely to have time remaining at the end of an epoch.

Whenever a task forks, its remaining timeslice is split between parent and child. This is to prevent processes from taking an unfair amount of CPU time by simply forking continuously.

In addition to this linked list, a global hash table is used to quickly map from PID to `struct sched_struct*` – aptly named `pidhash`.

This design is appealing for a couple of reasons. One being its simplicity. The scheduler is small, fairly easy to understand, and works well for many types of systems.

It does, however, have some unfortunate traits. The global runqueue is a point of lock contention that only gets worse as modern processors add more and more cores. The linear runtime of the scheduler is an obvious scalability bottleneck, at least in theory. More weaknesses are thoroughly described in [6].

2.4.2 The O(1) Scheduler

The O(1) scheduler was written by Ingo Molnar to address the shortcomings of the old O(n) scheduler; the greatest of which was its linear runtime. As this problem is deeply embedded in the scheduler design, the O(1) was a complete redesign.

Unlike the older O(n) scheduler, this one can schedule tasks in constant time; hence the name. It also maintains per-CPU runqueues instead of a single, global one.

Each runqueue consists of two priority arrays: one for active tasks and one for expired tasks. Active tasks are the ones which have yet to be scheduled in this epoch while expired ones have. It also contains a lock, pointer to the current and the special *idle* task, as well as numerous scheduling-related bookkeeping variables. An overview is shown in Figure 2.5.

Each priority array consists of three variables:

```
unsigned int nr_active;
unsigned long bitmap[BITMAP_SIZE];
struct list_head queue[MAX_PRIO];
```

Listing 2.2: `prio_array` definition in the O(1) scheduler

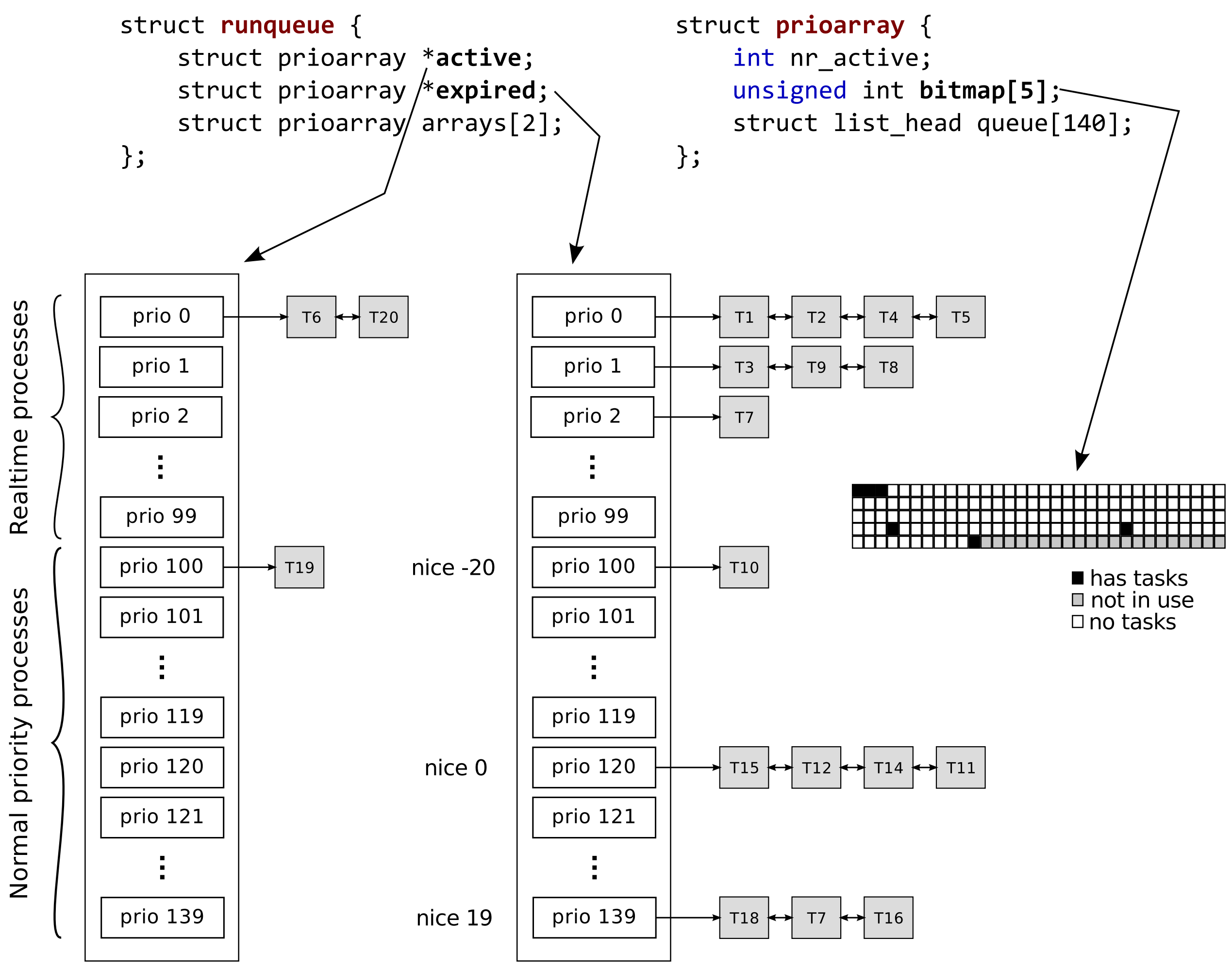
The MAX_PRIO macro is the number of distinct priorities used internally by the Linux kernel. It is the sum of the number of real-time tasks plus the number of distinct user priorities: $100 + 40 = 140$. BITMAP_SIZE is the number of bytes needed to express MAX_PRIO+1 different bits in a bitmap. This bitmap is used to quickly look up the highest-priority list_head; this is done by locating the most significant bit in the bitmap. The `nr_active` variable contains the number of non-empty runlevels.

The O(1) scheduler always schedules the task with the highest priority from the active array. Like the O(n) scheduler, different priorities are expressed through variable-length timeslices. Using the bitmap from the priority array, the highest non-empty queue is quickly located in constant time. It then selects the head of this queue: a constant time operation.

Once a task is finished, two important things happen:

1. The task's timeslice is recalculated for the next epoch
2. It is removed from the active array and appended to the expired array

If the newly expired task was the last one in the active array, the active and expired arrays are swapped by changing the two pointers. All of these operations can be performed in constant time, making the scheduler vastly more efficient than its predecessor.



```
struct runqueue {
    struct prioarray *active;
    struct prioarray *expired;
    struct prioarray arrays[2];
};
```

```
struct prioarray {
    int nr_active;
    unsigned int bitmap[5];
    struct list_head queue[140];
};
```

Figure 2.5: The entities involved in the O(1) scheduler. Tasks are gradually moved from `active` to `expired` as they finish. Tasks are maintained in linked lists per static priority, from 0 through 139. A bitmap is used to keep track of the linked lists that are non-empty so that the next task can quickly be located.

There are also quite complex heuristics involved in computing the effective priority of a task. Interactive tasks are given bonuses while CPU-intensive tasks are penalized. While this sounds simple enough, there are a number of scenarios that must be accounted for to achieve acceptable levels fairness. The result is a complex array of heuristics that strives to satisfy the end user. Still, there are complaints about the scheduler's performance with regard to interactivity.

2.4.3 CFS (The Completely Fair Scheduler)

The O(1) scheduler was a vast improvement upon the O(n) scheduler, but it still had its problems. One problem was the large body of code needed to compute heuristics. Ingo Molnar once again designed a new scheduler, inspired by Con Kolivas' work on a scheduler called the Rotating Staircase Deadline Scheduler.

The new scheduler took a completely new approach to scheduling: it wasn't based on priority arrays or runqueues at all. Instead, tasks are kept in a time-sorted red-black tree. Each node in the tree is keyed by a 64-bit field called `vruntime`, which expresses the amount of *virtual runtime* the task has been given.

With this design, identifying the next task to run is as simple as locating the leftmost node in the tree. Once it is finished running, its `vruntime` field can be updated and the tree rebalanced. Once again, the next task to run is the leftmost node in the tree.

Instead of using priorities directly (by, for instance, maintaining a separate tree for each priority), the CFS uses priorities as decay factors to the `vruntime`. That way time passes slower for high-priority tasks while it passes quicker for low-priority task. The key structures involved in scheduling are shown in Figure 2.6.

The CFS scheduler also introduced an important feature called *group scheduling*. Imagine two users running concurrently on a system, one with a single process and the other with 99. With traditional scheduling, one user would get 99% of the CPU. Group scheduling allows us to share the CPU time equally between users before further dividing it among per-user threads[7].

2.5 Existing Threading Libraries

There are some existing threading libraries that could have been ported instead of writing one from scratch. Some of these will be briefly discussed here.

2.5.1 TinyThread

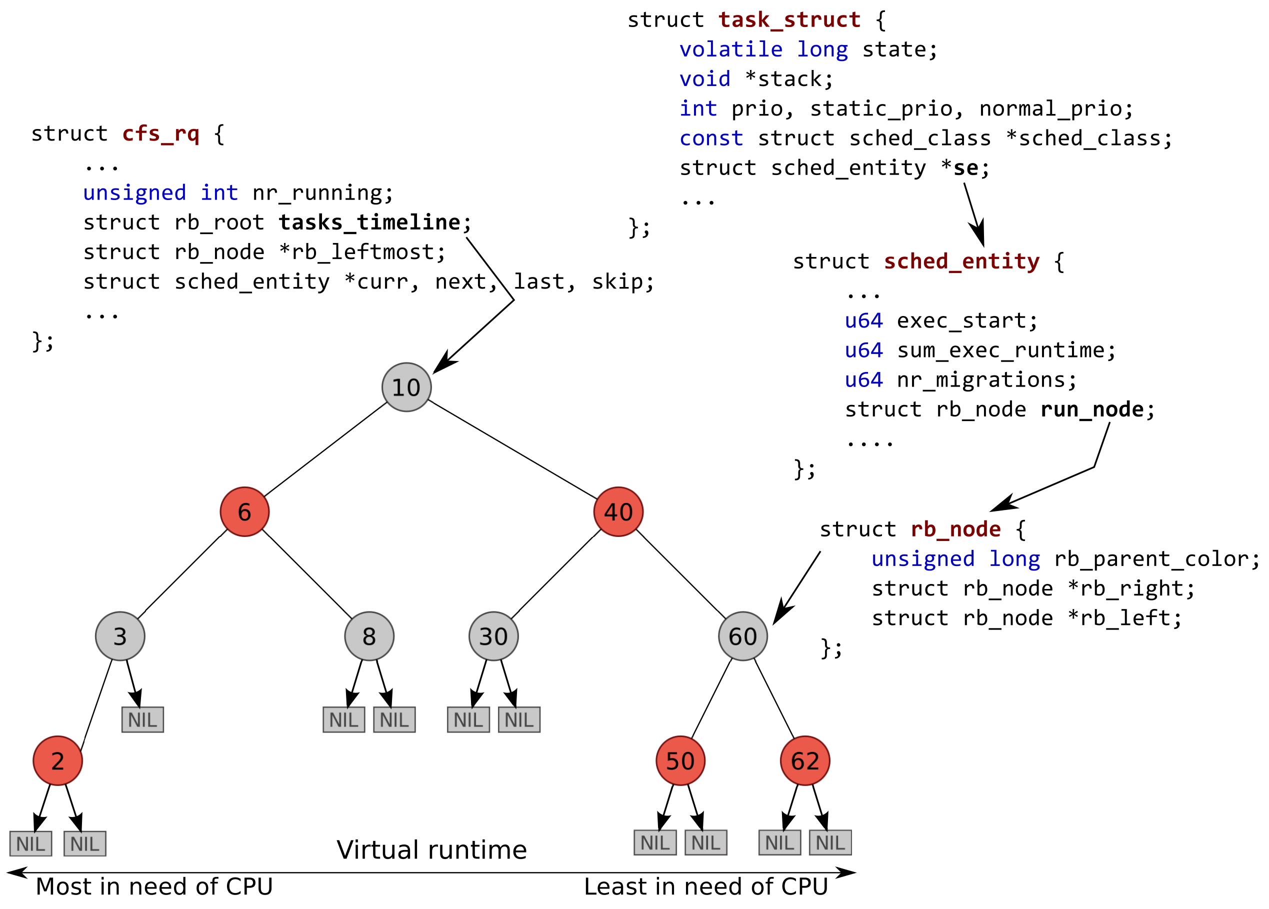
TinyThread[8] is a thread library with support for mutexes, semaphores, condition variables, message passing and simple threading.

In some ways, the library matches the needs of a scheduler for SHMAC:

1. It is preemptive
2. Since it supports Cortex M0 chips, porting any assembly code should be rather easy
3. It supports thread priorities

[7]The actual feature introduced, called TCG (Task Control Groups), has wider applicability outside of CFS.

[8]https://code.google.com/p/tinythread/



```
struct task_struct {
    volatile long state;
    void *stack;
    int prio, static_prio, normal_prio;
    const struct sched_class *sched_class;
    struct sched_entity *se;
    ...
};
```

```
struct cfs_rq {
    ...
    unsigned int nr_running;
    struct rb_root tasks_timeline;
    struct rb_node *rb_leftmost;
    struct sched_entity *curr, next, last, skip;
    ...
};
```

```
struct sched_entity {
    ...
    u64 exec_start;
    u64 sum_exec_runtime;
    u64 nr_migrations;
    struct rb_node run_node;
    ....
};
```

```
struct rb_node {
    unsigned long rb_parent_color;
    struct rb_node *rb_right;
    struct rb_node *rb_left;
};
```

Figure 2.6: The entities involved in the CFS scheduler. Tasks are kept in a red-black tree, ordered by their virtual runtime. Instead of keeping references ”back up” the reference tree (for instance from `rb_node` to `sched_entity`), this is done using C’s `offsetof()` macro.

However, after inspecting the code, it was found to not be of great value to the SHMAC project. Most of it would have had to be rewritten, either because of architectural differences or because of API incompatibility. The relevant parts of the project was around 1600 lines of code. The work involved in porting the code was thought to be comparable to that of writing similar functionality from scratch.

For that reason,the library was only ever used for inspiration; no code was ever directly taken from TinyThread.

2.5.2 POSIX Threads for embedded systems (PTE)

PTE[9] is an open source implementation of Pthreads designed to be easily portable. The API compatibility was an attractive quality in PTE and it was therefore considered carefully.

All the platform-specific components that must be provided are commonly referred to as the OS adaptation layer (OSAL). They must provide Functionality relating to threads, thread-local storage, mutexes, atomic operations, etc.

Some modifications and additions would have been necessary, but there is not really any good reason why the SHMAC pthread library could not have been based on PTE. In the end, the decision boiled down to wanting to write a scheduler from scratch rather than ”filling in the gaps” of an existing one.

[9] http://pthreads-emb.sourceforge.net/

Chapter 3

The SHMAC Scheduler

This chapter will present the scheduler that was implemented for this master thesis. After explaining its overall design, the scheduler will be described component by component. The library's completeness with respect to the POSIX standard is then addressed. The chapter is rounded off with short descriptions of the source code structure and unit test coverage, as well as known bugs and issues.

3.1 Overall Design

Before looking at every component in detail, here is a brief overview of how the scheduler works:

The first time a call to `pthread_create()` is made, the scheduler is initialized[1]. Up until that point, cores 1 through N has remained idle while core 0 has been executing `main()`, but not under the supervision of a scheduler.

When this function call is made, core 0 initializes all necessary state for all cores and starts them. Every core maintains its own state and runqueue – there is no centralized queue of tasks.

When each core starts, it is responsible for scheduling the idle task on itself. When that is done, it starts a core-local timer that will regularly raise an interrupt to invoke the scheduler. That is, the scheduler grants equal-length timeslices to all tasks, but varies the number of timeslices according to task priority. Core 0 has an additional requirement: it has to turn its own execution of `main()` into a schedulable task so that it can be preempted and scheduled like all other tasks in the system.

[1]There are a few other entry points that will also initialize the scheduler. However, most of the time the entry point will be `pthread_create`.

Core 0 then waits for all cores to become ready. When they all are, it proceeds to schedule the function the user passed in to `pthread_create()`.

Each core maintains schedulable tasks in priority queues. Although the storage back-end is easily replaceable, the one currently used is based on binary heaps where tasks are prioritized according to the number of times they have been allowed to run before[2]. Tasks that are blocked for any reason are temporarily removed from its task heap until it is ready to continue execution, at which point it is put back into the heap.

All cores can request work from other cores if they become idle. A core is considered idle if it is executing a special idle-task, which simply executes an infinite while-loop. This task is just like any other task in terms of scheduling, except in two significant ways:

1. It is pinned to a specific core so it cannot be moved by work requesting from another core.
2. It is given a special priority 0 (all regular tasks are in the range [1,100]) that means it will never be selected if there are any other tasks available for scheduling.

In addition to per-core heaps of tasks, all tasks are put in a global hash table indexed by thread ID. Since the API functions often take a thread identifier as parameter to identify a task, this is used to quickly locate to internal structure used to store all relevant state.

Thread IDs are chosen in increasing order, starting at 16, which is what the `MAX_CPUS` macro expands to[3]. All thread identifiers less than 16 are considered *idle* tasks. When core i creates and assigns itself an idle task, it chooses the ID i.

Since the first non-idle task to be created is always the *main* task, this task will always be given the ID 16. Any IDs greater than that belongs to regular threads, created either by *main* or another thread.

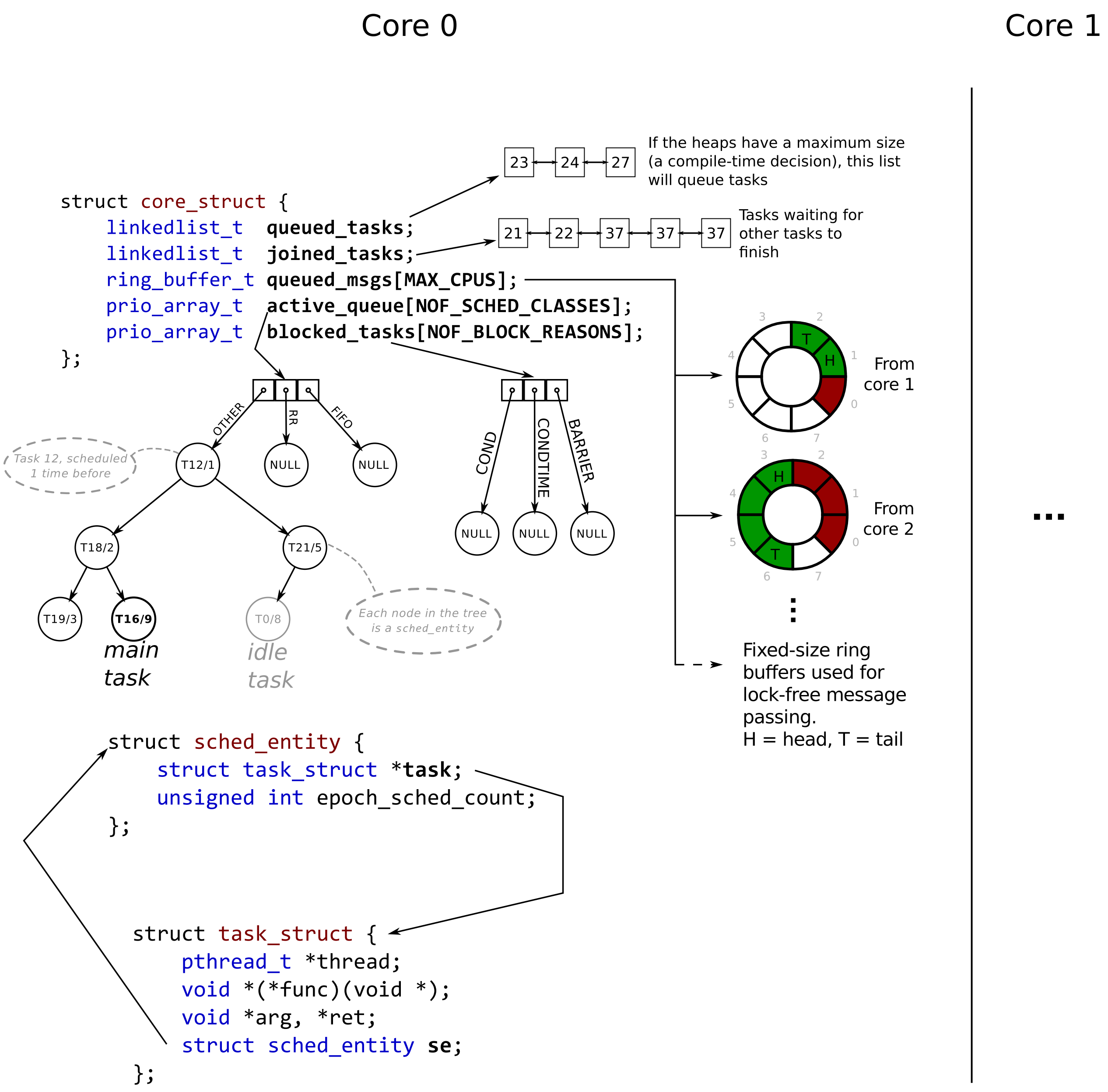
The overall design of the scheduler is shown in Figure 3.1.

3.2 Components In Detail

The following subsections will describe each component of the scheduler in greater detail. In some sections, alternative design choices that could have been made will be discussed.

[2]A half-truth: see Section 3.2.11 for details

[3]This is defined in *shmac.h*. 16 was chosen because the FPGA generally can't fit more than around 12 cores.


Core 0
Core 1
23 24 27
If the heaps have a maximum size (a compile-time decision), this list will queue tasks
21 22 37 37 37
Tasks waiting for other tasks to finish
OTHER
RR
FIFO
NULL
NULL
Task 12, scheduled 1 time before
T12/1
T18/2
T21/5
T19/3
T16/9
T0/8
main task
idle task
Each node in the tree is a sched_entity
COND
CONDTIME
BARRIER
NULL
NULL
NULL
3 2 4 1 5 0 6 7
T
H
From core 1
3 2 4 1 5 0 6 7
H
T
From core 2
Fixed-size ring buffers used for lock-free message passing.
H = head, T = tail

```
struct core_struct {
    linkedlist_t  queued_tasks;
    linkedlist_t  joined_tasks;
    ring_buffer_t queued_msgs[MAX_CPUS];
    prio_array_t  active_queue[NOF_SCHED_CLASSES];
    prio_array_t  blocked_tasks[NOF_BLOCK_REASONS];
};
```

```
struct sched_entity {
    struct task_struct *task;
    unsigned int epoch_sched_count;
};
```

```
struct task_struct {
    pthread_t *thread;
    void *(*func)(void *);
    void *arg, *ret;
    struct sched_entity se;
};
```

Figure 3.1: The overall design of the SHMAC scheduler. Each core stores its state in a `struct core_struct`. Most important among the members defined here are the three priority arrays used to hold tasks. Blocked tasks are also held in priority arrays. Inter-core messaging is lock-free and uses N^2 ring buffers in total. Tasks waiting to run (because the heap potentially has a maximum size) are lined up in a linked list. Tasks waiting for other tasks to finish are also kept in a linked list.

