ABSTRACT

LEE, SANG HOON. Exploration of Fine-Grained Helper Computing Parallelism on a Chip Multiprocessor. (Under the direction of James Tuck.)

As software systems become more complicated, due to the importance of reliability and security, prior studies have proposed inlining meta-functions into applications for detecting bugs and security vulnerabilities. However, because these software techniques add frequent, fine-grained instrumentation to programs, they often incur large runtime overheads. Because they increase the critical path of the main application, they sometimes increase it by orders of magnitude. For example, Memcheck, a tool that runs using Valgrind, has been reported to slow down applications by a factor of $2 \times$ to $30 \times$ [32, 52, 45], and even then it only checks for a subset of memory errors. Fortunately, because multicore systems become ubiquitous and there is abundant parallelism in meta-functions, many of these meta-functions can be extracted and executed in parallel with the main application. The goal of this work is to speed up main application with fine-grained meta-functions by efficiently offloading meta-functions from main application into helper threads and by efficiently communicating through queues.

This dissertation considers two systems to achieve the goal. One is a software-based technique and the other is a hardware-based technique. First, in order to concurrently run meta-functions and main application in parallel on a chip multiprocessor, we propose Automatic Parallelization for Fine-Grained Meta-Functions on a CMP, which is a software-based technique. We seek to understand how to extract meta-functions from a program and execute them efficiently on a multi-core processor. Our parallelization strategy automatically extracts meta-functions from the main program and executes them in customized helper threads — threads constructed to mirror relevant fragments of the main program’s behavior in order to keep communication and overhead low. To get good performance, we consider optimizations that reduce communication and balance work among many threads. We evaluate our parallelization strategy on Mudflap, a pointer-use checking tool in GCC. To show the benefits of our technique, we compare it to a manually parallelized version of Mudflap. We run our experiments on an architectural simulator with support for fast queueing operations. With static heuristics, it achieves an average $1.64 \times$ speedup using 8 cores. Compared to a manually optimized version of the Mudflap framework, our approach is only 19% slower. With dynamic heuristics, it achieves on average $1.96 \times$ speedup using 8 cores. Our automatically parallelized code using dynamic load balance is competitive, on average, to the manually parallelized version on a simulated 8-core system. Furthermore, our approach introduces very little overhead in the main program — it is kept under 100%, which is more than a $5.3 \times$ reduction compared to serial Mudflap.

Second, queues are commonly used in multithreaded programs for synchronization and com-
munication. However, because software queues tend to be too expensive to support fine-grained parallelism, hardware queues have been proposed to reduce overhead of communication between cores. Hardware queues require modifications to the processor core and need a custom interconnect. They also pose difficulties for the operating system because their state must be preserved across context switches. To solve these problems, we propose HAQu: Hardware-Accelerated Queueing for Fine-Grained Threading on a CMP, which is a hardware-based technique to reduce fine-grained queueing overheads in multithreaded programs as well as to support OS transparency. HAQu adds hardware to a CMP that accelerates operations on software queues. Our design implements fast queueing through an application’s address space with operations that are compatible with a fully software queue. In addition, our design provides accelerated and OS-transparent performance. We evaluate our design in the context of application domains: offloading fine-grained checks for improved software reliability using Mudflap, and automatic, fine-grained parallelization using decoupled software pipelining.
Exploration of Fine-Grained Helper Computing Parallelism on a Chip Multiprocessor

by
Sang Hoon Lee

A dissertation submitted to the Graduate Faculty of North Carolina State University in partial fulfillment of the requirements for the Degree of Doctor of Philosophy

Computer Engineering
Raleigh, North Carolina
2012

APPROVED BY:

Gregory Byrd
Yan Solihin

Frank Mueller
James Tuck
Chair of Advisory Committee
DEDICATION

To my lovely wife, Hyunjoo Park and my precious daughter, Rebecca(Jiyoo)
for their support and love.
BIOGRAPHY

Sang Hoon Lee was born in Seoul, Korea in 1975. He obtained his B.S. and M.S. degrees in Electrical Engineering from Yonsei University in Seoul, Korea in 2001 and 2003, respectively. He worked as a software engineer at Samsung Electronics in Suwon, Korea, where he developed software for mobile phones. He joined North Carolina State University in 2007 to pursue his Ph.D degree. During his Ph.D study, he has focused his research on computer architecture, micro-architecture and automatic parallelization on a Chip Multiprocessor under the direction of Dr. James Tuck. On March 1, 2012, he defended his Ph.D dissertation.
ACKNOWLEDGEMENTS

First of all, I would like to thank my advisor Dr. James Tuck. He has always encouraged me to continue my research. His advice and discussion with him always motivated me to continuously focus on my work. With his encouragement and guidance, I was able to finish my dissertation. In addition, he was my mentor in my personal life. He always gave me advice whenever I had personal issues. I am really fortunate to have Dr. Tuck as my advisor.

I would like to thank my Ph.D committee members, Dr. Gregory Byrd, Dr. Yan Solihin and Dr. Frank Mueller for their suggestions and encouragement.

I would like to thank all the CESR student colleagues. Specially, Liang Han, George Patasilaras, and Rajeshwar Vanka shared research ideas and personal life with me. In addition, I would like to thank Joonmoo Huh and Ying Yu for discussion about research ideas.

I would like to thank all the Korean friends I’ve met at NC State University since 2007. I have been so happy to enjoy my life in Raleigh with them.

Last but not least, I would like to express my sincere appreciation and gratitude to my wife Hyunjoo and my precious daughter Rebecca. I’ll never forget your love and sacrifice.
# TABLE OF CONTENTS

List of Tables ........................................................................................................ vii
List of Figures ........................................................................................................ viii

## Chapter 1 Introduction ......................................................................................... 1
  1.1 Meta-functions .................................................................................................. 2
  1.2 Exploiting parallelism ....................................................................................... 3
  1.3 Problems ........................................................................................................... 4
  1.4 Major Proposals of This Thesis ........................................................................ 6
    1.4.1 Automatic Parallelization of Fine-Grained Meta-Functions on a Chip
    Multiprocessor .................................................................................................... 7
    1.4.2 HAQu: Hardware-Accelerated Queue ......................................................... 8
  1.5 Report Organization ......................................................................................... 9

## Chapter 2 Automatic Parallelization of Fine-Grained Meta-Functions on a CMP ......................................................................................................................... 10
  2.1 Fine Grained Meta-function Parallelization .................................................... 10
    2.1.1 Our Parallelization Approach: Custom Helpers ...................................... 12
    2.1.2 Providing Scalability ............................................................................... 12
    2.1.3 Summary ................................................................................................... 13
  2.2 Parallelization Algorithm ................................................................................ 13
    2.2.1 Helper Function Parallelization Algorithm .............................................. 14
    2.2.2 Baseline Helper Extraction ..................................................................... 15
    2.2.3 Custom Helper Extraction ..................................................................... 18
    2.2.4 Example .................................................................................................... 21
    2.2.5 Supporting Multiple Helpers ................................................................... 21
  2.3 Case Study: Mudflap ....................................................................................... 24
    2.3.1 Mapping Mudflap to our Framework ....................................................... 25
    2.3.2 Manual Parallelization ............................................................................ 27
  2.4 Related Work .................................................................................................. 27
  2.5 Summary ......................................................................................................... 28

## Chapter 3 HAQu: Hardware-Accelerated Queueing for Fine-Grained Threading on a Chip Multiprocessor ........................................................................................................ 30
  3.1 Main Idea: Hardware Accelerated Queue ....................................................... 30
    3.1.1 Motivation ............................................................................................... 30
    3.1.2 Our Proposal: Hardware-Accelerated Queue ........................................... 32
  3.2 System Architecture ....................................................................................... 33
    3.2.1 Allocation and Layout ............................................................................ 33
    3.2.2 Enqueue and Dequeue Operations ......................................................... 34
    3.2.3 Making Fine-Grained Queueing Efficient .............................................. 34
    3.2.4 Ensuring Programmability and OS Transparency .................................. 39
3.3 Implementation .................................................. 43
  3.3.1 Microarchitecture Description ............................ 43
  3.3.2 Pipelined Execution Description .......................... 43
  3.3.3 Managing QFU Resources ................................... 45
  3.3.4 Instruction Implementation Details ....................... 46
3.4 Related Work ................................................... 46
3.5 Summary ......................................................... 47