3.2.1 Per-Core State

Every core maintains its own state that is completely independent of the others. The only time the cores have to communicate is for work balancing purposes. It is a design goal that they should never lock each other's queues.

All per-core state is stored in a struct called `struct core_struct`. Its fields are:

state – *enum core_state*

An enum field that keeps track of which state the core is in. The possible values are:

1. `NOT_IN_USE` – the core is not to be used for scheduling purposes.
2. `NOT_READY` – the core has begun initialization, but is not ready. It should not be assigned any work at this point.
3. `IDLE` – The core is finished with initialization and can be given work.
4. `BUSY` – The core has started executing its first task, even if that might be the special *idle* task.

The states will evolve in monotonically increasing order – a `BUSY` core will never go back to being `IDLE`.

queued_tasks – *linkedlist_t*

A linked list of queued tasks. These are the core's "backlog" work items, tasks that haven't been put into a runqueue yet. Normally this list is always empty because the runqueue can grow arbitrarily large. This will only be used if the priority queue rejects new tasks. Tasks are dequeued whenever another task leaves the runqueue.

joined_tasks – *linkedlist_t*

A linked list of tasks that have tried to join another thread that is not yet finished. Tasks are then removed from the runqueue and put here. The scheduler will remove a task from `joined_tasks` when the thread to be joined finishes. It is then put back in the runqueue.

has_queued_msgs – *unsigned int*

A simple flag to indicate whether the core has pending messages it should process.

queued_msgs – *ring_buffer_t[MAX_CPUS]*

A fixed-size buffer for each possible core in the system (to avoid locking). Overflow is dropped, so the receiving core must process its messages regularly.

queue – *prio_array_t[3]*

A priority queue for each of the three supported scheduling classes. Each on uses its own ordering function, as described in section 3.2.11.

active_queue – *prio_array_t**

A pointer to the queue the current task belongs to.

current_task – *struct task_struct**
The task the core is currently executing.

blocked_tasks – *prio_array_t[3]*
Priority queue used to maintain tasks blocked by calling `pthread_cond_wait`, `pthread_cond_timedwait` or `pthread_barrier_wait`. Whenever a task needs to be awaken, the priority queue updates the task's location in it. The scheduling function peeks at the next task to decide if it should unblock tasks.

epoch_count – *unsigned int*
The number of times the scheduler has run on this core.

lock_object – *shmac_mutex_t*
The lock object used to protect the core. Any task wanting to modify a core's state should obtain this lock first.

3.2.2 The `task_struct` Struct

As the most important structure in the library, the `task_struct` deserves a close look.

The task object maintains all state that is required to keep track of execution context and scheduling properties, as well as some useful statistical fields. The fields are:

state – *enum task_state*
The task's state. Can be one of the following:

1. `QUEUED` – The task object has been created, but has not been assigned to a core yet.
2. `RUNNING` – The task has been assigned to a core and put into the runqueue. It might not have actually been granted a timeslice yet, though.
3. `BLOCKED` – The task has been removed (temporarily) from the runqueue because it called a blocking function.
4. `FINISHED` – The task has finished, but the scheduler has not started the reaping process yet. The task is still not joinable in this state.
5. `DEQUEUED` – When the scheduler finds a task in the `FINISHED` state, it removes it from the runqueue and changes its state to `DEQUEUED`. The task is now joinable.
6. `REAPED` – When a joinable (not created in detached mode) thread is joined, its state is set to `REAPED` before calling `free()` on it. After this, the task should never be referenced again.
7. `CANCEL_PENDING` – Some other task has called `pthread_cancel` on this task. It will be canceled the next time the scheduler runs.

8. `CANCELED` – The scheduler makes the task jump to a cleanup routine. It still needs to be joined for it to be freed and removed from the global task tree, unless it was created with `PTHREAD_CREATE_DETACHED`.

cpu_no – *int*
The core number the task is assigned to run on.

thread – *pthread_t**
The `pthread_t*` the user passed to `pthread_create`. This value is used to identify tasks externally.

attrs – *pthread_attr_t*
A copy of the attributes passed to `pthread_create`, or default attributes if `NULL` was passed.

stack_free – *int*
Set to 1 if the stack should be freed be the library when a task finishes. If the user allocated the stack before calling `pthread_create`, this will be 0.

task_free – *int*
Should the task object be freed when it finishes? The only tasks that should *not* be freed are the idle tasks for all cores but core 0, because these were allocated on the stack.

task – *void* (*)(void*)*
The function the user specified to `pthread_create`.

arg – *void**
The argument the user specified to `pthread_create`.

ret – *void**
A location for storing the return value of the function `task`.

waits_for – *struct task_struct**
Set when a thread calls `pthread_join`. It is used to detect possible deadlocks as well as indicate to the scheduler which task we have to wait for to finish.

join_ll_item – *linkedlist_item_t**
The linked-list node that is inserted into the list of blocked task if this task should block. This is allocated here to avoid calling `malloc()` in the interrupt handler.

pinned – *int*
A pinned task cannot be moved to another core through work requesting. The idle tasks are pinned.

blockable – *int*
This needs to be set to 1 for a task to be removed from the runqueue. This is used to prevent the scheduler from preempting tasks when in an inconsistent state.

cond_waiting – *int*
: Set by a thread that called `pthread_cond_wait`. The scheduler will detect this and block the task (if it is `blockable`).

cond_wait_timeout – *unsigned int*
: Set by a thread that called `pthread_cond_timedwait`. The scheduler will detect this and block the task (if it is `blockable`). The task will be placed in a different priority queue than the tasks that block indefinitely.

barrier_waiting – *int*
: Set to 1 whenever a task is blocked at a `pthread_barrier_wait` call.

join_blocking – *int*
: Set to 1 whenever a task tries to join a task that hasn't finished yet. The scheduler will remove the task from the runqueue until the task-to-be-joined (found in `waits_for`) has finished.

cancel_state – *int*
: Decides whether a task is cancelable. A call to `pthread_cancel` for a non-cancelable task is a no-op.

cancel_type – *int*
: Is usually used to determine how the task is to be canceled; either asynchronously or at a cancellation point. We do not support cancellation points, so this field is never used. All cancellation is asynchronous, which we take to mean "*the next time the scheduler is ran*".

register_set – *irq_regs_t*
: Stores the register values of the task at the time the scheduler is run. Used to restore execution context the next time the task is selected by the scheduler.

cleanup_handlers – *linkedlist_t*
: Used to hold cleanup functions specified by the user. These are called when a task finishes, no matter how it finishes.

epoch_count – *unsigned int*
: The total number of times this task has been selected by the scheduler to run. Not all epochs are equally long, as tasks can decide to explicitly invoke the scheduler. This is done in a few cases when the task would otherwise busy-wait.

se – *struct sched_entity**
: A struct that contains all scheduling-related information. This is disjoint from the `task_struct` struct to achieve looser coupling, thus simplifying the work of adding new data storage backends besides the binary-heap.

3.2.3 The `sched_entity` Struct

This struct keeps track of anything the scheduler might want to use to prioritize tasks. The exact definition depends on the data storage used, which is a rather loosely coupled component of the scheduler. For binary heaps, the structure looks as follows:

task – *struct task_struct**
A pointer back to the task this entity represents.

epoch_sched_count – *unsigned int*
A value similar to `epoch_count` in the task struct, albeit with one important difference: this one does not necessarily start counting from zero. This is explained in Section 3.2.11.

vruntime – *unsigned int*
The total time the task has been allowed to run, measured in clock ticks.

created_time – *unsigned int*
The time the task struct was created.

start – *unsigned int*
The time the task was first scheduled.

last_start – *unsigned int*
Timestamp representing when the task was last started.

3.2.4 Priority Queues

To keep track of schedulable tasks, three priority queues are used – one for each scheduling class. The three supported classes are: SCHED_OTHER, SCHED_RR (round-robin), SCHED_FIFO (First in, first out). As previously stated, the data structure used is interchangeable as long as supports the required operations. The binary heap is primarily used in this scheduler.

The semantics of the three scheduling classes are as follows:

SCHED_OTHER
This is the normal (default) scheduling class. Tasks are scheduled according to their priority level among other SCHED_OTHER tasks. However, both SCHED_RR and SCHED_FIFO will preempt these tasks.

SCHED_RR
Will preempt any SCHED_OTHER task and be preempted by any SCHED_FIFO task. Tasks are scheduled in round-robin fashion, in equal-length timeslices (as all tasks are with this scheduler). Priorities do not affect these tasks at all.

SCHED_FIFO

Will preempt any tasks from the other two scheduling classes. A SCHED_FIFO task can only be preempted by a higher-priority SCHED_FIFO task. If no such task is scheduled, this task runs to completion without being preempted.

The current task to execute is found by popping the highest-prioritized heap. That is, FIFO > RR > OTHER. When a task is to be preempted, it is pushed back into its priority queue before popping the next task, possibly from a different queue[4].

3.2.5 Global PID Hash

Like the O(n) scheduler had its pidhash array to enable quick lookups by PID, this scheduler keeps a similar structure for the same reason. This is widely used by the API functions which take a `pthread_t` as argument.

The O(n) scheduler's pidhash table is a fixed-size array which handles collisions by chaining. The SHMAC scheduler uses the same – an array consisting of 16 linked lists. However, this was a fairly recent choice: originally, a red-black tree was used.

A red-black tree has the benefit of performing lookups in $O(\log n)$, while the hash array requires a full search, requiring $O(n)$. However, insertions are $O(1)$ with the hash array. In practice, however, it was suspected that hash tables would be faster in spite of its worse asymptotic runtime.

To make a final decision between the two, some simple benchmarking was performed. The results are shown in Figure 3.2. As the results show, insertions are faster (although the logarithmic scales somewhat hides this) by around 25-40% at 8-32 tasks and approaching 50% with hundreds of tasks. The conclusion for searches is also clear: for a reasonable number of tasks, the hash table is significantly faster.

Another approach based on hashing was also considered, one that would insert in $O(\log n)$ time: once we reach a certain load factor, allocate a new, larger hash table. Make a note of the PID that caused this, so that lookups could efficiently be partitioned between the old and new hash table. This scheme works well since we operate with monotonic PIDs. This extension was however not deemed necessary, as the fixed-size hash table should be fast enough for all reasonable use cases.

[4]If the next task to be run is from the same queue as the current one, a combined push/pop is performed instead, which is faster.

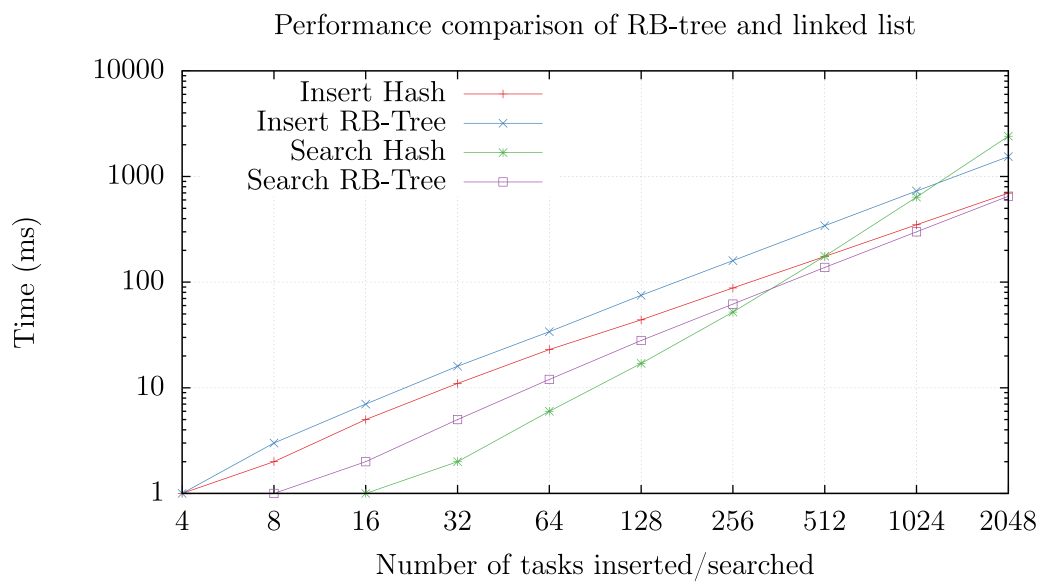


Figure 3.2: A comparison of red-black trees and hash tables with chaining.

3.2.6 Task Blocking

In addition, there are three priority queues used to keep track of blocked tasks. Tasks can block when they call `pthread_cond_wait`, `pthread_cond_timedwait` or `pthread_barrier_wait`. When tasks are blocked, they are removed from their normal priority queue and put into one of these three instead. On every call to the scheduler, the root element of every blocking queue is checked to see if it should be rescheduled.

If a task is to be woken up, the following operations are performed (in user mode, not by the scheduler itself):

1. The core in question is locked (thus preventing the scheduler from interrupting the process).
2. The blocking queue is scanned until the task is found[5].
3. The task has its `blocking` flag updated and the task is moved towards the head of the priority queue.

The next time the scheduler runs on the core to which the task belongs, the task will be woken up and put back into the normal runqueue.

[5]An unfortunate O(n) operation. This implementation should be optimized

3.2.7 Task Joining

Task joining works somewhat differently: There is no priority queue onto which the task is pushed; rather it is added to a linked list. A task A calling `pthread_join` on a task B typically goes through the following steps:

1. If B's PID is higher than that of the most recently started thread, A will busy-wait until the task exists[6].
2. If B is running, queued or blocked, A will suspend itself. This is done by setting the `task_joining` flag on itself and invoking the scheduler. The scheduler will then remove A from the runqueue along with a reference to B.
3. Every call to the scheduler will iterate through its list of (A, B) pairs. If task B has moved to state `DEQUEUED`, task A will be immediately granted a timeslot.
4. When A resumes, it will collect the return value of B and call `free_task(B)`. This removes B from the global task queue and frees up all memory allocated by B.

Once again, the scheduler will have to perform an $O(n)$ operation: scanning the list of join-blocked tasks. In practice, the expected number of items is low. For most workloads, it will be zero or one. Certain workloads, however, could have to scan through tens of tasks – still a fairly cheap operation.

Two other implementations were considered for task joining:

1. Use a priority queue like, for instance, `pthread_cond_wait` does.
2. Add a backward-reference to task A on task B so task B can wake task A up when it exits.

The first approach was in some ways the simplest, as we could simply reuse the structures that were already in place to support this. However, it would have required different locking semantics than the other blocking queues. This would require inter-core locking when a task was to be woken up. This was deemed too costly.

The second approach has the advantage of being both very simple and elegant. However, it still requires us to lock another core when adding the reference, in case task B finishes while we're in the process of blocking task A. In contrast to the first approach, the looking period would be constant.

[6]The standard does not dictate any particular behavior in this case. This implementation chooses to wait for the task instead of failing immediately.

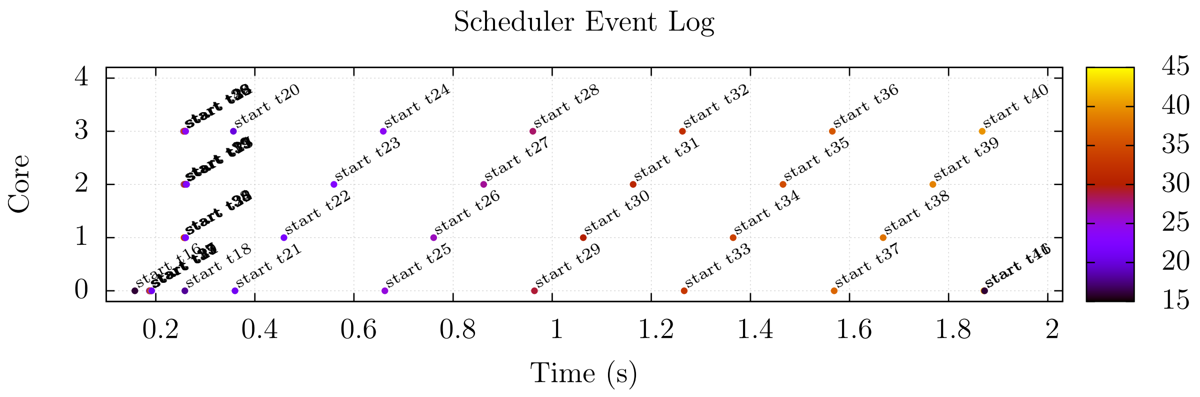


Figure 3.3: A program with 25 threads (plus the main thread) each waiting for one another. Each dot represents a scheduler event, labeled according to type. Each core has its own "timeline" parallel to the X-axis, which represents time. This particular plot shows the (relatively) long time spent joining tasks across cores.

The implementation of the second approach would be simple: replace the `waits_for` field in the task struct with a `waited_by` field. Add some checks to the task finalization process to restore the task found in this field, if non-null. This would have the side-effect of simultaneously (potentially, at least) moving the task to another core.

The relative performance of this second approach to the one currently used was never examined. For our intents and purposes, the choice seemed arbitrary: both solutions offer more than acceptable performance. However, the second approach has an elegance to it that makes it appealing.

Another point worth noting here is related to the explicit invocation of the scheduler. When task A realizes that task B is not finished, it yields to the next task immediately. Once a task finishes, it does the same. This leaves us with a very effective scheme for intra-core task joining: no time is unnecessarily spent busy-waiting.

There is no way to interrupt another core, so inter-core joining does not enjoy the same benefits. Here task A must wait for the timer interrupts to realize that task B has finished.

Throughout the rest of this thesis, a specific type of event plot will be used to illustrate the choices the scheduler makes. One such plot is shown in Figure 3.3. Each dot in the plot represents an event related to a specific task. The type of event and corresponding task ID is shown in a label next to the dot. Events are colored by task ID, although the coloring is not consistent between plots. The plots are generally filtered by event type to keep the them easily readable. Time is represented linearly along the X-axis while the Y-axis shows different cores. Events at $y = 0$ has events for core 0, $y = 1$ for core 1, and so on. Since tasks can migrate between cores, they can move between different "y-lines" in the plots as well.

As well as introducing the plot figures, Figure 3.3 shows behavior related to task joining. It illustrates a simple recursive summing scheme, which heavily depends on fast task joining. Explicitly created tasks are given IDs from 17 through 40. Each one joins its predecessor, except 17, which immediately returns a constant value (one). All other threads return its predecessor's return value plus one. The main task (task 16) joins the last task (task 40).

In the 0.2-0.3 second time range, tasks are rapidly created and started. Cores 1 through 3 have synchronized timers whereas core 0 is slightly ahead of the others. When each core starts scheduling the tasks it has been assigned, they are all suspended quickly once they reach their respective `pthread_join` calls. One by one, they are awakened as their predecessors finish.

Because of the way tasks are assigned (to the least busy core), tasks on core C always join a task on core $(C - 1) \mod 4$. With a timer rate of $10Hz$, this takes $100ms$ as long as the scheduler timers are synchronized. Because of lucky timing differences, core 0 can join tasks on core 3 in approximately $3ms$.

3.2.8 Work Requesting

To balance tasks between cores, there are two basic approaches that can be taken: work requesting or work stealing.

This scheduler chose work requesting. The main reason for this choice was to avoid locking. In order for core 1 to steal tasks from core 2, core 1 would have to lock core 2's runqueue. Since the scheduler is primarily triggered by synchronized timer events, core 2 would in all likelihood already have locked its own runqueue. In addition, there could be several other cores waiting for the same lock – only to find that there might not be work there to be stolen.

For that reason, the more elegant solution of work requesting was chosen. This works the following way:

1. When a core is about to schedule the idle task, it sends out a work request to another core[7]. Each core has N message queues, N being the maximum number of CPUs in the system. It also sets a flag to signal to the other core that it has pending messages. This is purely an optimization to avoid having to scan all messages queues every time.
2. On every call, the scheduler function checks to see if the flag has been set. If so, it scans all N message queues (implemented using ring buffers, overflow messages are simply dropped). If it has a work request, it checks to see if it has more than two tasks running. If so, one of them is removed from the runqueue and sent in a response to the requesting core.
3. The sending core will now see a work response message when the scheduler is invoked and add it to its own runqueue.

A few things are of importance here:

[7]Specifically, it is sent to core `idle->epoch_count % CPU_COUNT`. This ensures that each core will ask all other cores in a round-robin fashion as the epoch counter is incremented.

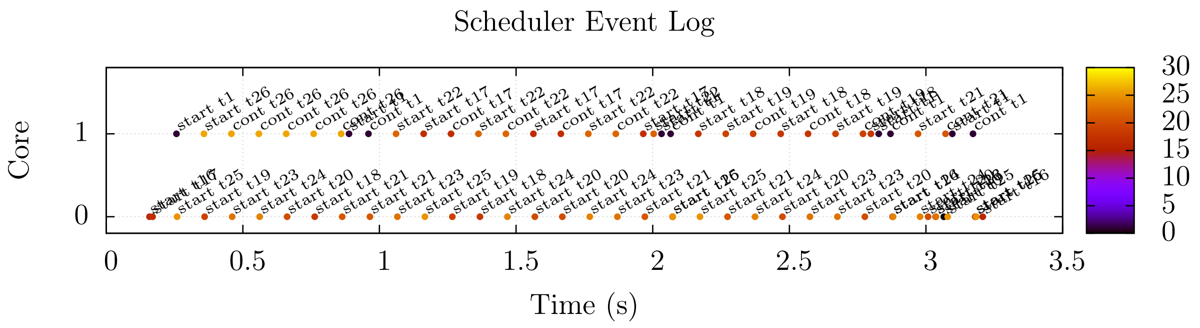


Figure 3.4: Core 1 requesting work from core 0. All tasks are initially given to core 0 (for illustrative purposes) and core 1 initially starts executing the idle task.