Chapter 4 Evaluation .............................................. 48
  4.1 Evaluation of Automatic Parallelization ...................... 48
    4.1.1 Experimental Setup ...................................... 48
    4.1.2 Baseline Helper Extraction ............................... 48
    4.1.3 Custom Helper Extraction ................................ 49
    4.1.4 Dynamic Load Balancing .................................. 50
    4.1.5 Optimizations ............................................. 53
    4.1.6 Overheads ................................................ 56
  4.2 Evaluation of HAQu ........................................... 57
    4.2.1 Experimental Setup ...................................... 57
    4.2.2 Micro Benchmarks ......................................... 58
    4.2.3 Pointer-use Checking Acceleration ....................... 63
    4.2.4 Decoupled Software Pipelining ......................... 65

Chapter 5 Conclusion and Future Work .......................... 68
  5.1 Conclusion .................................................... 68
    5.1.1 Automatic parallelization of fine-grained meta-functions 68
    5.1.2 HAQu: Hardware-Accelerated Queue ....................... 69
  5.2 Future work .................................................. 69

References ......................................................... 71
LIST OF TABLES

Table 1.1  Runtime overheads of Mudflap. ........................................ 2
Table 2.1  Characterization of Mudflap overheads normalized to an optimized base-line without it. .......................................................... 11
Table 2.2  Meta-functions used by Mudflap. ........................................ 24
Table 4.1  Architecture simulated. All cycle counts are in processor cycles. 49
Table 4.2  Configurations and parallelizations. .................................... 50
Table 4.3  Architecture simulated. All cycle counts are in processor cycles. 59
Table 4.4  Mudflap configurations. ...................................................... 59
Table 4.5  DSWP configurations with varied queues. ............................. 59
LIST OF FIGURES

Figure 1.1 Bottlenecks due to workload imbalance. ....................... 5
Figure 2.1 A simple example of our Helper Extraction. ...................... 20
Figure 2.2 Generated main and helper functions with Dynamic Load Balancing using HASH. .......................................................... 22
Figure 2.3 Generated main and helper functions with Dynamic Load Balancing using counters. ....................................................... 23
Figure 2.4 Example code with Mudflap annotation. .............................. 25
Figure 2.5 Mudflap Manual Parallelization. ........................................ 26
Figure 3.1 Lamport’s queueing operations single-producer, single-consumer queue on a sequentially consistent processor. .................. 31
Figure 3.2 Enqueue (a) and dequeue (b) operations. Assumed cache hits with a dual-ported, 2-cycle access L1 cache. ............................... 35
Figure 3.3 Enqueue and dequeue instructions. ..................................... 35
Figure 3.4 Relaxed memory ordering for queue operations. ..................... 36
Figure 3.5 Algorithm for reducing accesses to shared variables. ............. 37
Figure 3.6 Important cases for providing OS transparency and programmability. .. 40
Figure 3.7 Memory operations through context switch. ......................... 41
Figure 3.8 Context switch codes of Linux Kernel 3.2.6 [1]. ..................... 42
Figure 3.9 Hardware Accelerated Queue added to a CMP. ....................... 44
Figure 4.1 Scalability of Base Helper Extraction with 4-cores and 8-cores. ...... 51
Figure 4.2 Speedups of Parallelized Mudflap with empty run-time functions. .. 52
Figure 4.3 Speedups of Parallelized Mudflap. ..................................... 53
Figure 4.4 The number of _mf_check in each helper thread on 8-core CMP (gap). 53
Figure 4.5 The number of _mf_check in each helper thread on 8-core CMP (gzip). 54
Figure 4.6 Speedups of Parallelized Mudflap with various load balancing techniques. 54
Figure 4.7 Speedups of pipelining communications compared to other parallelizations ............................................................ 55
Figure 4.8 Scalability of automatic parallelization using static load balance with 4-cores, 8-cores and 16 cores. ................................. 56
Figure 4.9 Scalability of automatic parallelization using static load balance with 4-cores, 8-cores and 16 cores. ................................. 57
Figure 4.10 The relative size of the binary. ........................................ 58
Figure 4.11 The main thread’s dynamic instruction counts. ..................... 60
Figure 4.12 Micro benchmarks using queues on NarrowRing. ................. 61
Figure 4.13 Micro benchmarks using queues on WideRing. ..................... 62
Figure 4.14 Parallelizing Mudflap using Hardware Accelerated Queue. ........ 64
Figure 4.15 Speedup of Parallelized Mudflap. ..................................... 65
Figure 4.16 Speedup of DSWP over Base using 2 cores. ......................... 67
Chapter 1