1. A work response is a binding agreement that the receiving core will execute the task it is given, even if it receives tasks from several cores. This to simplify the semantics of guaranteeing that no tasks are lost
2. Messages could potentially be lost. Each core can only receive N messages from any one core. This should not happen as long as the sending core doesn't "spam" any one core, and all cores process their message queues regularly.
3. A core could potentially idle for two timeslices waiting for a response to come.

The work requesting process is shown in Figure 3.4. As the figure illustrates, a core can potentially end up with more than one task. In this case, core 1 had time to send out two work requests to core 0, both of which were responded to.

3.2.9 Core Bootstrapping

The SHMAC starts with a single core active: core 0. This core has to manually trigger the other cores to start by setting a memory mapped register to 1: `*SYS_READY = 1`. When core 0 starts, it is not executing a schedulable task: there is no timer-triggered scheduler active on the core.

All the bootstrapping is done when core 0 first makes a call to `pthread_create`. The entire function can be seen in Listing 3.1.

```
int pthread_create(pthread_t *t, const pthread_attr_t *attrs,
                   void *(*func)(void *), void *arg)
    if (!boot_pthreads_called()) {
        boot_pthreads();
    }

    sched_entity_t *se = task_create(t, attrs, func, arg, 1, 1, 0,
            PTHREAD_CANCEL_ENABLE);
    if (se == NULL) {
        errno = ENOMEM;
```

```c
        return ENOMEM;
    }

    if (se->task != NULL && task_enqueue(se)) {
        return 0;
    }

    return ENOMEM; // probably
}
```

Listing 3.1: The `pthread_create` function.

The first if-block is the interesting part: it is responsible for initializing the rest of the cores the first time the function is called. The entire `boot_pthreads` function looks like this:

```c
void boot_pthreads()
{
    assert(!boot_pthreads_called());

    sched_init();
    start_cores();
    schedule_me();
    wait_for_cores();
}
```

Listing 3.2: The four steps involved in bootstrapping the threading library.

These four steps bootstrap all cores, set up necessary per-core state and wait for it all to get ready. In more detail, this is what happens:

1. `sched_init()` initializes global task state and calls `core_init()` for each core. This function sets up all core-specific state, such as locks and runqueues.
2. `start_cores()` is the one that actually sets `*SYS_READY = 1`. It also sets a global variable `_main_function` to the address each core should start executing from. There is a check in `crt0.S` that will branch to this address if it is non-zero, otherwise `main()` is executed. This is where the rest of the core's execution deviates from the first core's.
3. This branch target schedules the idle task on the local core. It then starts the local timer, which will periodically invoke the scheduler. It also sets the core state to *ready*. If the calling core is not 0, the function enters an infinite loop at this point, waiting for the scheduler to change context. For core 0, the function returns normally.
4. Subsequently, `schedule_me()` is called. This is a special function that makes the current execution context a schedulable task. In this case, the context is core 0's execution of `main()`. This step is required to put core 0 in the same state as the other cores; ready to execute tasks assigned by the scheduler.
5. When core 0 returns from `schedule_me()` as a scheduler controlled task, it calls `wait_for_cores()`, which blocks until the other cores are ready. They are typically already ready, so this waiting period is short.

3.2.10 Event Logging

A simple event logging mechanism was included to help analyze the internals of the scheduler. If compiled with `-DENABLE_EVENT_LOGGING`, the library will register events such as thread creation, preemption, joining and exiting. A call to `dump_log()` (or `dump_log_mem()`, which stores it to memory) will print a chronologically merged view of all logs.

To get as precise event timestamps as possible (and to avoid possible deadlocks), the library is lock-free. Each core has two event logs: one for user mode and one for privileged mode (used in IRQ/supervisor mode). When `dump_log()` is called, the logs are merged according to timestamps.

Each event contains the following fields:

1. A timestamp, measured in ticks.
2. The core the event was triggered on.
3. The actual event, represented both as an integer and a string.
4. Optional: the thread in-memory address and ID that was associated with the event.
5. Optional: event-specific parameters.

The output is dumped as comma-separated values which can easily be plotted, for instance with *Gnuplot*[8]. An example output is shown in Listing 3.3.

Since there are no easy way to debug on SHMAC other than using `printf` statements, the event logging mechanism was the foremost source of data used for debugging purposes. `printf` is problematic for several reasons:

1. If used lock-free, the outputs from different cores will collide and result in unreadable output.
2. If locking is added to prevent that problem, deadlocks can easily occur if printing is used in both user mode and interrupt mode, which it typically is.
3. `printf` statements add substantial overhead, both because of locking and because outputting text is relatively slow. If the cause of a bug is timing-related, this is particularly unfortunate. Printing a single log statement like the ones shown in Listing 3.3 takes around $1ms$. Logging the same data to memory takes around 25 to $35\mu s$, depending on the number of optional parameters.

These shortcomings were the reason a logging mechanism was implemented in the first place.

[8]http://www.gnuplot.info/

```
/* The columns are:
 *  - timestamp (in ticks; there are 60e6/1024 ticks per second)
 *  - core ID (all cores are numbered from 0 through N-1)
 *  - event ID (a unique integer ID for each event type)
 *  - event string (a short, textual representation meant for plotting)
 *  - task address (the in-memory address of the task)
 *  - task ID (the thread ID (pthread_t) as exposed to the user)

 * Additionally, each event can specify extra integer parameters.
 */

114110,3,12,irqs,0x1f9ba14,3    // irq start on core 3
114120,3,3,cont,0x1f9ba14,3     // task 3 (idle) continues on core 3
114123,3,16,irqf,0x1f9ba14,3    // irq finishes on core 3
114179,1,12,irqs,0x1fdba14,1    // irq start on core 1
114190,1,18,mqreqr,0x0,0,0      // core 1 received request from core 0
114206,1,18,mqreqr,0x0,0,2      // core 1 received request form core 2
114228,1,3,cont,0x1fdba14,1     // task 1 (idle) continues on core 1
114231,1,16,irqf,0x1fdba14,1    // irq finishes on core 1
114239,0,12,irqs,0x27f08,0      // irq starts on core 0
114253,0,18,mqreqr,0x0,0,3      // core 0 received request from core 3
114276,0,4,stop,0x27f08,0       // core 0 stops the idle task
114279,0,11,jnviv,0xb4080,16,56 // core 0 wakes task 16 because 56 is finished
114286,0,2,start,0xb4080,16     // core 0 starts task 16
114289,0,16,irqf,0xb4080,16     // irq finishes on core 0
114293,0,9,jnd,0xb4080,16,56    // task 16 joined task 56 (user-space event)
```

Listing 3.3: A sample output from the event logger. The comments are not part of the output. The event names are abbreviated so they can easily fit in an event plot.

3.2.11 Task Prioritization

In order to decide which task to schedule, the binary heap needs a way to establish a partial ordering of tasks. This is done differently for each scheduling class. This is a fairly straightforward process, at least for the RR and FIFO classes.

The SCHED_OTHER prioritization function is the most complex. It compares two schedulable entities (s1, s2) according to the following rules:

1. If s1 or s2 represents the *idle* task, select the other one.
2. Otherwise, compare with respect to the following 'goodness' function:
   `goodness(s) = s->epoch_sched_count / s->task->prio`.
   Return the one with the lowest value.

Where `s->epoch_sched_count` is (essentially) the number of times the task has been scheduled. It does, however, deviate from the true count in one significant way: it is initialized to the value of the calling thread when the task is created. The following case illustrates the purpose of this slight modification:

Imagine a system with a single CPU and two threads of equal priority. The first thread is started at $t = 0$, while the second thread is started at $t = 5s$. With a timer frequency of, say, 10, this first task will have been re-chosen by scheduler 49 times already. The second task will have a count of zero and will thus have to "catch up" with the first thread. The first thread will then wait five seconds before it is scheduled again.

This is obviously not what we want. Instead, each task's internal counter should optimally be reset whenever a new task starts. Or equivalently, the new task can start its epoch counter roughly equal to that of the currently running task (and adjusted to account for possibly different task priorities).

The difference between these two cases are shown in Figure 3.5 and Figure 3.6. Both plots illustrate what happens when four tasks are scheduled (task 17 through 20). Tasks are created with fixed intervals and each perform a fixed amount of work. Figure 3.5 shows how task 18 is granted the same amount of time task 17 has already run for. This is obviously not an acceptable trait in a scheduler.

Figure 3.6 shows the result of copying the counter from the active task each time a new task is assigned to the same core. Once task 18 is spawned, it is immediately sharing its CPU time evenly with task 17.

The comparison for SCHED_RR tasks is simpler: simply chose the one that has been scheduled the fewest number of times. This is identical to the previous function, minus the "idle" task special case and dividing by `p->prio`.

The comparison for SCHED_FIFO simply orders tasks (of equal priority) by when they were

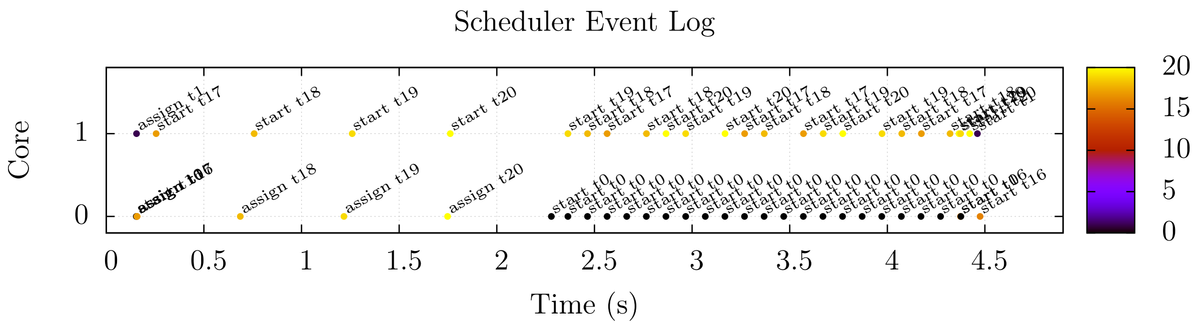


Figure 3.5: Scheduling tasks by selecting the one that has been allowed to run the fewest number of times. This allows task 18, 19 and 20 to run for approximately 0.5 seconds each before better balancing is achieved.

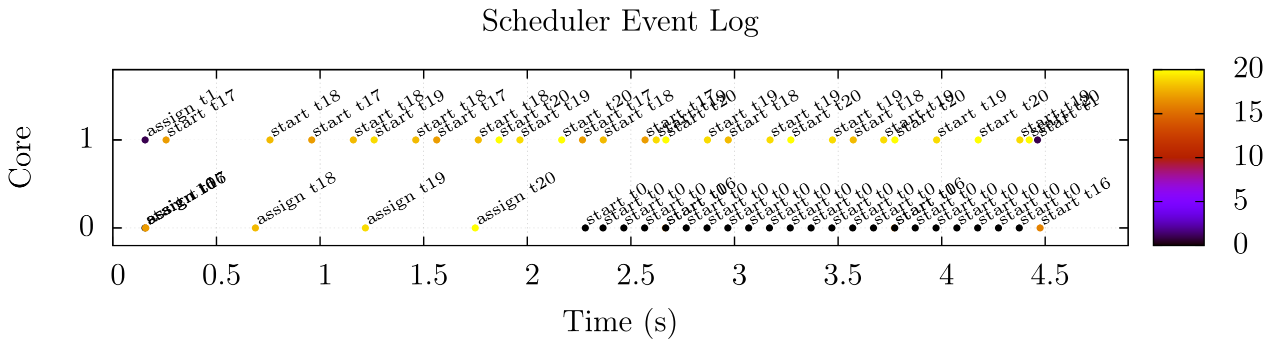


Figure 3.6: Choosing task to run by using the modified `task_sched_count` field, instead of the true count as in 3.5. As a result, tasks are perfectly balanced immediately.

created
(`p->created_time`), effectively turning the heap into a FIFO queue.

The three ordering functions can be found in Appendix D.

3.3 Pthreads API

The previous sections described the inner workings of the scheduler. It is now time to take a look at the external interface used to interact with the scheduler: the Pthreads API.

There was no fixed goal regarding function implementation coverage before starting this work. Obviously as much as possible should be implemented, but 100% coverage was never thought of as a feasible goal.

This stems both from the fact that architectural limitations would render some functions impossible to implement[9] and the limited time available on this project. However, many Pthreads functions are infrequently used – implementing the top 10-20 functions alone would suffice for a very high degree of compatibility.

The Newlib header file `pthread.h` consist of function definitions from draft 10 of the POSIX.1c standard (IEEE Std. 1003.1c-1995). Two functions are present in the standard but not in the header file: `pthread_kill` and `pthread_sigmask`. Thread signaling is thus not included in the implementation.

So while not all functions were implemented, a large enough subset to support most applications is considered to be fully functional. Table 3.1 shows all functions mentioned in Newlib's `pthread.h` header file, their implementation status and possibly a related comment. A status of "Ok" means that the function should be fully POSIX compliant. A status of "Ok*" means that the function exists, but should not be expected to behave in a useful manner. It is likely a dummy function that has no actual effect. This status is typically applied to functions that cannot be implemented due to either SHMAC or software platform limitations (no operating system).

Similarly, "Missing" means that the function is completely absent from the library and could probably be implemented. However, a status of "Missing*" indicates that a function is missing for a good reason.

[9]Without a complete operation system, it makes little sense to implement the functions that relate to processes. The lack of virtual memory and memory protection makes stack guard implementations too inefficient to implement.

Table 3.1: Implementation coverage of Newlib's `pthread.h` header file.

Function	**Impl. status**	**Comment**
pthread_atfork	Missing*	No processes, no forking
pthread_attr_destroy	Ok	
pthread_attr_getdetachstate	Ok	
pthread_attr_getguardsize	Ok*	No memory segments, dummy function
pthread_attr_getinheritsched	Ok	
pthread_attr_getschedparam	Ok	
pthread_attr_getschedpolicy	Ok	
pthread_attr_getscope	Ok*	No processes, no scope
pthread_attr_getstack	Ok	
pthread_attr_getstackaddr	Ok	
pthread_attr_getstacksize	Ok	
pthread_attr_init	Ok	
pthread_attr_setdetachstate	Ok	
pthread_attr_setguardsize	Ok*	No memory segments, dummy function
pthread_attr_setinheritsched	Ok	
pthread_attr_setschedparam	Ok	
pthread_attr_setschedpolicy	Ok	
pthread_attr_setscope	Ok*	No processes, no scope
pthread_attr_setstack	Ok	
pthread_attr_setstackaddr	Ok	
pthread_attr_setstacksize	Ok	
pthread_barrierattr_destroy	Ok	
pthread_barrierattr_getpshared	Ok*	No processes, no sharing
pthread_barrierattr_init	Ok	
pthread_barrierattr_setpshared	Ok*	No processes, no sharing
pthread_barrier_destroy	Ok	
pthread_barrier_init	Ok	
pthread_barrier_wait	Ok	
pthread_cancel	Ok	
pthread_cleanup_pop	Ok	
pthread_cleanup_push	Ok	
pthread_condattr_destroy	Ok	
pthread_condattr_getpshared	Ok*	No processes, no sharing.
pthread_condattr_init	Ok	
pthread_condattr_setpshared	Ok*	No proocesses, no sharing.
pthread_cond_broadcast	Ok	
pthread_cond_destroy	Ok	
pthread_cond_init	Ok	
pthread_cond_signal	Ok	
pthread_cond_timedwait	Ok	

Continued on next page

Table 3.1 – continued from previous page

Function	Impl. status	Comment
pthread_cond_wait	Ok	
pthread_create	Ok	
pthread_create	Ok	
pthread_detach	Ok	
pthread_equal	Ok	
pthread_exit	Ok	
pthread_getcpuclockid	Missing*	No `clockid_t` in Newlib.
pthread_getschedparam	Ok	
pthread_getspecific	Ok	
pthread_join	Ok	
pthread_key_create	Ok	
pthread_key_delete	Ok	
pthread_mutexattr_destroy	Ok	
pthread_mutexattr_getprioceiling	Ok*	
pthread_mutexattr_getprotocol	Ok*	
pthread_mutexattr_getpshared	Ok*	No processes, no sharing.
pthread_mutexattr_gettype	Ok*	
pthread_mutexattr_init	Ok	
pthread_mutexattr_setprioceiling	Ok*	
pthread_mutexattr_setprotocol	Ok*	
pthread_mutexattr_setpshared	Ok*	No processes, no sharing.
pthread_mutexattr_settype	Ok*	
pthread_mutex_destroy	Ok	
pthread_mutex_getprioceiling	Missing	
pthread_mutex_init	Ok	
pthread_mutex_lock	Ok	
pthread_mutex_setprioceiling	Missing	
pthread_mutex_timedlock	Ok	
pthread_mutex_trylock	Ok	
pthread_mutex_unlock	Ok	
pthread_once	Ok	
pthread_rwlockattr_destroy	Ok	
pthread_rwlockattr_getpshared	Ok*	No processes, no sharing.
pthread_rwlockattr_init	Ok	
pthread_rwlockattr_setpshared	Ok*	No processes, no sharing.
pthread_rwlock_destroy	Ok	
pthread_rwlock_init	Ok	
pthread_rwlock_rdlock	Ok	
pthread_rwlock_timedrdlock	Missing	No timeout support implemented.
pthread_rwlock_timedwrlock	Missing	No timeout support implemented.
pthread_rwlock_tryrdlock	Ok	
pthread_rwlock_trywrlock	Ok	

Continued on next page

Table 3.1 – continued from previous page

Function	Impl. status	Comment
pthread_rwlock_unlock	Ok	
pthread_rwlock_wrlock	Ok	
pthread_self	Ok	
pthread_setcancelstate	Ok	
pthread_setcanceltype	Ok	
pthread_setschedparam	Ok	
pthread_setspecific	Ok	
pthread_spin_destroy	Ok	Spinlocks are just wrappers
pthread_spin_init	Ok	for pthread_mutex_*.
pthread_spin_lock	Ok	Both are spin locks, there is no
pthread_spin_trylock	Ok	scheduler cooperation here.
pthread_spin_unlock	Ok	
pthread_testcancel	Ok*	All cancellation is asynchronous

Some functions that are not in Newlib's `pthread.h` were implemented. These functions are marked as "non-portable" by including the "_np" suffix to the function same, such as "pthread_setaffinity_np".

The non-portable functions included are:

pthread_delay_np
This is a simple sleep function that delays execution by the specified `struct timespec *interval`. Internally, it is a busy-wait loop.

pthread_num_processors_np
Returns the number of processors in the system.

Unfortunately, the most useful of the non-portable functions are missing: those that control task affinity. This is a useful mechanism to control which cores a task can be run on.

Internally, the scheduler supports task pinning, but not full affinity control. This is not implemented because Newlib does not provide the required type definition (`cpu_set_t`) and the related functions to modify CPU sets.

3.4 Project Structure

As a starting point for possible future work on the library, an overview of the source code is included. Figure B.1 shows the source code's placement in the overall SHMAC project structure. Figure 3.7 shows the folder structure of the project.

Folder	**SLOCCount**	**GNU wc**
src/	1,652	2,286
src/data/	828	1,425
src/sched/	262	419
src/external/	2,157	2,825
src/tests/	1,380	1,955
Total	**6,279**	**8,910**

Table 3.2: The amount of code per folder in the project.

The *Makefile* provides two important rules: *pthread.a* and *test*. The default rule is *pthread.a*, which builds the library as an archive. The *test* rule creates a test binary. This application can be run on SHMAC and will perform a series of unit tests.

bin/ contains a collection of scripts[10] that can be used to analyze scheduler logs. For instance, all event plots in this thesis is generated by *shmac_timeline*. There are other scripts that extract several other kinds of useful data as well.

src/ is the main source code folder. All files related to core scheduler features are placed directly in this folder, such as *boot.c*, *task.c* and *timer.c*. Header files are placed next to their corresponding *.c* files, not in a separate directory.

The *data/* sub-folder contains data structures commonly used by the scheduler. The complete list in shown in the figure.

sched/ contains all common header files used, definitions of the `task_t` and `core_t` structs and implementation of the heap backend. If one were to write a second backend, (say, one based on linked lists) one would add *linkedlist.{c,h}* here and add a few lines in *sched.h* to incorporate it.

external/ defines all functions found in `pthread.h`. Most of them are fairly short functions that interact with either the scheduler or the `task_t` struct in some way.

The *tests/* folder contains unit tests. These are logically grouped into source code files and each declare one top-level invocation function. *test.c* contains the code that calls each of these top-level unit test functions in turn. All tests are based on *assertions*, as defined in `assert.h`.

obj/ contains all object files generated by the compiler. It duplicates the folder structure from *src/*.

The total amount of code per folder (as counted by *SLOCCount*[11] and *GNU wc*[12]) is shown in Table 3.2.

[10] Mostly Perl, with one or two exceptions in Bash.
[11] `http://www.dwheeler.com/sloccount/`
[12] `https://www.gnu.org/software/coreutils/manual/html_node/wc-invocation.html`

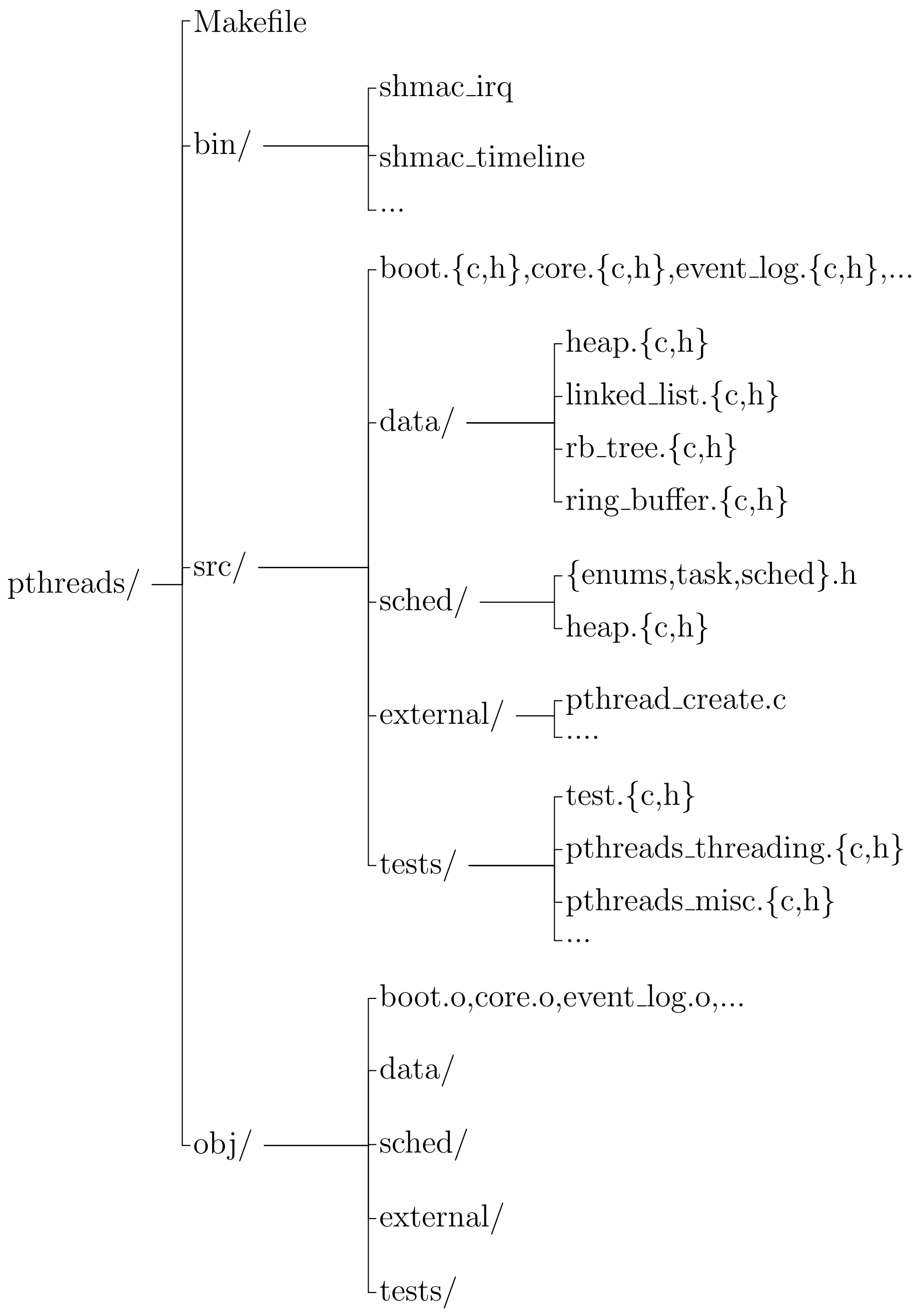


Figure 3.7: The code structure of the Pthreads library project.

Library component	**Unit test status**
Data structures	
Red-black trees	Very good
Binary heaps	Very good
Ring buffer	Very good
Linked lists	Very good
Core components	
Basic threading (create/join/exit/cancel/etc.)	Very good
Scheduling behavior (prioritization/fairness)	None
Thread-local storage (get-/setspecific)	Very good
Mutexes	Very good
Spin-locks	None
Condition variables	None
R/W-locks	None
Barriers	None

Table 3.3: Test coverage status by library component.

3.5 Test Coverage

There are some basic unit tests included in the project. These tests cover the internal data structures used, such as red-black trees and heaps, as well as the external Pthread API functions. The test coverage is unfortunately not complete. Table 3.3 shows test coverage status by library component.

In addition to writing unit tests from scratch, many were copied from the *POSIX threads for Embedded Systems* project. Some tests were excluded because they were either too similar to one another or because they went beyond the Pthread specification and tested behavior specific to that implementation.

As table 3.3 shows, scheduler behavior is generally not verified through unit testing. This area has been easier to confirm by extracting data from the event logs and verifying that the scheduler does what it is expected to. The scheduler has been thoroughly verified in this fashion.

The overall status of the library is that the core features are thought to be bug-free. More peripheral features, such as R/W-locks and barriers, are less extensively tested.

3.6 Known Issues

The goal of the library should of course be to be completely bug-free. While this is unfortunately hard to verify, there are currently no known bugs. Unit tests and general usage of the library for the purposes of porting existing applications have given reasonable confidence that the core features of the library behave as expected and perform well.

One serious bug was discovered, but as it seemed to be related to the hardware platform rather than the software, it will not be discussed here. This bug is presented in Chapter 6.

Chapter 4

The SHMAC Profiler

To get a clear picture of how well the scheduler performed, some kind of profiler was needed. The logging mechanism described in section 3.2.10 is good for examining program flow in a very coarse-grained fashion, but there are some limitations to it:

1. The timestamps included are not precise enough for general profiling.
2. The `log_event` function is too heavy to be called tens or hundreds of times per second.
3. To be used for profiling, the programmer would have to manually insert log statements at every point to be profiled.

For these reasons, a full-fledged profiler was written. Two approaches were considered:

1. Add support for GCC's `-pg` flag, which makes the application profile itself through a combination of instrumentation and sampling. The application will then record every call arc, but not function returns. The application will write its profile to a file called *gmon.out* to be read by GNU gprof.
2. Use GCC's `-finstrument-functions` flag to have GCC insert special function calls to every function: `__cyg_profile_func_enter` and `__cyg_profile_func_exit`. Use this to trace the call tree. The sampler can then increment a counter for the current node in the tree.

The first approach has the clear benefit of being able to utilize the existing profiler *gprof*. However, `-pg` only registers when a function starts, not when it exits. As such it does not have a precise trace of the program's execution. It uses sampling to gauge how much time a function takes, but it cannot tell several calls to the same function apart, and thus assumes that a function's execution time is independent of where it was called from.

It also requires that an instrumented version of libc is available, called `libc_p`. In addition, it requires a modified version of the startup script, `gcrt0`. Finally, the profile is written to disk by a exit routine installed by `atexit()`.

The requirements would then be:

1. Modify Newlib to compile the addition *libc*_p archive.
2. Modify our current startup script and compile it to *gcrt0*.
3. Implement Newlib's I/O function stubs to write to an in-memory file which we can then extract afterwards.

It was decided that it would be easier to write a minimal profiler using `-finstrument-functions` and a simple program to produce the profile output in a human-readable fashion. This solution is also more flexible since the user has complete control over which functions are instrumented. On the other hand, the required instrumentation will be more expensive than the one required for `-pg`.

4.1 Profiler Implementation

The profiler uses the special function calls inserted by GCC to trace program execution. This data is used to create an in-memory call tree.

Every distinct function[1] is represented by its own node in the call tree. Every node stores the following:

- A reference to its parent node
- The address of the function this node represents
- The address this node (function) was called from
- A list of pointers to child function nodes
- Two fields used for statistical purposes: call arc traversal count and PC (Program Counter) sample counter

The current state of the tracing is stored by keeping a reference to the `current` node in the tree. This reference will be updated on every call to the enter/exit routines. On enter, it will look up a call in the *children* list. If a match is found, the `current` pointer is updated

[1]Two function nodes are indistinct if and only if their respective functions have the same address and were reached through the exact same call path. A call arc is uniquely identified by its (lr, pc) tuple.

to point to that node. If no match is found, an entry is created before `current` is updated. On exit, it will use the parent reference to move up one level.

An interesting design decision is thus how child references are stored. Every call to the enter function will need to do a lookup in this list to check if a given call arc has been traversed before. This is an important decision with regards to performance.

It was decided that hashing by function call address was likely the best option. However, since reallocating memory in the enter/exit functions can cause spurious delays in the instrumentation functions, a solution that didn't rely on growing memory at profile-time was needed. The solution became a linked list of fixed-size blocks, each having room for 16 function calls. The hashing logic thus became:

1. Check at index $hash(lr, pc)$ in the first block. If a match is found, use this index.
2. If no match is found, check index $hash(lr, pc)$ in the next block (if it exists). If a match is found, return this index and a reference to the block. If not, keep following the linked list of blocks and checking index $hash(lr, pc)$ for a match.
3. If the end is reached without finding a match at index $hash(lr, pc)$, default to scanning each block sequentially, but starting at index $hash(lr, pc) + 1$ and stopping at index $hash(lr, pc) - 1$ (accounting for wrap-around to index 0, of course).

This gives us an open-addressing scheme that allows for functions to have a virtually unlimited number of function calls in them. The hashing function is:

$$hash(lr, pc) \quad = \quad (lr/4) \mod \text{CHILD\_BLOCK\_SIZE}$$

We obviously expect the compiler to optimize this into right shifts and bitwise AND, since SHMAC executes divisions (and modulo) in software.

Also note that we are not using the program counter in the hashing. For normal function calls, this is not important since the link register uniquely identifies it. However, this is not true in the case of function pointers. The profiler does look at both the LR and PC values to check for uniqueness so indirect function calls would not be coalesced by accident. However, extensive use of indirect jumps from the same function node could potentially lead to slow hash table lookups. In the interest of keeping the common case as fast as possible, PC was not included in the hash function.

The actual data structures are shown in Listing 4.1. The constant `CHILD_BLOCK_SIZE` is set to 16. Figure 4.1 illustrates how this works for a block size of 2.

Buffers of both these structs are allocated and zeroed out when the profiler is initialized to avoid the cost of doing so mid-profiling. The rather arbitrary constant sizes chosen for these two struct buffers are 8,192 and 16,384. This means that the call tree can contain no more than 8,192 functions, and, on average, each one of these can take no more than two

```c
struct call_list {
   struct function_node *calls[CHILD_BLOCK_SIZE];
   struct call_list *next_block;
   struct call_list *first_block;   // same as function_node.calls
};

struct function_node {
    struct function_node *parent;
    unsigned int sample_count;       // incremented by irq
    unsigned int call_count;         // incremented in 'enter' function
    void *pc;
    void *lr;

    struct call_list *calls;
};
```

Listing 4.1: The data structures used to represent an in-memory call tree. Each `function_node` can contain any number of `call_lists`, which are chained in a linked list. All structures are pre-allocated.

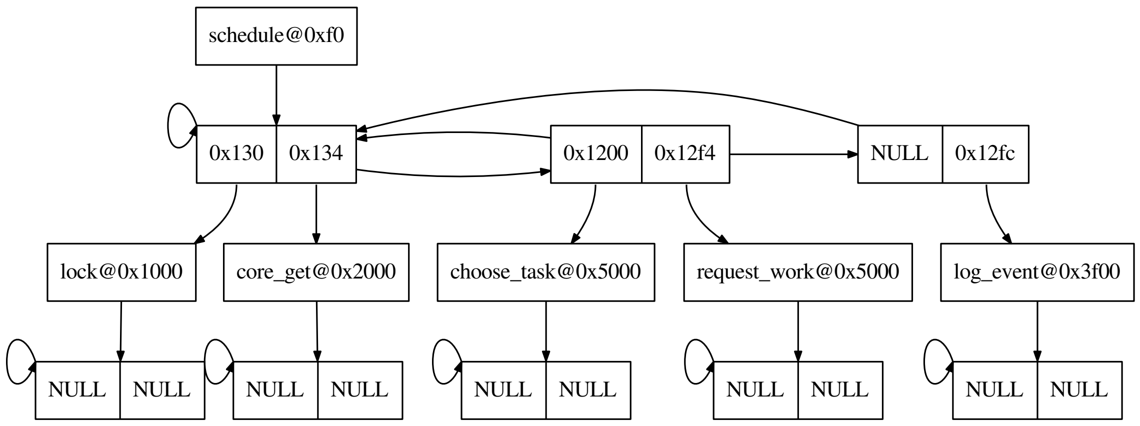


Figure 4.1: A visualization of how functions and call lists, as defined in Listing 4.1, relate to one another. This example shows the `schedule` function calling five other functions.

```
(
  calltree=(
    pc=0,lr=0,samples=491,calls=0,
    children=[
      (
        pc=8dd8,lr=92c4,samples=0,calls=7,
        children=[
         (pc=2ca4,lr=8e08,samples=0,calls=7,children=[])
        ]
      ),(...)
    ]
  ),
  histogram=(
    1198,1360,13dc,145a0,1748,1718,
    16fc,1a10,1ab0,1a80,1d40,1db4,
    1d20,20e0,20e0,2368,23c8
  )
)
```

Listing 4.2: An example of the kind of output the profiler can generate.

`call_lists` (32 function calls). For these constants, the profiler will require 1,312kB of memory plus whatever it estimates for the histogram, which is typically anywhere from 10kB to 1MB.

In addition to the tree, every PC sample is stored in a histogram. When the profile output is finalized, the histogram is scanned and addresses with non-zero sample counts are printed (possibly repeatedly). Since GCC does not provide us with any symbols to determine the exact size of the `.text` segment, we use `__data_start` (and the assumption that `.text` starts at address zero) to estimate the size of the `.text`. We allocate a buffer of this size to serve as our histogram; the PC register is used directly to index it during a timer interrupt event.

Just like its in-memory representation, the profiler produces a tree-like output format. Instead of using a binary format like *gprof*, the output is human-readable. An example is shown in Listing 4.2. The whitespace is added here for clarity.

4.2 Parsing The Output

A simple Python program (named *shmac_prof*) was written to generate a call tree and print some useful statistics. It currently supports the following outputs:

- Printing the critical path along with time spent in each function.
- Printing the full call tree, annotated will timing information.

- Printing a flat list of all functions and how much time was spent in each one.

In addition, some functionality exists to print only the interesting parts of the call tree. An example is shown in Listing 4.3.

```
shmac_prof --pt -u core_tq_update_current \
           --uc 2 -f ./shmac.elf <prof.txt
```

Listing 4.3: An example invocation of the `shmac_prof` program.

`--pt` prints the call tree and `-u` specifies that we are specifically looking for the `core_tq_update_current` function in the call tree. `--uc 2` means that the tree should be pruned to be only two levels deeper than the functions specified with `-u`. `-f` specifies the binary that produced the profiling output in prof.txt. This is needed to look up function names by their addresses using *addr2line*. Finally, the instrumentation data is read from standard input.

An example of the output is shown in Listing 4.4.

```
schedule (0x0) - 100.00% (20074) - 13.95% (2800) - 0 calls
  pick_from_heap (0x6fc4) -  9.65% (1937) -  2.99% (601) - 14984 calls
    core_tq_update_current (0x6e2c) -  2.01% (404) -  1.57% (315) - 14984 calls
      update_active_queue (0x28a8) -  0.44% (89) -  0.44% (89) - 14984 calls
      push_residual_tasks (0x297c) -  0.00% (0) -  0.00% (0) - 9591 calls
        linked_list_head (0x25b8) -  0.00% (0) -  0.00% (0) - 9591 call
```

Listing 4.4: A textual representation of the call tree, as outputted by the SHMAC profiler.

The output in Listing 4.4 contains:

- Function name.
- Where the function was called from.
- Percentage of samples that belong to this node or a child node.
- Number of samples that belong to this node or a child node.
- Percentage of samples that were taken in this function.
- Number of samples taken in this function.
- Number of times the function was called through this exact call path.

For a more graphical representation, `--pt-graph $filename` can be used to generate a call tree and save it as an EPS file. This requires *graphviz* to be installed. An output example from this function is showed in Figure 4.2. Note that this representation coalesces nodes slightly more than the textual output does: if a function A makes three calls from

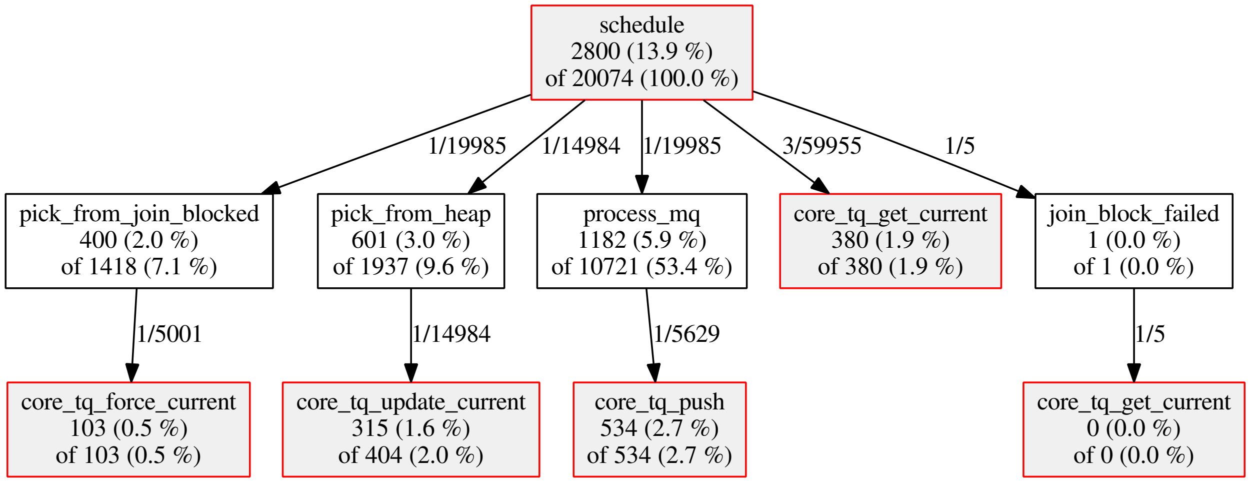


Figure 4.2: An example of a call tree generated by the profiler. This one is pruned to only contain the functions marked in red (and grey-filled) boxes. Each edge is annotated with the number of distinct call sites and total call count. Each node contains sample counts, both excluding and including children nodes.

different locations to function B, this will only show up as a single node in the call tree (although the number of distinct call sites are shown in edge labels).

Recognizing that this utility might not provide all features the user is looking for, a tool to convert the profile data into a *GNU prof* compatible file was written. `shmac_prof2gprof` converts output like the one shown in Listing 4.2 into a binary file *gprof* understands. An example is shown in Listing 4.5.

```
shmac_prof2gprof --sample-rate 250 -o gmon.out <./prof.txt
arm-none-eabi-gprof --brief --flat-profile shmac.elf gmon.out
```

Listing 4.5: An example of how the `shmac_prof2gprof` tool can be used.

One final thing to keep in mind when it comes to profiling is that function instrumentation skews the timing behavior of the functions being profiled – the small functions will appear to be, relative to their uninstrumented counterparts, a lot slower. Program counter sampling alone does not have this effect. For that reason, it might some times be worth considering running the profiler with just the sampling enabled (not compiled with `-finstrument-functions`). This gives an output profile that does not contain a call tree, but still has a complete histogram. This can then be converted to a *gprof*-compatible file.

This was done in the case of the `x264` benchmark (see Section 6.4), which, on average, performed 20,000 function calls per second. The *enter*/*exit* functions became the most time-consuming functions in the application, causing almost a 100% increase in runtime. Disabling instrumentation and running only with PC sampling brought the overhead back down to around 1.3%.

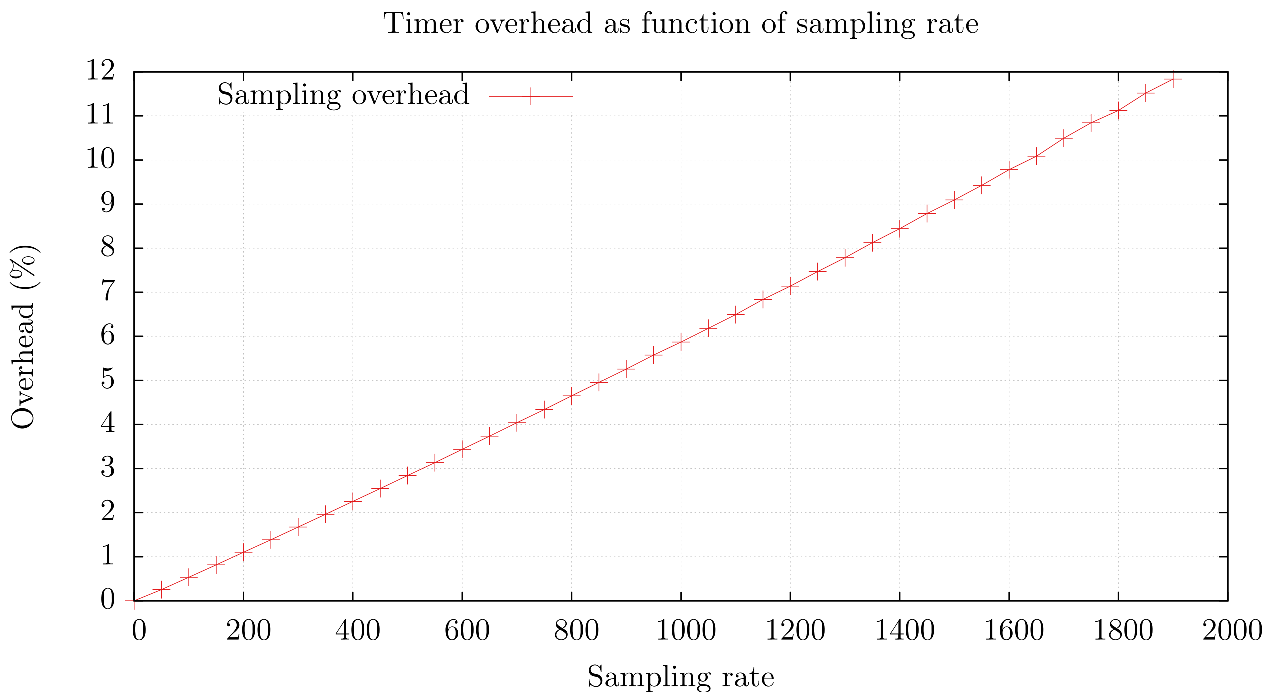


Figure 4.3: The overhead incurred by the profiler, measured as a function of program counter sampling rate.

4.3 Performance Results

The performance is obviously an extremely important metric by which to judge the profiler. A profiler should have as little impact on the application being profiled as possible. To investigate how well the scheduler performed, two tests were run:

- One that tests the overhead incurred by the sampling process.
- One that tests the cost of the *enter/exit* functions that are inserted into every function.

The results of the first test is shown in Figure 4.3. As expected, the overhead increases linearly with the sampling rate. 1% overhead is reached at around 160Hz. It should be noted that the interrupt service routine used is shared between all applications, including the scheduler. As such, it needs to push the complete register set to the stack before calling the profiler-specific interrupt routine. Since the profiler only needs the PC register, the pushing and popping of the other ∼16 registers incurs undue overhead.

Since Figure 4.3 shows overhead relative to a timer rate of 0, the cost of instrumentation (the *enter/exit* functions) is still present in the baseline. The overhead caused by instrumentation is completely different from that of the timer: rather than being sampling rate dependent, it depends on the application being profiled.

To gauge the overhead of this instrumentation, two different approaches were taken:

1. Compare the runtime of an instrumented application (without timer interrupts) to a version without instrumentation. If the total number of instrumented function calls are known, we can calculate the cost per instrumentation call.
2. Use the profiler to measure how often the program counter samples fall within the instrumentation functions. Even if these functions are not stored in the call tree, their samples will still be readily available in the histogram. Knowing the total number of function calls in the profiled application and the application runtime, we can calculate the cost of a single instrumentation call.

Both methods require knowing the number of instrumentation calls made for a given application. This is readily available from the profile data. The second approach also needs the total runtime. This is found by multiplying the number of PC samples to the sample rate.

For testing this, a Sudoku solver application was run to solve a fairly difficult puzzle; one that takes 3-4 seconds to solve with a backtracking algorithm (and without compiler optimization).

Running the application without any instrumentation took a total of 3,258.47 milliseconds. Running it with instrumentation and without PC sampling took 3,356.35 milliseconds – 3.00% more. The instrumentation results can tell us that a total of 2,660 *enter* functions were called (equally many *exit* calls, of course). That is, each *enter*/*exit* pair has a cost of $36.80\mu s$, or 2,208 clock cycles. For comparison, we can calculate the cost of a single interrupt event using data from Figure 4.3:

$$\text{time/irq} \;=\; \frac{\text{time for 1,800 irqs}}{\text{1,800}} = \frac{\frac{11.12\%}{100\%+11.12\%}}{1,800} = \underline{\underline{55.6\mu s}}$$

So an interrupt event is 51% more expensive than a combined *enter*/*exit* function call.

The second approach was ran six times at a sample rate of 1,500. Over these six runs, the average number of samples captured was 5,526 (a runtime of 3,684ms). On average, 52 samples fell in the *enter* routine and 35 in the *exit* function.

Combining these two sample counts with the known total cost of 2,208 cycles, we can now split the cost between them: `__cyg_profile_func_enter` takes approximately 1,320 cycles while `__cyg_profile_func_exit` takes ∼890 cycles. Considering that a single-cycle Amber instruction takes 17 cycles (because of instruction fetch latency), this translates into upper bounds of 78 and 53 instructions executed in the *enter*/*exit* functions, respectively. This seems like very acceptable performance results.

After having profiled the Sudoku application, we can examine how well the hashing mechanism worked. For that we need a metric by which to measure misses in our lookup function. These will either lead to a hashed lookup in a subsequent block or – in the worst case – a sequential scan.

The metric used to judge the hashing by is simply the number of slots considered minus one. This gives an optimal cost of zero and no upper bound on the cost. The goal should obviously

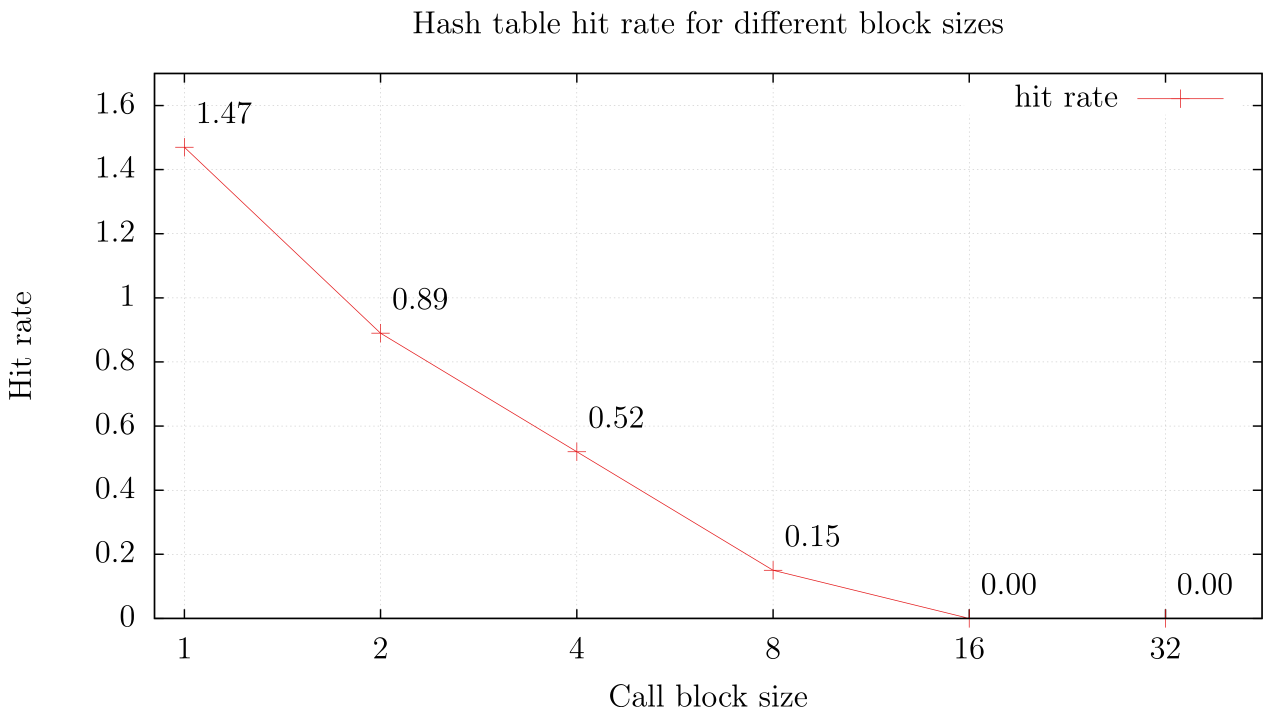


Figure 4.4: The hit rate for different block sizes. As the figure shows, sizes greater than 8 achieved perfect hit rates in the Sudoku application. Applications with a higher degree of function call fan-out might need even bigger block sizes to achieve this.

be an average that is far closer to zero than to one. We plot this metric for different values of `CHILD_BLOCK_SIZE`. This is shown in Figure 4.4. In the case of Sudoku, there is no point in using block sizes greater than 16.

To compare the simple hashing function used to one that takes the branch target into account, we take a closer look at the hash table lookup distance distribution for two different hash functions. The first hash function is the one described so far, and the second XORs the two registers together. This is shown in Figure 4.5. The second hash function performs better in the case of indirect function calls, but is more expensive to compute. In terms of the hit rate metric just described, the results for these two algorithms are 0.80 and 0.52, respectively.

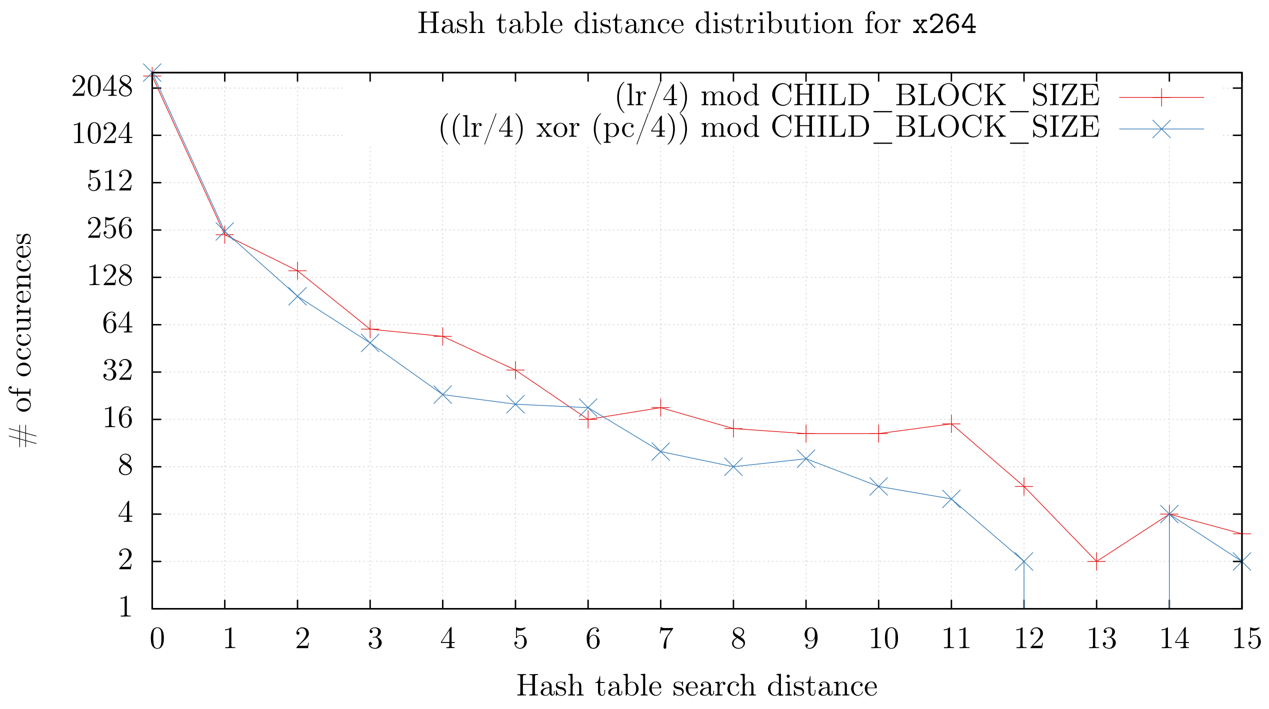


Figure 4.5: The distance distribution for the profiler's in-memory call tree for the `x264` benchmark, for two different hash functions. All in all, the results are decent for both functions. Hit rates are similar for a distance of zero or one, while the second hash function is more or less an order of magnitude better after this point.

Chapter 5

Evaluating The SHMAC Scheduler

To evaluate the scheduler, a few sample applications will be used throughout that hopefully capture a range of typical usage patterns. For each case, the behavior of the scheduler will be presented in detail and evaluated in terms of performance. Each sample application is compiled without data- and instruction cache, which strongly affects its performance: previous work suggests that instruction cache alone can yield a 10× speedup.

5.1 Sample Applications

Two threading patterns were thought to be enough to capture the most important performance characteristics of the scheduler. Those two patterns, *master/worker* and *join chain*, are briefly described here.

5.1.1 Master/Worker

Perhaps the most common of all thread creation/joining patterns: a single thread serves as *master* and creates *worker* threads as necessary. The master will typically just immediately join the worker threads once they are launched.

In this application, different numbers of workers will be used. Typically, the number of workers will be equal to the number of cores. However, it is also worth testing how well the scheduler performs if there are significantly more threads than cores. The core layout used for testing is showed in Figure 5.1. The timer tick rate used will be 10Hz. The benchmark code is included in Appendix E.

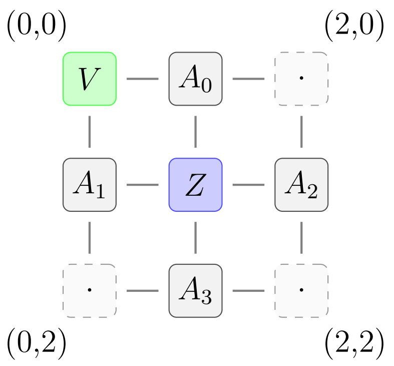


Figure 5.1: The 4-core layout used for testing purposes. A_i are Amber cores, V is the APB tile and Z is main memory. The dot represents a tile with just a router, containing neither functional capabilities nor memory storage.

5.1.2 Join Chain

In this test, the master core spawns a number of threads. The first thread immediately returns a constant, while all other threads X joins thread $X-1$, adds a constant to its return value before itself returning. The master thread joins the last of the worker threads. The code is shown in Appendix F.

5.2 Performance Results

This section presents the performance results the sample applications yielded. For each one, different sizes will be tested to give an impressions of how well the scheduler scales.

All results are extracted from event logs generated by the logging facility described in Section 3.2.10. The plots are generated by filtering events and plotting them using Gnuplot.

The tables contain information extracted from the event logs. Specifically, each interrupt start and stop event can be used to extract detailed information about interrupt handling time. Note that this does not include all overhead involved in an interrupt event, just the call to `schedule()`.

Since the event logger is only capable of tracing what happens inside of `schedule()`, an alternative test was used to capture the total overhead from the scheduler. A single worker thread was run on core 0 and work requesting was disabled. This way, both the main thread and the worker thread are pinned to core 0, but with the main thread blocked, waiting for the worker to finish. Then, the total time the application took to finish was measured for different scheduler rates. The results are shown in Figure 5.2.

The first plot shows the total application overhead (compared to the theoretical runtime of a $0Hz$ scheduler, found by linear regression), measured in percent. The second plot shows the total overhead (measured in ms) divided by total number of interrupts, which is the total time per interrupt. The straight line shows the average time spent in `schedule()`, according to the event logger. The difference between these last two (the filled region in Figure 5.2) is the overhead caused by:

1. Switching to interrupt mode, invoking a pre-registered interrupt handler.
2. Pushing registers $r0 - r12, lr, pc, spsr$ and $cpsr$ to the stack.
3. Switching to supervisor mode[1], but retaining a pointer to the *irq* stack where the registers are pushed.
4. Calling the global C-function that handles all interrupts.
5. Let said C-function call the appropriate timer tick handler.
6. Move the register structure to the *svc* stack.
7. Clear the interrupt flag and re-enable interrupts.
8. Disable interrupts.
9. Copy the register structure back to the *irq* stack.
10. Return to the interrupt handler.
11. Popping all registers from the stack.
12. Returning to user mode.

The `schedule()` function is called between steps 7 and 8. All these steps account for roughly a third of the total overhead, as Figure 5.2 shows. As this overhead is constant, it is not re-computed for all future performance benchmarks in this Chapter; all future results refer only to the time spent in `schedule()`.

5.2.1 Master/Worker, 4 threads

In this test, only four worker threads are created – one per core. The master thread will immediately join all worker threads and thus not be granted any timeslots until the slaves finish. Each worker is given a fixed (and equal) size of work to do. The schedule selected for this workload is shown in Figure 5.3.

[1]To enable nested interrupts, we need to move all state to the supervisor mode stack and continue handling the interrupt from this mode.

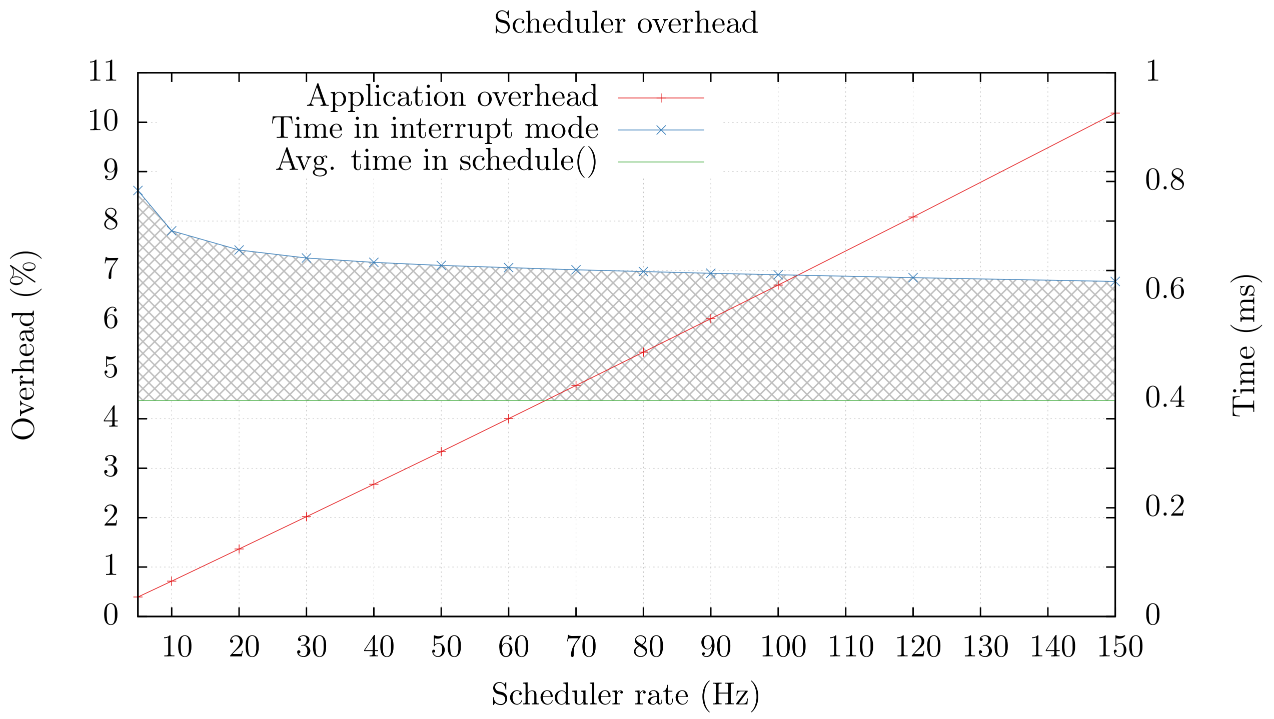


Figure 5.2: The total overhead induced by the scheduler. The first plot shows total application overhead in percent along the left y-axis. The two next plots show per-interrupt time (in ms), plotted along the right y-axis.

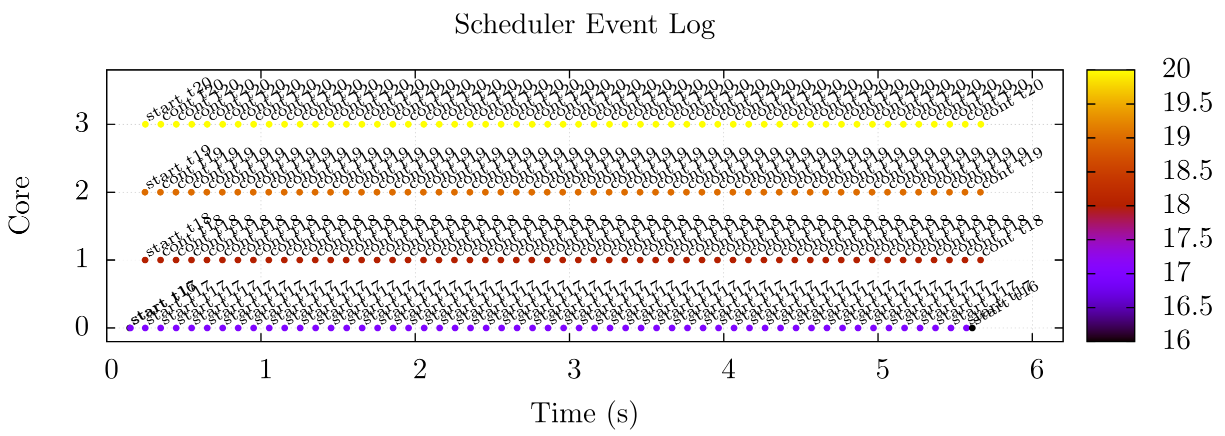


Figure 5.3: Scheduling 4 worker threads on 4 cores with 10Hz timer rate.

	Core			
	0	**1**	**2**	**3**
Number of interrupts	58	56	56	55
Total program time (ms)	5487	5420	5419	5383
Total time in interrupts (ms)	21.814	13.332	12.756	12.433
Total time in interrupts (%)	0.398	0.246	0.235	0.231
Average time in interrupts (ms)	0.376	0.238	0.228	0.226
Interrupts per second	10.6	10.3	10.3	10.2
Shortest interrupt (ms)	0.322	0.204	0.204	0.221
Longest interrupt (ms)	0.543	0.695	0.475	0.492

Table 5.1: Statistics gathered from running 4 worker threads on a 4-core computer with a 10Hz timer.

This is in itself fairly uninteresting: the threads are mapped directly to a core and stay there until finished. Some statistics relating to the scheduler are shown in Table 5.1.

There is no simple standard to judge these performance metrics by. In general, the scheduler should incur negligible overhead. This test revealed an average overhead per core of 0.28% at a reasonable timer rate and very light load. It is hard to say whether this is ”good enough”. Then again, this metric is dependent on the computer’s general performance. Turning on instruction cache would likely drop the same percentage down to well below 0.1%. At this point it is probably best to say that the performance is acceptable, as long as it scales well. That means that even with hundreds of threads, the percentage should be in the $1 - 2$ % range. Subsequent tests will determine this.

5.2.2 Master/Worker, 40 threads

This is the same test as in section 5.2.1, only with ten times as many threads. To keep the application runtime fairly low (which keeps the plots somewhat readable), each thread’s work size was reduced by a factor of 10. The schedule and performance results can be found in Figure 5.4 and Table 5.2, respectively.

As we can see, the average time in interrupts increased from 0.278 to 0.550 ms, an increase of 98% – roughly a 2x performance loss at a 10x increase in work.

5.2.3 Master/Worker, 400 threads

A final benchmark of the scheduler’s performance with the master/worker configuration will be with 400 threads – a hundred threads per core. This should be well beyond what one would ever do on an embedded platform, but it serves us well in stress testing the scheduler.

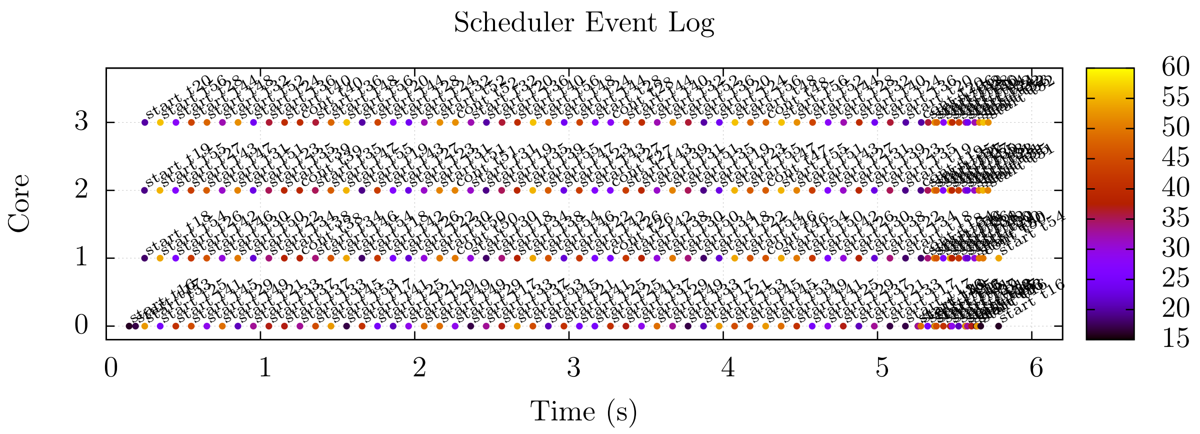


Figure 5.4: Scheduling 40 worker threads on 4 cores with 10Hz timer rate.

	Core			
	0	**1**	**2**	**3**
Number of interrupts	73	65	66	66
Total program time (ms)	5564	5504	5503	5503
Total time in interrupts (ms)	38.318	36.944	36.605	36.622
Total time in interrupts (%)	0.689	0.671	0.665	0.665
Average time in interrupts (ms)	0.525	0.568	0.555	0.555
Interrupts per second	13.1	11.8	12.0	12.0
Shortest interrupt (ms)	0.322	0.237	0.187	0.204
Longest interrupt (ms)	1.187	0.746	0.678	0.678

Table 5.2: Statistics gathered from running 40 worker threads on a 4-core computer with a 10Hz timer.

	Core			
	0	**1**	**2**	**3**
Number of interrupts	157	160	162	157
Total program time (ms)	5907	5909	5917	5905
Total time in interrupts (ms)	81.572	83.048	82.437	77.688
Total time in interrupts (%)	1.381	1.405	1.393	1.316
Average time in interrupts (ms)	0.520	0.519	0.509	0.495
Interrupts per second	26.6	27.1	27.4	26.6
Shortest interrupt (ms)	0.237	0.187	0.204	0.187
Longest interrupt (ms)	1.340	3.850	0.916	0.899

Table 5.3: Statistics gathered from running 400 worker threads on a 4-core computer with a 10Hz timer.

Function	**Time (%s)**
pick_from_heap	48.37 %
retire_current_task	15.86 %
core_tq_unblock_tasks	12.20 %
process_mq	4.13 %
pick_from_join_blocked	3.58 %
core_tq_get_current	1.79 %
core_lock	1.24 %
task_tick_hook	0.96 %
should_cancel	0.60 %
core_tick	0.87 %
core_unlock	0.64 %
join_block_failed	0.09 %
flush_finished_task	0.00 %

Table 5.4: Time spent in each of the functions called directly from the main `schedule()` function for the 400-thread master/worker benchmark.

The results are displayed in Table 5.3. The plot is omitted since it would contain too much data to plot in any meaningful fashion.

A load factor of 100 brought the average time in interrupts down from 0.550 to 0.511 ms: a decrease of 7% compared to 40 threads. This is surprising, to say the least.

The average time spent in the scheduler increased beyond one per cent for the first time: 1.37%. This is due to the increased number of interrupts, not because each interrupt took any longer than before. With that in mind, these results are a pleasant surprise.

A detailed profile is shown in Table 5.4. As expected, swapping tasks on the heap is by far the most expensive operation.

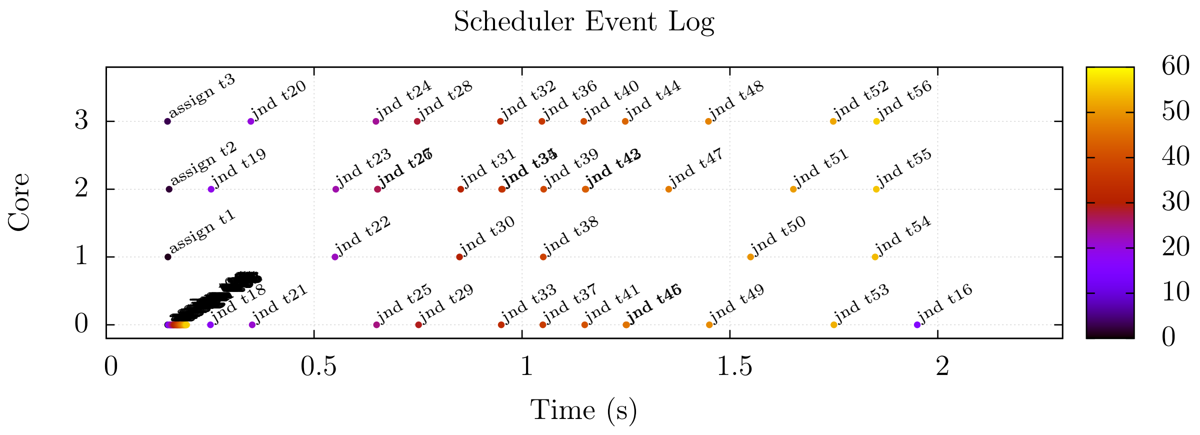


Figure 5.5: Scheduling 40 threads on 4 cores with 10Hz timer rate, where each thread joins its predecessor. The gaps illustrate points of time where the cores have to schedule the idle task (not explicitly shown). The *jnd* event occurs when a thread has joined another. This, along with *assign*, is the only event type shown.

	Core			
	0	**1**	**2**	**3**
Number of interrupts	39	21	40	37
Total program time (ms)	1746	1690	1690	1690
Total time in interrupts (ms)	24.799	15.673	23.544	20.491
Total time in interrupts (%)	1.420	0.927	1.393	1.213
Average time in interrupts (ms)	0.636	0.746	0.589	0.554
Interrupts per second	22.3	12.4	23.7	21.9
Shortest interrupt (ms)	0.356	0.356	0.221	0.221
Longest interrupt (ms)	1.408	1.306	1.120	1.153

Table 5.5: Statistics gathered from running a 40-thread join chain on a 4-core computer with a 10Hz timer.

5.2.4 Join Chain, 40 threads

While this is highly inefficient use of threads, it serves as a benchmark for thread creation/joining performance. Optimally, the speed at which we can join threads should be limited only by our timer frequency, as we have no efficient inter-core communication mechanism.

The event plot from this test is shown in Figure 5.5. The statistics is shown in Table 5.5. The entire test took around two seconds. With an average scheduling frequency of $20.1Hz$, this is very close to what we could have predicted: 40threads/$20.1Hz \approx 2s$. The somewhat higher *Total time in interrupts (%)* stems from this increased effective scheduling rate.

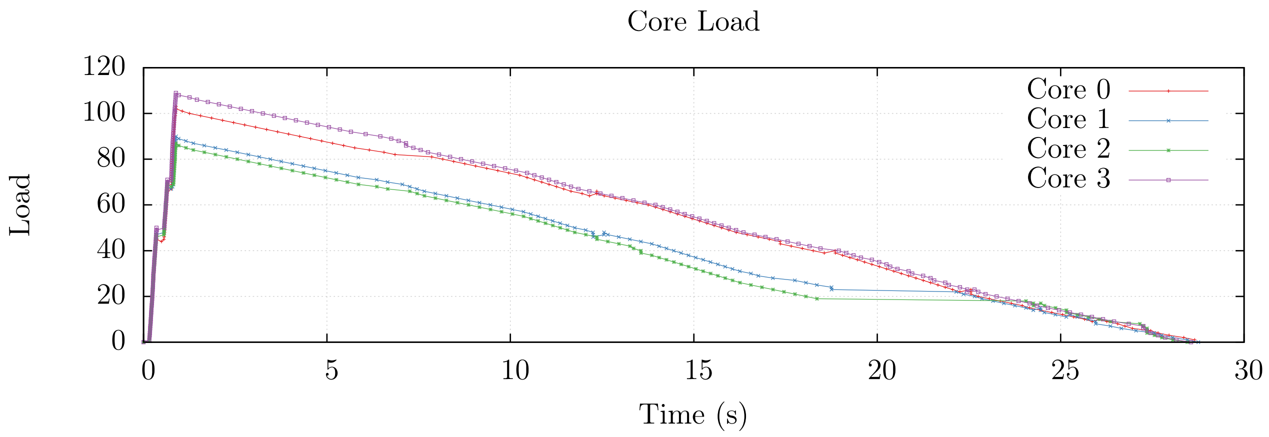


Figure 5.6: Scheduling 400 threads on 4 cores with 10Hz timer rate where each thread joins its predecessor. The plot illustrates the load factor for each core.

5.2.5 Join Chain, 400 threads

Once again, this is just a longer version of the previous test. The results are shown in Figure 5.6 and Table 5.6. Since using event plots are not feasible with this number of threads, a different visualization is provided: the load factor for each core over time.

Profiling the scheduler execution yields some interesting information about what actually takes time. The somewhat surprising results are shown in Table 5.7. As expected, most time was spent iterating through the tasks that have called `pthread_join` and deciding if they can wake up (`pick_from_join_blocked`). If this fails, the scheduler tries to pick from the running tasks (`pick_from_heap`), which it spends almost as much time doing.

The surprise is number three on the list, `core_tq_unblock_tasks`. This is the function that processes tasks blocking on condition variables, which in this test is an empty list. This part of the scheduler should probably be optimized in some way.

Number four and five on the list are as expected. With lots of idling, the cores are going to send a large number of messages to other cores, so `process_mq` was expected to take some time to process. `retire_current_task` decides if the current task should be switched out and, if so, saves its register set.

	Core			
	0	**1**	**2**	**3**
Number of interrupts	441	406	453	502
Total program time (ms)	28326	28425	28326	28425
Total time in interrupts (ms)	350.884	320.199	316.365	357.228
Total time in interrupts (%)	1.239	1.126	1.117	1.257
Average time in interrupts (ms)	0.796	0.789	0.698	0.712
Interrupts per second	15.6	14.3	16.0	17.7
Shortest interrupt (ms)	0.390	0.221	0.221	0.221
Longest interrupt (ms)	6.988	4.716	2.188	1.340

Table 5.6: Statistics gathered from running a 400-thread join chain on a 4-core computer with a 10Hz timer.

Function	**Time (%)**
pick_from_join_blocked	22.58
pick_from_heap	16.47
core_tq_unblock_tasks	14.60
process_mq	14.60
retire_current_task	11.21
core_tq_get_current	3.40
core_lock	1.36
task_tick_hook	1.19
core_tick	1.19
core_unlock	1.19
should_cancel	0.85
flush_finished_task	0.00
join_block_failed	0.00

Table 5.7: Time spent in each of the functions called directly from the main `schedule()` function for the 400-thread join chain benchmark.

Chapter 6

Ported Benchmarks

To prove that the Pthreads library works with existing benchmarks as-is, some benchmarks had to be evaluated. While the foremost goal of this was to ensure that the library works as expected, the performance results obtained are obviously also of interest.

This chapter presents the benchmark suites that were considered. Some benchmarks that were considered appropriate for the platform were ported to SHMAC, run and evaluated, These findings are presented here.

6.1 The PARSEC Suite

PARSEC[3] was an obvious candidate for porting. It is a benchmark suite intended for CMPs that covers a wide range of working sets, data sharing and off-chip traffic patterns. As the most widely known parallel benchmark it was an obvious candidate for porting.

The main challenge in selecting benchmarks are the limitations inherent in the SHMAC platform:

1. No I/O support. This means that applications must fit entirely in memory. It is a plus if rewriting the benchmarks to be I/O-free is fairly straightforward.
2. Limited memory: all benchmark data must fit in 32MB of RAM. This is a significant limitation as many benchmarks have data sets of considerable size, such as `x264`.
3. No floating point support. This means that floating point-intensive applications will suffer poor performance.

These factors limit the choice of benchmarks quite a bit. Table 6.1 sums up the benchmarks found in PARSEC. Only two benchmarks do not use floating point operations: `dedup` and

Program	**Problem size**	**Instructions (Billions)**				**Synchronization**		
		Total	**FLOPS**	**Rds**	**Wrts**	**L**	**B**	**C**
`blackscholes`	65,536 options	2.67	1.14	0.68	0.19	0	8	0
`bodytrack`	4 frames, 4,000 particles	14.03	4.22	3.63	0.95	114,621	619	
`canneal`	400,000 elements	7.33	0.48	1.94	0.89	34	0	0
`dedup`	184 MB data	37.1	0	11.71	3.13	158,979	0	1,619
`facesim`	1 frame, 372,126 tetrahedra	29.90	9.10	10.05	4.29	14,451	0	3,137
`ferret`	256 queries, 34,973 images	23.97	4.51	7.49	1.19	345,778	0	1255
`fluidanimate`	5 frames, 300,000 particles	14.06	2.49	4.80	1.15	17,771,909	0	0
`freqmine`	990,000 transactions	33.45	0.00	11.31	5.24	990,025	0	0
`streamcluster`	16,384 points per block, 1 block	22.12	11.6	9.42	0.06	191	129,600	127
`swaptions`	64 swaptions, 20,000 simulations	14.11	2.62	5.08	1.16	23	0	0
`vips`	1 image, 2662 × 5500 pixels	31.21	4.79	6.71	1.63	33,586	0	6,361
`x264`	128 frames, 640 × 360 pixels	32.43	8.76	9.01	3.11	16,767	0	1,056

Table 6.1: The benchmarks in the PARSEC benchmark suite, table copied from [3]. *Rds* = Reads, *Wrts* = Writes, *L* = Locks, *B* = Barriers, *C* = Conditions.

`freqmine`. Unfortunately, `freqmine` is the only benchmark listed in Table 6.1 that does not use Pthreads. For that reason it was excluded from further consideration.

Since floating point acceleration is an ongoing effort, it was thought that at least one benchmark should be fairly floating point intensive. `blackscholes` was selected for this purpose and was ported to SHMAC. It, along with its results, are presented in Section 6.2.

Since `freqmine` does not support Pthreads, only `dedup` was left as a floating point-free alternative. For that reason it was also ported to SHMAC. It is discussed in Section 6.3.

Porting a video encoding benchmark such as `x264` would also be a nice contribution to SHMAC's benchmark suite. This benchmark was ported without threading so that it could be accurately profiled[1]. The `x264` benchmark and its results are found in Section 6.4.

6.2 Blackscholes

`blackscholes` is a floating-point intensive benchmark that calculates the price of European options using the Black-Scholes formula. The work division is simple: the input set is divided equally between N worker threads.

[1]Since the profiler is currently only capable of running at a single core.

Threads	**Time**	**Time***
1	33.43	34.79
2	81.76	18.00
3	42.53	12.16
4	–	9.29
5	–	8.63
6	–	11.01
7	–	9.00
8	–	9.29

Table 6.2: The time (in seconds) the benchmark took to run for different thread counts. In each case, the number of options was set to 24 and the number of runs was 100. Error checking was turned on.

Since it is almost entirely made up of floating point operations and transcendental functions on floating point numbers, its performance is not expected to be great. It is, however, a nice benchmark to gauge the speedup of accelerators, such as an FPU accelerator. Researching accelerators for SHMAC is a parallel effort to this thesis.

Results

The performance of `blackscholes` was poor, as expected. Table 6.2 shows the results. Breaking the (single-threaded) execution time down with the SHMAC profiler did not yield any surprises. The top time-consuming functions are listed in Table 6.3.

The two different *time* columns are due to a bug that was discovered: sometimes, the application froze completely. This is denoted as ”–” in the table. Interestingly, it was discovered that the application could be ”woken back up” by simply running `shmac_dump` in the background. The exact command that was used to run this benchmark was:

```
./shmac.sh && while :; do shmac_dump 0 0x1f00000 /dev/null; done
```

The third column thus denotes the runtime of the benchmark with `shmac_dump` running in the background. This probably degrades the performance of memory operations somewhat. The bug also means that the data in the second column does not represent a correct execution of the benchmark. The third column, however, shows the exact behavior one would expect from such a parallelizable problem.

This bug does not seem to be caused by software, neither from the Pthreads library nor the `blackscholes` benchmark. It is more likely that it is a bug in the memory subsystem of the architecture. If this is the case, the bug should be investigated further.

% Time	Self Seconds	Function
61.09	20.98	__aeabi_dmul
8.79	3.02	__ieee754_sqrt
6.80	2.34	__mulsf3
6.12	2.10	__adddf3
4.52	1.55	__aeabi_ddiv
2.91	1.00	__ieee754_exp
1.90	0.65	__aeabi_fadd
1.25	0.43	__divsf3
1.04	0.36	CNDF(float)
0.94	0.32	__ieee754_log
0.69	0.24	exp
0.62	0.21	__aeabi_d2f
0.45	0.16	__cmpdf2
0.43	0.15	__extendsfdf2

Table 6.3: The top time-consuming functions in the `blackscholes` benchmark. As expected, *libgcc*'s math functions dominated the execution time.

6.3 Dedup

`dedup` is a parallel benchmark that performs compression in a pipelined fashion. Specifically, it works by splitting the task into five stages stages that run as separate threads[2]. The five stages are as follows:

Fragment
Read input and break it up into coarse-grained chunks. These chunks serve as work units for further processing. This is a serial stage.

FragmentRefine
The second stage uses Rabin-Karp fingerprints [9] to locate finer-grained data blocks within each input chunk. These fine-grained chunks are sent to the next stage.

Deduplicate
The third stage calculates SHA1-hashes of each chunk. These are used to identify possible duplicate chunks.

Compress
The fourth stage is compression. Each unique block is compressed only once since the previous stage does not send duplicates to this one.

Reorder
The final stage assembles the output stream from the distinct, possibly out-of-order blocks it receives from the fourth stage. This stage is inherently serial.

[2]Groups of threads, to be precise, as the benchmark pools threads for each stage.

Input Size	**Threads**				**Output Size**	**Compression Ratio**
	1	**2**	**3**	**4**		
1k	16.83	7.85	4.47	2.50	919	1.11x
2k	26.75	8.66	8.22	4.49	1.67k	1.20x
4k	44.45	13.75	7.78	3.38	3.21k	1.25x
8k	79.28	20.38	9.12	-	6.39k	1.25x
16k	180.47	44.19	-	-	13.22k	1.20x

Table 6.4: The runtime (in seconds) for the `dedup` benchmark for different input sizes and thread counts. "-" means the benchmark failed to run, at least intermittently.

The benchmark supports three types of compression: *gzip*, *bzip2* and *none*. *gzip* was chosen for this particular test case. This created an extra dependency: *zlib*. Fortunately, compiling *zlib* for SHMAC was as straightforward as exporting the correct `CC` and `CFLAGS` variables before building.

The input size listed in Table 6.1 of 184MB was not possible since SHMAC only has 32MB of RAM. This limitation notwithstanding, any input size in this order of magnitude would have taken too long to run in any case. It was decided that an input size between 1k and 16k was more appropriate.

It would obviously be beneficial to run the benchmark with different thread counts. Unfortunately, running with five threads caused memory allocation errors. Thus, the results presented are for one through four threads.

Results

Table 6.4 summarizes the results for different input sizes and thread counts. In general, the speedup results are linear, except for the switch from one to two threads, which is consistently superlinear. The reason for this was not investigated.

In addition, compression ratios are generally poor – this is due to the changes that were made in order to make the benchmark run at all. Table 6.5 details the changes that were made. This was necessary because the default values caused too coarse-grained partitioning of the input data, leaving one worker thread to do all the work.

Table 6.4 also shows that some benchmarks failed to run. This seemed to be due to memory errors. There were failures with four threads even without the changes listed in Table 6.5, which makes it unlikely that these changes caused the failures. In all likelihood, these errors are caused by the same bug that was discovered in the `blackscholes` benchmark.

Another issue that came to light was related to initializing the hash table. By default, the application used almost 60 seconds in the initialization phase, before a single thread was spawned. It spent 99% of this time initializing mutexes – apparently, the benchmark uses

Constant	In File	Original Value	Changed To
`NWINDOW`	rabin.h	32	4
`MinSegment`	rabin.h	1024	128
`RabinMask`	rabin.h	0xfff	0xff
`QUEUE_SIZE`	dedupdef.h	1M	512k
`MAXBUF`	dedupdef.h	128M	128
`ANCHOR_JUMP`	dedupdef.h	2M	8
`ITEM_PER_FETCH`	dedupdef.h	20	1
`ITEM_PER_INSERT`	dedupdef.h	20	1
`CHUNK_ANCHOR_PER_FETCH`	dedupdef.h	20	1
`CHUNK_ANCHOR_PER_INSERT`	dedupdef.h	20	1
Size parameter to `hashtable_create`	encoder.c	65536	1024

Table 6.5: Necessary changes to the `dedup` benchmark in order to achieve parallelism for our input sizes. The change in hash table size was not strictly speaking necessary, but it saves a lot of memory and startup time. It had no measurable effect on performance.

almost 100,000 of them by default. Since our Pthread library implementation stores these internally in red-black-trees, this takes some time to set up. While the necessity of 100,000 mutexes is debatable, the mutex library should probably handle this behavior better.

An interesting point is how well this pipelined, multi-threaded pattern maps to the different cores. The question of interest is: with four cores and four threads per stage, are we able to utilize all cores efficiently?

Taking advantage of the Pthreads logging mechanism described in Section 3.2.10, we can extract this information from the scheduler. Table 6.6 shows the results.

Disappointingly, a large fraction of the time is spent idling. Apart from the idle task, the compression stage is the dominant one. It would be interesting to see the effect of doubling the number of threads. Unfortunately, the benchmark failed to run with any more than four threads[3].

The benchmark exhibited optimal work distribution when only two threads were used (the total idle percentage among the cores was ∼200), while three threads did nothing to decrease that number. It could very well be that tuning the knobs in Table 6.5 would have resulted in better performance, although this was not tested.

It is also hard to exclude the possibility that the aforementioned bug caused this subpar performance, like it did with `blackscholes`. In this regard, Table 6.4 is ambiguous. On one hand, the super-linear speedups are unexpected. On the other hand, there are no spurious jumps in time like `blackscholes` exhibited.

[3]Possibly due to the bug discovered by `blackscholes`.

Core	Task						
	Idle	**Main**	**Frag.**	**Fragm. Ref.**	**Dedup.**	**Comp.**	**Reorder**
0	57.6 %	14.4 %	5.3 %	6.2 %	1.8 %	14.3 %	0.4 %
1	35.6 %	-	-	3.1 %	1.3 %	56.6 %	3.4
2	68.9 %	-	-	5.8 %	0.1 %	25.1 %	-
3	69.0 %	-	-	3.0 %	1.3 %	26.7 %	-

Table 6.6: Busy-time breakdown for each core, running a 4-threaded `dedup` benchmark on input of size 2kB. *Main* is the task that calls `Encode`. *Fragmentation* and *Reorder* are the sequential first and last steps of the pipeline. *Reorder* migrated to core 1, as the only task to migrate during the benchmark.

6.4 x264

`x264` is a lossy video encoder for the H.264/AVC codec. The standard was finalized in 2003 and it is today one of the most common video compression formats available. Its widespread use makes it an important application to benchmark. It also represents a fairly substantial increase in complexity compared to the previous applications described. For these reasons, it was ported to SHMAC.

H.264 is a block-oriented (with macroblocks being 16×16 pixels), motion compensation-based codec. Motion compensation is an important technique to reduce temporal redundancy in the video stream and often the most time-consuming part of the compression. In H.264, frames are compressed in one of the three following ways:

I-Frames: These are frames that do not reference other frames at all. Frames are encoded using *intra-prediction*, meaning that it uses previously compressed blocks in the same frame as reference.

P-Frames: P-frames use a previous I- or P-frame to encode the current one. Like I-frames, P-frames also use intra-prediction for compression. The added feature of referencing previous frames is called *inter-prediction*. These frames can typically attain a compression ratio of 50% compared to I-frames.

B-Frames: These use both previous and next I- or P-frames to construct the current one, thus achieving even better compression.

The typical GOP arrangement is IBBPBBP...BBI, although newer versions of the standard and modern encoders offer some more flexibility here. Other improvements over previous video encoding standards include CABAC, higher bit-depth color coding and variable block-size motion compensation.

Results

The benchmark was run on a 80×60 movie consisting of 125 frames. This file was generated by scaling PARSECs *eledream_640x360_128.y4m* sample input file using *ffmpeg*[4].

The application was compiled with `-O2`. Without any profiler instrumentation or PC sampling, this took 1,351 seconds to run – the complete output is shown in Listing 6.1. The exact commandline used to run the benchmark was:

```
./x264 -q 13 -o output.mkv eledream_80x60_128.y4m --progress
```

The application was also run with instruction cache turned on, which resulted in a 5.6× speedup. The results in this section will however refer to the cache-free version, as in the rest of this thesis.

```
PARSEC Benchmark Suite
yuv4mpeg: 80x60@25/1fps, 0:0
x264 [warning]: width or height not divisible by 16 (80x60), compression
will suffer.
x264 [info]: using cpu capabilities: none!
x264 [info]: profile Main, level 1.0
x264 [info]: slice I:3     Avg QP:10.00  size:  2772  PSNR Mean Y:51.17
U:57.88 V:58.33 Avg:52.50 Global:52.50
x264 [info]: slice P:125   Avg QP:13.00  size:  1219  PSNR Mean Y:47.62
U:54.40 V:54.61 Avg:48.95 Global:48.90
x264 [info]: mb I  I16..4:  0.0%  0.0% 100.0%
x264 [info]: mb P  I16..4:  0.0%  0.0%  3.6%  P16..4: 42.4% 21.4% 27.8%
0.0%  0.0%    skip: 4.7%
x264 [info]: SSIM Mean Y:0.9954008
x264 [info]: PSNR Mean Y:47.703 U:54.482 V:54.693 Avg:49.030
Global:48.956 kb/s:251.16

encoded 128 frames, 0.09 fps, 251.95 kb/s
```

Listing 6.1: Output generated by running the `x264` benchmark.

When the application was profiled with a sampling rate of 200 – but no instrumentation – the total execution time was 1,368 seconds, an overhead of 1.26%. Table 6.7 shows the most time consuming functions. The top two causes of poor performance seem to be lack of out-of-order- and speculative execution. There are a lot of small loops here that would benefit from vectorization and/or superscalar, speculative execution. The top five functions are shown, annotated with a line-level profile, in Appendix G.

Whereas the `blackscholes` benchmark clearly hit an achilles heel, this is not the case with `x264`. There are (virtually) no floating point operations and there are not enough multiplications for that to became an issue. Ways to increase performance further include

[4]http://www.ffmpeg.org/

% Time	Seconds	Name	Comment
15.04	205.73	pixel_satd_wxh	No time-consuming operations
12.57	171.89	mc_chroma	Spends 90% of its time in `mul`
7.01	95.93	quant_4x4	95% in `mul`
5.99	81.96	get_ref	`mul` is slowest part, but not by much
4.97	67.92	sub4x4_dct	Would benefit from OoO/speculation
4.40	60.12	add4x4_idct	Would benefit from OoO/speculation
3.62	49.50	block_residual_write_cabac	
3.37	46.09	hpel_filter	
3.22	44.09	memcpy	
3.15	43.03	dequant_4x4	
2.56	35.01	x264_pixel_ssd_16x16	
2.35	32.18	pixel_hadamard_ac	
2.28	31.20	x264_pixel_sad_x4_16x16	
2.26	30.87	x264_pixel_sad_x4_8x8	
1.78	24.28	x264_cabac_encode_decision_c	
1.75	23.97	ssim_4x4x2_core	
1.62	22.11	x264_pixel_sad_x4_8x16	
1.60	21.83	x264_pixel_sad_x4_16x8	
1.39	19.04	x264_pixel_ssd_8x8	
1.13	15.44	x264_pixel_sad_x3_16x16	

Table 6.7: Single-threaded, flat profile of the `x264` benchmark. Only the most time-consuming functions are shown.

1. Increasing multiplication performance.
2. Turning on instruction cache.
3. Fixing the data-cache bug that is causing us to currently run without any form of data cache.
4. Making a higher-performance core with advanced microarchitectural features, such as branch prediction, speculation, out-of-order execution and superscalar execution.
5. Writing accelerators for some of the most time-consuming functions. The profiler can be used to identify possible candidates.

The task of making a high-performance core is ongoing, in a project called Turbo Amber [1]. This core improves on the standard Amber core by adding an instruction fetch buffer, a multiscalar frontend, branch prediction, faster multiplications and a return address stack for faster function returns. These features will, in general, result in a 4x speedup. `x264` was in that regard no different, achieving a speedup just below 4x.

Chapter 7

Conclusion and Future Work

This thesis has presented an implementation and evaluation of a threading library for SHMAC, a platform for research into heterogeneous systems. While the library includes several synchronization primitives, the focus of this thesis has been on describing the arguably most important one: the preemptive task scheduler. Other notable contributions of this work include a fully-fledged profiler and a few automation tools to simplify certain SHMAC-related tasks.

This final chapter will reiterate some of the more important findings and relate these to the initial task assignment. The work that resulted in this thesis has also uncovered some areas that might be of interest for future work. These suggestions are proposed in Section 7.2.

7.1 Conclusion

Section 1.2 presented four distinct tasks that should be addressed. This section will address each one in turn and answer the question of how it has been solved.

Threading Library

The main goal of this work was to simplify the task of porting and/or implementing parallel benchmarks for the SHMAC platform. To that end, an abstraction layer was needed on top of the bare-metal platform SHMAC currently is.

The solution was to implement a POSIX-compliant library capable of scheduling and multiplexing tasks without any operating system support. This enables existing benchmarks built on top of Pthreads to be easily ported to SHMAC. It also provides a familiar abstraction to new developers working on SHMAC.

Chapter 3 has, in elaborate detail, explained the heart of the Pthreads library: the task scheduler. Section 3.3 details the status of the library with respect to API compatibility. It is thought that – in its current state – the library will be compatible with practically all existing applications that might be considered for porting. Most of the shortcomings are due to a lack of an MMU (Memory Management Unit) and virtual memory. Other functions do not make sense to implement without the existence of processes, something SHMAC obviously does not have.

The threading library has been verified through a series of unit tests as well as porting existing benchmarks.

Port existing benchmarks

This task was contingent on there being enough time once the other tasks were completed. It turned out to be enough time to port a couple of benchmarks: `blackscholes` and `dedup` from the PARSEC suite. The benchmarks are their respective results are presented in Chapter 6.

Simplify toolchain setup process

This task came to be as a result of many hours spent – by many parties – setting up a working toolchain complete with a *libc* implementation. As it turns out, this can be a challenge when using legacy instruction sets such as ARMv3.

The solution to this problem is described in [21].

Implement a testing framework

For hardware developers, it is often necessary to verify the correctness of certain changes. The existing suite of ported benchmark can here serve as a tool for regression testing. The challenge was that the build process for these benchmarks was not well unified: a simple call to `make` often either failed or produced applications that required in-depth knowledge to run.

A testing framework was intended to simplify the task of running a series of applications and verify that they all run correctly without requiring user intervention. The solution to this problem is described in [22].

Another contribution was made that is thought to be useful for future SHMAC work: the implementation of a profiler. The need for a profiler came from needing to verify the performance of the scheduler, rather than an explicit assignment task. This initial purpose notwithstanding, the profiler is completely generic in nature. An important part of working with SHMAC is inarguably performance analysis – a field in which a profiler is an invaluable tool. Chapter 4 has presented the profiler's implementation in detail.

7.2 Future Work

As the SHMAC project is still fairly young, there are several places where there is room for improvement – some of which were discovered during this work.

While there is ample room for improvement to the works described in this thesis, a few suggestions can be made to the platform in general as well. This section will provide suggestions both to the SHMAC project in general and to the contributions presented by this paper.

7.2.1 Pthreads Library

In general, the Pthreads library seems to perform admirably at the most central tasks it provides: creating and joining threads. There are other components of the library that do not perform equally well.

The `blackscholes` benchmark revealed that the **red-black trees** used internally to keep track of mutexes, condition variables and locks did not perform very well: in this benchmark, we managed to initialize condition variables at a rate of $\sim 1600/s$. An alternative would be the allocate all necessary state on the heap and put the pointer to this state in the handles passed to the API functions. For instance, a `pthread_mutex_t` could contain a pointer to a `pthread_internal_t` struct that store the necessary state. This approach has not been evaluated in detail, but should provide a substantial performance gain.

There are also no **scheduler-assisted mutexes** in the library. That is, the scheduler is happy to run a task that will spend its entire timeslice busy-waiting on a mutex. This is the way it is because the mutex library implementation predates the Pthreads library. The Pthreads library was thus written as a simple wrapper for the existing SHMAC mutex library. Additionally, the spinlocks are simply wrapping the mutex library.

A better solution would be the let the spinlocks wrap the existing SHMAC mutexes and reimplement the Pthreads mutexes with scheduler support.

The R/W lock component of Pthreads are **missing two functions**:
`pthread_rwlock_timedrlock` and `pthread_rwlock_timedwlock`. Extending the library with these functions is not particularly hard, it was just not a prioritized task. They should be implemented.

As SHMAC is a platform for heterogeneous research, a threading library should preferably possess **architectural awareness**, so it can schedule tasks at the appropriate tile types to maximize performance or energy efficiency. While the Pthread library does not expose any such facilities, it could be implemented without the client being aware. This would, however, require the hardware platform to expose more detailed information about the underlying tile configuration at runtime. This can be done the same way it currently exposes certain information, such as tile identifiers and XY coordinates: through memory-mapped, tile-local registers.

7.2.2 The Profiler

The profiler's feature set is to some extent dictated by architectural limitations. For instance, no core can start a timer on a different core. This imposes the limitation that each core must start its own instance of the profiler.

Second, the profiler is completely threading-unaware. The reason for this is twofold:

1. There are no threads in privileged modes, so there would always be a need to be able to profile a core without requiring a thread context. The interrupt routine found in `crt0.S` was written to support nested interrupts, which is needed for profiling in privileged processor modes.
2. More importantly, each core does not start out by executing a thread. The threading library is an optional library that will (probably) more often be excluded from an application than not.

Optimally, the profiler should be able to detect the use of threads and adapt. The difficulty here lies in making the scheduler and profiler work well together without making either dependent on the other.

Currently, the profiler works on a single core that is decided at compile-time. A simple, iterative improvement on the profiler would be to make it **profile all cores**, not just the one calling `prof_init`. Because of the aforementioned need for all cores to start their own timer, all cores would still have to initialize their own profiler instance. The output format would have to be adapted to support a variable number of cores and both `shmac_prof` and `shmac_prof2gprof` would need some modifications.

7.2.3 Other Improvements to SHMAC

What is thought to be a fairly serious **memory bug** was uncovered when porting the `blackscholes` and `dedup` benchmarks to SHMAC. This bug caused either complete system freeze or unstable performance. Running `shmac_dump` in the background seemed to make the application continue normally. This behavior could point to a bug in the memory subsystem of the platform rather than a software bug. This issue should be investigated further.

Inter-core interrupts was perhaps the most sorely missed feature from the hardware platform in implementing the scheduler. With the current implementation, a task invokes the scheduler explicitly in a number of cases. This ensures that the next task can be started immediately rather than having to wait for the next timer interrupt. For instance, this leads to highly efficient create/join performance on a single core.

However, the fact that cores cannot interrupt one another creates situations were one simply has to wait for the other core to reach a timer interrupt event. Figure 5.5 illustrates the problem well; when a task that is being joined exits, it cannot inform the joining task's core that it can immediately schedule said task. Instead, it has to wait for the scheduler to discover that by itself.

While this feature would have been beneficial to the scheduler, it also has wider applicability in inter-core communications.

Appendices

Appendix A

Automating the SHMAC Build Environment Setup

A.1 Problem Description

As the SHMAC project grows, a number of people have had to independently set up a working cross-compiler in order to work on SHMAC software development. This is a cumbersome process for several reasons:

1. SHMAC uses an old instruction set (ARMv3) for which all of GCC's features do not compile out of the box.
2. The memory map used by SHMAC does not match the standard ARM memory map.
3. In addition to a cross-compiler, a working standard library implementation is usually needed. This process of compiling GCC with a third-party C standard library implementation is not entirely intuitive.

For these reasons, a way of automating this setup process was needed. This reduced the overhead involved in setting up a new development environment and introduces a common toolchain for all developers.

A.2 Implementation

The obvious solution to the problem was chosen: script the install process. This includes doing sanity checks to determine if all required dependencies (such as GNU Make, a C++ compiler and libgmp) are in place, as well as downloading all the needed tarballs.

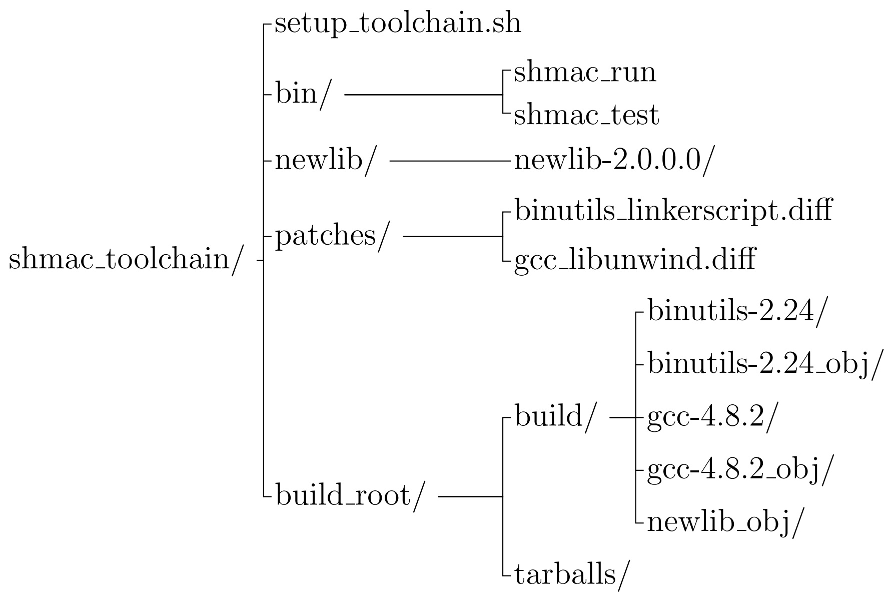


Figure A.1: The (top-level) folder structure of the toolchain installer. *bin/* contains binaries related to the testing framework that can optionally be installed. *newlib/* is a modified version of Newlib that compiles for SHMAC. Finally, the *patches/* are applied to GCC/Binutils once these are downloaded and unpacked. *build_root/* is the temporary folder used to compile everything. While the shown location is the default, it can be specified when the script is invoked.

In addition to the install script, some additional data was needed: the patches we need to apply and Newlib. We chose two different approaches to patching GCC/Binutils and Newlib: GCC and Binutils are patched when the compiler toolchain is installed. Newlib, on the other hand, is included in the installer as an already-patched folder.

The toolchain setup is included in the pre-existing repository used for all SHMAC-related development, where it can be found in the *toolchain* or *master* branch. On one of those branches, a folder structure as shown in Figure A.1 can be found in *shmac/software/shmac_toolchain*.

The entire setup process goes as follows:

1. First, sanity-check the host environment. This includes checking for the required build tools, such as GNU Make and texinfo. It also that the three required library header files are found: libgmp >= 4.2, libmpfr >= 2.4.0 and libmpc >= 0.8.0. If any requirement is found to be missing, the build process terminates immediately.
2. If all requirements are satisfied, the installer downloads the two required tarballs: GCC and Binutils. These are downloaded and extracted to a temporary build folder.
3. Subsequently, GCC and Binutils are patched. This includes patching GCC's libunwind to make sure it compiles for ARMv3. The Binutils linker script code is patched to

ensure that the built-in linker script satisfied SHMAC's requirements.

4. We are now ready to compile our binary utils, which include tools like *ld*, *cpp*, *readelf*, *ranlib*, etc.

5. The process is compiling GCC and Newlib is not entirely obvious. We need to cross-compile Newlib for the SHMAC architecture, but for that we need a working cross-compiler. Having a working cross-compiler includes having a C library (Newlib) compiled for SHMAC.

   This cyclicity requires us to split the GCC compilation process into stages. First, we compile a partial compiler (the Make rule called *all-gcc*). Then we use our partial cross-compiler to compile Newlib. Once that is done, we can return to building a fully working compiler with our recently compiled Newlib as its standard library.

   In short:

   (a) Compile GCC (stage 1)
   (b) Compile Newlib using the GCC from stage 1
   (c) Compile the full version of GCC

6. Finally, the installer will ask if the user wants to install two additional binaries (`shmac_run` and `shmac_test`). These are not directly related to the toolchain, but provides a mechanism for automatically running applications on SHMAC. This is primarily intended for benchmarking and regression testing. The `shmac_run` utility, intended to be run on the SHMAC host, requires the Python module *serial*.

At this point, a working cross-compiler for the *arm-none-eabi* target has been installed to /usr/local (unless the user specified another path to the setup script). Using *arm-none-eabi-gcc* (or *g++*), the user should now be able to compile applications that run on SHMAC.

A.3 Usage

The *setup_toolchain.sh* script can be run without additional parameters. If it is invoked in this fashion, it will use *./build_root/* as a temporary working directory and install the toolchain to */usr/local/*.

If the user wants to change either of these parameters, they can be supplied to the script. Two sample invocations are shown in Listing A.1.

```
# will install to /usr/local using /tmp/shmac as working directory
./setup_toolchain.sh /tmp/shmac

# will install to $HOME/toolchain
./setup_toolchain.sh ./build_root $HOME/toolchain
```

Listing A.1: Sample usage of the `setup_toolchain.sh` script.

The script does not clean up its working directory when it is finished. This is by design, as this might be useful if the user finds it necessary to make any changes at a later point. If this is not the case, the directory can be deleted once the installation is finished.

Appendix B

Implementing a Testing Framework for SHMAC

B.1 Problem Description

As the number of applications and benchmarks ported to SHMAC grows, running each one manually becames an increasingly tedious task. Hardware developers might want to quickly run a regression suite to verify the correctness of a hardware change, but the current suite of benchmarks is diverse and lacks consistency with regard to the compilation process. In addition, some benchmarks require user input during the benchmark run. Thus, running benchmarks often requires an in-depth understanding of each benchmark or, at the very least, more knowledge than should be required to run a simple benchmark.

This report details the implementation of a solution to this problem.

B.2 Introduction

The problem description outlines two distinct problems:

1. Automating the task of running several applications/benchmarks, verifying that each one runs correctly.
2. The need to be able to run each benchmark without requiring any form of user input.

The testing framework consists of a fairly simply tool to achieve both of these goals. Before looking at the implementation, an understanding of the typical development workflow is useful.

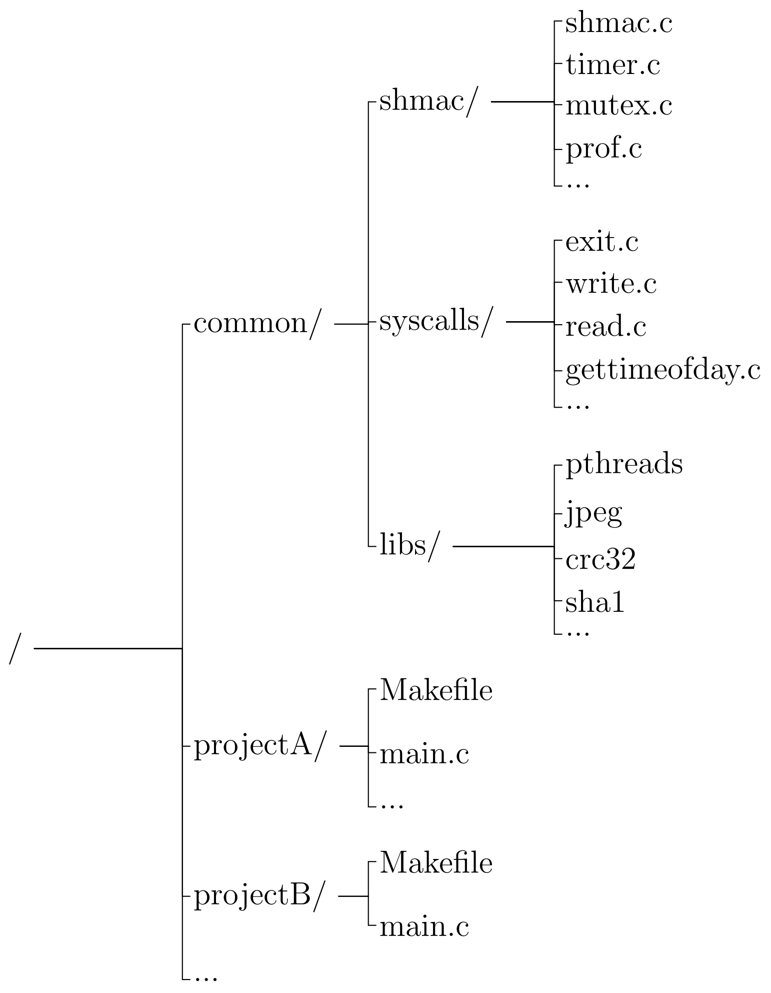


Figure B.1: The current SHMAC project structure. *common* contains code that is typically shared across projects.

B.3 SHMAC Development

The project structure is shown in Figure B.1. Every project has its own source files and Makefile. Optionally, they might choose to include the *shmac.a* archive and other libraries. *syscalls.a* is required as long as the project is compiled with libc since it contains stubs required by Newlib.

Every project currently has its own Makefile. An extremely simple Makefile typically provides the following rules:

1. A rule to compile source code into object files. The implicit built-in rule could also be used here.
2. A rule to combine the object files into an ELF (Executable and Linkable Format) file, optionally linking it with libc and *common.a*.

3. A rule to dump the ELF file as a raw binary.

This binary file is the one that is copied over to the Versatile Board, which is the development platform SHMAC uses. It is basically a normal computer running Linux with an FPGA attached to a COM port. The host operating system communicates with the FPGA via a set of utility programs (which again communicate with a kernel module, *shmac.ko*):

shmac_dump
A program used to extract data from SHMAC's memory space

shmac_program
A program used to write to SHMAC's memory space

shmac_reset
A program used to toggle the reset signal to every SHMAC core

Running a program on SHMAC thus typically consists for the following sequence of operations:

```
shmac_reset on
shmac_program 0 shmac.bin
shmac_reset off
```

Listing B.1: Programming a .bin file on the FPGA

Where shmac.bin is the binary file produced by our project Makefile. The kernel module exposes SHMAC's I/O facilities through a TTY device, `/dev/ttySHMAC0`. Any serial communication program can be used to communicate with SHMAC.

B.4 The Testing Framework

The testing framework is very much based on the existing work flow. It automates the task of compiling the test binary, `scp`-ing it to the SHMAC host, running the application and capturing output and exit code. Finally, it presents the status to the user. If the test failed, the entire output is shown.

Every project in the root folder has the responsibility of implementing a *test version* of itself (or not – developer's choice). Some times, applications will require user input during execution. For instance in the case of an image benchmark application, the application might ask the user to supply the image early on in the execution. For a test application, all of this has to be automated.

While this problem is left to the project developer, the general strategy will often be to compile any resources into object files and link them statically. Finally, the application could be compiled with a special flag that instructs it to use the linked resource instead of asking for user input.

The test framework is really just an application that scans a given folder for projects that seem to provide a *test feature*. The way the test framework locates tests is to simply recursively scan a given directory for Makefiles that provide a *test* rule. If it finds this, it is assumed to be a SHMAC test.

The test framework consists of two binaries: `shmac_test` and `shmac_run`. `shmac_test` is the main test binary and is run locally on the development system[1]. `shmac_run` is an application that is run on the SHMAC host. Its responsibility is to run a given binary on SHMAC and echo output until the application exits. The *_exit()* routine is expected to print "exit code: $code" as its final output. When `shmac_run` reads this, it exits with the same status code.

This binary was designed to be easily executed remotely. If the developer has a local *test.bin* file, he can simply execute:

```
ssh shmac_host shmac_run < test.bin && echo $?
```

This will execute the remote `shmac_run` utility, echo any output and exit with the correct status code. This is the process the `shmac_test` utility automates.

In short, this is what `shmac_test` does:

1. It scans a given directory for Makefiles that provide a *test* target. For each one, it does:
   (a) `chdir` to the project folder.
   (b) Execute "make test". If this fails, the user is informed that the compilation stage failed. Otherwise, this stage is expected to produce a `test.bin` file.
   (c) Execute `test.bin` remotely using ssh and `shmac_run`. For this to be seamless, the user should install his ssh key on the SHMAC host computer.
   (d) If ssh exits with status code 0, the test is assumed to have passed. If it does not, the exit code and output is echoed to the user.

Listing B.2 shows the relevant parameters `shmac_test` accepts. This, along with the description for each parameter, should give some insight into its intended usage.

[1] That is, the developer's local system, not the Versatile board

```
usage: shmac_test [-h] [-d host] [-r rule_name] [-o] [-c] [-s] [-v]
                  [root_directory]

A simple utility for running automated tests on SHMAC.

positional arguments:
  root_directory      The directory under which to look for test
                      applications.
                      Defaults to the current directory

optional arguments:
  -h, --help          show this help message and exit
  -d host             The remote host to upload test cases to.
  -r rule_name        The make rule to invoke.
  -o, --compile-only  Do not actually run the test.
  -c, --clean         Run 'make clean' before 'make test'?
  -s, --scan-only     Does not compile or run any tests, just scans
                      for projects that would have been run.
  -v, --verbose
```

Listing B.2: The help text produced by `shmac_test -h`.

Appendix C

The `schedule()` Function

```c
void schedule(irq_regs_t *registers)
{
    if (!core_lock(ID, TRYLOCK)) {
        join_block_failed();
        log_task(IRQ_FAILED_CORE_LOCK, core_tq_get_current());
        return;
    }


    log_task(IRQ_STARTED, core_tq_get_current());

    process_mq();
    core_tq_unblock_tasks(NONE);

    task_t *current_task = core_tq_get_current();

    if (current_task == NULL)
    {
        pick_from_heap(registers);
    }
    else if (should_cancel(current_task))
    {
        cancel(registers, current_task);
    }
    else
    {
        task_tick_hook(current_task);

        int switch_task = 0;
        if (current_task->state == FINISHED) {
            flush_finished_task(current_task);
            switch_task = 1;
        } else {
            switch_task = retire_current_task(registers, current_task);
        }

        if (switch_task) {
            if (!pick_from_join_blocked(registers)) {
                pick_from_heap(registers);
            }
        } else {
            // not switching, but request work if continuing idle task
            if (is_idle_task(current_task)) {
                request_work(current_task);
            }
        }
    }

    core_tick(ID);
    core_unlock(ID, 1);

    log_task(IRQ_FINISHED, core_tq_get_current());
}
```

Listing C.1: The `schedule()` function.

Appendix D

Task Prioritization Functions

```
#define se2t(ts,se) struct task_struct *(ts) = \
    (struct task_struct*)((struct sched_entity*)(se))->task


/* Comparison function for SCHED_OTHER */
static int task_comparator_heap_other(void *task1_se, void *task2_se)
{
    se2t(task1,task1_se);
    se2t(task2,task2_se);

    int s1 = task1->attrs.schedparam.sched_priority,
        s2 = task2->attrs.schedparam.sched_priority;

    // special case for idle function: always down-prioritize
    if (s1 == 0 && s2 != 0) return -1;
    if (s2 == 0 && s1 != 0) return  1;

    // if tasks have different priorities, order them by
    // 't->epoch_sched_count / t->priority' (in increasing order).

    // note that mul is slow on the Amber core: 33 cycles _with_
    // cache and almost 600 cycles without cache.
    int t1 = ((sched_entity_t*)task1_se)->epoch_sched_count * s2;
    int t2 = ((sched_entity_t*)task2_se)->epoch_sched_count * s1;

    return (t1 < t2) ? 1 : (t1 > t2 ? -1 : 0);
}


/* Comparison function for SCHED_RR */
static int task_comparator_heap_rr(void *task1_se, void *task2_se)
{
    // round-robin scheduling is simply ordering by
    // the number of times a task has been run.
    int t1 = ((sched_entity_t*)task1_se)->epoch_sched_count,
        t2 = ((sched_entity_t*)task2_se)->epoch_sched_count;

    return (t1 < t2) ? 1 : (t1 > t2 ? -1 : 0);
}

/* Comparison function for SCHED_FIFO */
static int task_comparator_heap_fifo(void *task1_se, void *task2_se)
{
    // order by (priority, created_time)
    se2t(task1,task1_se);
    se2t(task2,task2_se);

    int t1 = ((sched_entity_t*)task1_se)->created_time,
        t2 = ((sched_entity_t*)task2_se)->created_time;

    int s1 = task1->attrs.schedparam.sched_priority,
        s2 = task2->attrs.schedparam.sched_priority;

    if (s1 != s2)
        return (s1 > s2) ? 1 : (s1 < s2 ? -1 : 0);
    else
        return (t1 < t2) ? 1 : (t1 > t2 ? -1 : 0);
}
```

Listing D.1: The functions used to establish a partial ordering of tasks in the three task heaps in the scheduler.

Appendix E

Master/worker benchmark

```c
#include <stdio.h>
#include <pthread.h>
#include <assert.h>

#include "shmac/prof.h"

#define THREADS 1
#define FIXED_WORK 1000000
#define PROFILE_SAMPLE_RATE 500

void *fixed_work(void *ptr) {
    volatile unsigned long work = FIXED_WORK;
    while (work--);
    return NULL;
}

int main(void) {
    // initialize the profiler here, but don't start it. It is started
    // early on in the schedule() function and stopped at the end of it.
    // That way we collect only the data we want.
    prof_init(PROFILE_SAMPLE_RATE, 0);

    // Create all threads, assert that everything works as expected
    pthread_t ts[THREADS];
    for (int x = 0; x < THREADS; x++) {
        assert(pthread_create(&ts[x], NULL, &fixed_work, NULL) == 0);
    }

    // wait for all tasks to finish
    for (int x = 0; x < THREADS; x++) {
        assert(pthread_join(ts[x], NULL) == 0);
    }

    // Stop the profiler and print its output
    char *ptr;
    unsigned int len;
    prof_finalize(&ptr, &len);
    printf("Profile is at 0x%x, read %d bytes.\n",
            (unsigned int)ptr, len);

    return 0;
}
```

Listing E.1: The master/worker benchmark

Appendix F

Join chain benchmark

```c
#include <stdio.h>
#include <assert.h>
#include <pthread.h>

#include "shmac/shmac.h"
#include "shmac/prof.h"

#define THREADS 1
#define PROFILE_SAMPLE_RATE 500

void *recurse(void *arg) {
    if ((int)arg == 0) {
        return (void*)1;
    } else {
        int nn;
        assert(pthread_join(pthread_self() - 1, (void**)&nn) == 0);
        return (void*)(nn + 1);
    }
}

int main(void) {
    // initialize the profiler here, but don't start it. It is started
    // early on in the schedule() function and stopped at the end of it.
    // That way we collect only the data we want.
    prof_init(PROFILE_SAMPLE_RATE, 0);

    // Create all threads, assert that everything works as expected
    pthread_t ts[THREADS];
    for (int x = 0; x < THREADS; x++) {
        assert(pthread_create(&ts[x], NULL, &recurse, (void*)x) == 0);
    }

    // Join only the last one and print its return value
    unsigned int sum = 0;
    assert(pthread_join(ts[THREADS -1], (void**)&sum) == 0);
    assert(sum == THREADS);

    // Stop the profiler and print its output
    char *ptr;
    unsigned int len;
    prof_finalize(&ptr, &len);
    printf("Profile is at 0x%x, read %d bytes.\n",
            (unsigned int)ptr, len);

    return 0;
}
```

Listing F.1: The join chain benchmark

Appendix G

x264 Line-level profile

To annotate the source code, it was necessary to extract the raw samples from the profile format described in Section 4.2. The following one-liner does that:

```bash
egrep -o "histogram=(.*)" prof.txt | \
    sed -e 's/histogram=(//g;s/)//g;s/,/\n/g' | \
    perl -pe "s/^/0x/g" | \
    addr2asm -a x264.elf | \
    arm-none-eabi-addr2line -sfpe x264.elf | \
    grep "^quant_4x4 at" | \
    shmac_annotate -c 2 -f ./x264/src/common/quant.c
```

Listing G.1: A Bash oneliner to extract samples from the profile. This specific script annotates the function `quant_4x4`, found in *quant.c*.

Two custom applications were used here:

1. *addr2asm* reads samples from standard input, uses the given ELF file to look up the instruction type and prints out a adjusted sample. For instance, `mul` instructions often take a long time. If they are followed by a `add` instruction, the program counter will stall at the `add` instruction while the core is actually busy performing the `mul`. This script uses knowledge of each instruction's execution time to adjust for this fact. While this approach is not perfect (it can wrongly shift a sample to the previous instruction when it shouldn't have), it certainly improves the result's precision.

2. *shmac_annotate* parses output from *addr2line*, correlates it with the source file and prints the annotated version shown in the listings shown below.

The following series of listings show the most time consuming functions from `x264`, in descending order.

```
#define HADAMARD4(d0,d1,d2,d3,s0,s1,s2,s3) {\
    int t0 = s0 + s1;\
    int t1 = s0 - s1;\
    int t2 = s2 + s3;\
    int t3 = s2 - s3;\
    d0 = t0 + t2;\
    d2 = t0 - t2;\
    d1 = t1 + t3;\
    d3 = t1 - t3;\
}
 0.0 %: static int pixel_satd_wxh(
 0.0 %     uint8_t *pix1, int i_pix1, uint8_t *pix2,
 0.0 %     int i_pix2, int i_width, int i_height )
 0.4 %: {
 0.0 %:   int16_t tmp[4][4];
 0.0 %:   int x, y;
 0.0 %:   int i_satd = 0;
 0.0 %:
 1.8 %:   for( y = 0; y < i_height; y += 4 )
 0.0 %:   {
 3.3 %:     for( x = 0; x < i_width; x += 4 )
 0.0 %:     {
 0.0 %:       int i;
 0.0 %:       uint8_t *p1 = pix1+x, *p2 = pix2+x;
 0.0 %:
 5.6 %:       for( i=0; i<4; i++, p1+=i_pix1, p2+=i_pix2 )
 0.0 %:       {
 4.1 %:         int a0 = p1[0] - p2[0];
 2.2 %:         int a1 = p1[1] - p2[1];
 2.2 %:         int a2 = p1[2] - p2[2];
 4.9 %:         int a3 = p1[3] - p2[3];
30.1 %:         HADAMARD4(tmp[i][0],tmp[i][1],tmp[i][2],tmp[i][3],a0,a1,a2,a3);
 0.0 %:       }
 2.2 %:       for( i=0; i<4; i++ )
 0.0 %:       {
 0.0 %:         int a0,a1,a2,a3;
32.3 %:         HADAMARD4(a0,a1,a2,a3,tmp[0][i],tmp[1][i],tmp[2][i],tmp[3][i]);
 9.2 %:         i_satd += abs(a0) + abs(a1) + abs(a2) + abs(a3);
 0.0 %:       }
 0.0 %:
 0.0 %:     }
 0.5 %:     pix1 += 4 * i_pix1;
 0.9 %:     pix2 += 4 * i_pix2;
 0.0 %:   }
 0.0 %:
 0.0 %:   return i_satd / 2;
 0.1 %: }
```

Listing G.2: The most time consuming function in x264. It spends most of its time in the HADAMARD4 macro.

```
 0.0 %: static void mc_chroma( uint8_t *dst, int i_dst_stride,
 0.0 %:                        uint8_t *src, int i_src_stride,
 0.0 %:                        int mvx, int mvy,
 0.0 %:                        int i_width, int i_height )
 1.7 %: {
 0.0 %:   uint8_t *srcp;
 0.0 %:   int x, y;
 0.0 %:
 0.0 %:   const int d8x = mvx&0x07;
 0.0 %:   const int d8y = mvy&0x07;
 0.0 %:
 0.1 %:   const int cA = (8-d8x)*(8-d8y);
 0.7 %:   const int cB = d8x  *(8-d8y);
 0.1 %:   const int cC = (8-d8x)*d8y;
 0.6 %:   const int cD = d8x  *d8y;
 0.0 %:
 0.2 %:   src  += (mvy >> 3) * i_src_stride + (mvx >> 3);
 0.0 %:   srcp = &src[i_src_stride];
 0.0 %:
 1.1 %:   for( y = 0; y < i_height; y++ )
 0.0 %:   {
 2.8 %:     for( x = 0; x < i_width; x++ )
 0.0 %:     {
27.8 %:       dst[x] = ( cA*src[x]  + cB*src[x+1] +
63.2 %:            cC*srcp[x] + cD*srcp[x+1] + 32 ) >> 6;
 0.0 %:     }
 0.8 %:     dst  += i_dst_stride;
 0.0 %:
 0.0 %:     src   = srcp;
 0.8 %:     srcp += i_src_stride;
 0.0 %:   }
 0.2 %: }
```

Listing G.3: The mc_chroma function is one where the performance of multiplications really plays a big part.

```
#define QUANT_ONE( coef, mf, f ) \
{ \
  if( (coef) > 0 ) \
    (coef) = (f + (coef)) * (mf) >> 16; \
  else \
    (coef) = - ((f - (coef)) * (mf) >> 16); \
}

 0.0 %: static void quant_4x4(
 0.0 %:     int16_t dct[4][4],uint16_t mf[16],uint16_t bias[16])
 0.7 %: {
 0.0 %:   int i;
 1.3 %:   for( i = 0; i < 16; i++ )
97.1 %:     QUANT_ONE( dct[0][i], mf[i], bias[i] );
 1.0 %: }
```

Listing G.4: Practically all time on quant_4x4 is spent in a multiplication macro.

```
 0.0 %: static uint8_t *get_ref( uint8_t *dst,   int *i_dst_stride,
 0.0 %:                          uint8_t *src[4], int i_src_stride,
 0.0 %:                          int mvx, int mvy,
 0.0 %:                          int i_width, int i_height )
10.5 %: {
 0.0 %:   int qpel_idx = ((mvy&3)<<2) + (mvx&3);
 4.8 %:   int offset = (mvy>>2)*i_src_stride + (mvx>>2);
66.5 %:   uint8_t *src1 = src[hpel_ref0[qpel_idx]] + offset + \
                        ((mvy&3) == 3) * i_src_stride;
 0.0 %:
 1.5 %:   if( qpel_idx & 5 ) /* qpel interpolation needed */
 0.0 %:   {
 6.2 %:     uint8_t *src2 = src[hpel_ref1[qpel_idx]] + offset + ((mvx&3) == 3);
 1.3 %:     pixel_avg( dst, *i_dst_stride, src1, i_src_stride,
 0.0 %:          src2, i_src_stride, i_width, i_height );
 0.0 %:     return dst;
 0.0 %:   }
 0.0 %:   else
 0.0 %:   {
 0.7 %:     *i_dst_stride = i_src_stride;
 1.4 %:     return src1;
 0.0 %:   }
 7.1 %: }
```

Listing G.5: The single multiplication performed in get_ref dominates the function's execution time.

```
 0.0 %: static void sub4x4_dct(int16_t dct[4][4],uint8_t *pix1,uint8_t *pix2)
 3.3 %: {
 0.0 %:   int16_t d[4][4];
 0.0 %:   int16_t tmp[4][4];
 0.0 %:   int i;
 0.0 %:
 0.0 %:   pixel_sub_wxh((int16_t*)d,4,pix1,FENC_STRIDE,pix2,FDEC_STRIDE);
 0.0 %:
 2.6 %:   for( i = 0; i < 4; i++ )
 0.0 %:   {
10.3 %:     const int s03 = d[i][0] + d[i][3];
11.7 %:     const int s12 = d[i][1] + d[i][2];
 1.3 %:     const int d03 = d[i][0] - d[i][3];
 1.4 %:     const int d12 = d[i][1] - d[i][2];
 0.0 %:
 9.7 %:     tmp[0][i] =   s03 +   s12;
 4.0 %:     tmp[1][i] = 2*d03 +   d12;
 2.5 %:     tmp[2][i] =   s03 -   s12;
 4.8 %:     tmp[3][i] =   d03 - 2*d12;
 0.0 %:   }
 0.0 %:
 2.1 %:   for( i = 0; i < 4; i++ )
 0.0 %:   {
10.5 %:     const int s03 = tmp[i][0] + tmp[i][3];
11.8 %:     const int s12 = tmp[i][1] + tmp[i][2];
 1.1 %:     const int d03 = tmp[i][0] - tmp[i][3];
 0.6 %:     const int d12 = tmp[i][1] - tmp[i][2];
 0.0 %:
 9.5 %:     dct[i][0] =   s03 +   s12;
 4.4 %:     dct[i][1] = 2*d03 +   d12;
 3.1 %:     dct[i][2] =   s03 -   s12;
 4.2 %:     dct[i][3] =   d03 - 2*d12;
 0.0 %:   }
 1.1 %: }
```

Listing G.6: Nothing is particularly time consuming in sub4x4_dct, but the function is called a larger number of times.

Bibliography

[1] Akre, A. T. and Bøe, S. (2014). Turbo amber, a high performance processor core for shmac. Master's thesis, Norwegian University of Science and Technology.

[2] Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., and Singhania, A. (2009). The multikernel: A new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 29–44, New York, NY, USA. ACM.

[3] Bienia, C., Kumar, S., Singh, J. P., and Li, K. (2008). The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81, New York, NY, USA. ACM.

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

[5] Borkar, S. (2007). Thousand core chips: A technology perspective. In *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, pages 746–749, New York, NY, USA. ACM.

[6] Bovet, D. P. and Cesati, M. (2005). *Understanding the Linux Kernel, 3d Edition*. O'Reilly Media.

[7] Durantom, M., Black-Schaffer, D., Bosschere, K. D., and Maebe, J. (2013). The hipeac vision for advanced computing in horizon 2020.

[8] Graham, S. L., Kessler, P. B., and Mckusick, M. K. (1982). Gprof: A call graph execution profiler. *SIGPLAN Not.*, 17(6):120–126.

[9] Karp, R. M. and Rabin, M. O. (1987). Efficient randomized pattern-matching algorithms.

[10] Kinsy, M. A., Pellauer, M., and Devadas, S. (2013). Heracles: A tool for fast rtl-based design space exploration of multicore processors. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '13, pages 125–134, New York, NY, USA. ACM.

[11] Kumar, R., Farkas, K. I., Jouppi, N. P., Ranganathan, P., and Tullsen, D. M. (2003). Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 81–, Washington, DC, USA. IEEE Computer Society.

[12] Kumar, R., Tullsen, D. M., Ranganathan, P., Jouppi, N. P., and Farkas, K. I. (2004). Single-isa heterogeneous multi-core architectures for multithreaded workload performance. *SIGARCH Comput. Archit. News*, 32(2):64–.

[13] Moore, G. (2005). Excerpts from a conversation with gordon moore.

[14] Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics*, 38(8).

[15] Pellauer, M., Adler, M., Kinsy, M., Parashar, A., and Emer, J. (2011). Hasim: Fpga-based high-detail multicore simulation using time-division multiplexing. In *In HPCA*, pages 406–417. IEEE Computer Society.

[16] Rotem, E., Ginosar, R., Mendelson, A., and Weiser, U. (2013). Power and thermal constraints of modern system-on-a-chip computer. In *Thermal Investigations of ICs and Systems (THERMINIC), 2013 19th International Workshop on*, pages 141–146.

[17] Rusten, L. T. and Sortland, G. I. (2012). Implementing a heterogeneous multi-core prototype in an fpga. Master's thesis, NTNU, Trondheim, Norway.

[18] Tan, Z., Waterman, A., Avizienis, R., Lee, Y., Cook, H., Patterson, D., and Asanović, K. (2010). Ramp gold: An fpga-based architecture simulator for multiprocessors. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 463–468, New York, NY, USA. ACM.

[19] Van Craeynest, K. and Eeckhout, L. (2013). Understanding fundamental design choices in single-isa heterogeneous multicore architectures. *ACM Trans. Archit. Code Optim.*, 9(4):32:1–32:23.

[20] Wee, S., Casper, J., Njoroge, N., Tesylar, Y., Ge, D., Kozyrakis, C., and Olukotun, K. (2007). A practical fpga-based framework for novel cmp research. In *Proceedings of the 2007 ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays*, FPGA '07, pages 116–125, New York, NY, USA. ACM.

[21] Wikene, H. O. (2014a). Automating the SHMAC build environment setup. Technical report, Norwegian University of Science and Technology.

[22] Wikene, H. O. (2014b). Implementing a testing framework for SHMAC. Technical report, Norwegian University of Science and Technology.