Introduction

One major trend in computing is the continuing increase in the complexity of software systems. Such an increase is motivated by the expectation of increasingly powerful hardware (faster processors, larger memory, etc.), programming environments that provide higher abstraction to the programmers, and the increasing diversity of environments in which the software runs. To ensure efficient, reliable, and secure execution of complex software systems, programmers increasingly rely on meta-functions, functionalities that do not contribute directly to the output of a program but help ensure efficient, reliable, and secure execution of the program. For example, software bugs are a major problem in industry. The National Institute of Standards & Technology estimated in 2002 that inadequate infrastructure for testing software cost close to $22 billion [48]. The same study predicts that if only half of the bugs that remained in post-production software were detected and eliminated, that cost could be recouped. Consequently, prior studies have proposed inlining meta-functions into applications for detecting bugs. However, because these software techniques add frequent, fine-grained instrumentation to programs, they often incur large runtime overheads. Because they increase the critical path of the main application, they sometimes increase it by orders of magnitudes. For example, Memcheck, a tool that runs using Valgrind, has been reported to slow down applications by a factor of 2× to 30× [32, 52, 45], and even then it only checks for a subset of memory errors. Other than that, various forms of meta-functions are implemented today in response to the diverse requirements of software execution, ranging from garbage collection, security checks, runtime correctness verification, performance instrumentations, and so on.

Fortunately, because multicore systems have become ubiquitous and there is abundant parallelism in meta-functions, many of these meta-functions can be extracted and executed in parallel with the main application on multicore systems. Therefore, concurrent execution between the meta-function code and the application was proposed as a solution in prior studies [34, 37, 53]. However, to achieve good performance, we find that efficient parallelization techniques and
Table 1.1: Runtime overheads of Mudflap.

<table>
<thead>
<tr>
<th>App</th>
<th>Normalized execution.</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2</td>
<td>11.1</td>
</tr>
<tr>
<td>gap</td>
<td>6.6</td>
</tr>
<tr>
<td>gzip</td>
<td>3.9</td>
</tr>
<tr>
<td>mcf</td>
<td>5.6</td>
</tr>
<tr>
<td>parser</td>
<td>7.1</td>
</tr>
<tr>
<td>twolf</td>
<td>6.1</td>
</tr>
<tr>
<td>vpr</td>
<td>5.4</td>
</tr>
<tr>
<td>Geo.Mean.</td>
<td>6.2</td>
</tr>
</tbody>
</table>

efficient communication on multicore systems are required.

In the remainder of this chapter, we will consider overheads due to meta-functions and previous techniques to reduce overheads in more detail. In addition, we will consider the problems motivating our work. Finally, we will introduce one proposal based on software and the other proposal based on hardware.

1.1 Meta-functions

Meta-functions are extra functionality added to a program to augment its behavior in some useful way. For example, to detect bugs of program, a program’s execution is typically monitored in detail to detect the first instance of a read or write to an invalid memory location, to detect double free, or to detect and uninitialized read. However, because these software techniques add frequent, fine-grained instrumentation to programs and they increase the critical path of the main application, they often incur large runtime overheads. For example, Memcheck [45] is known to slow the execution of some programs by as much as 30 times. Purify [40] and Mudflap [9], lightweight tools in comparison, often add non-trivial cost. Table 1.1 shows runtime overheads of Mudflap normalized to a subset of uninstrumented programs in SPECint. These overheads likely limit the frequency of use. Furthermore, if the execution time overhead from using these techniques can be reduced, not only can they be applied more aggressively during development, but they may also be employed in production runs. Therefore, in order to reduce runtime overheads, there have been many studies. Softbound [30] recently was proposed for spatial memory safety. It detects only spatial memory violations with less runtime overheads than other tools, but it doesn’t detect temporal safety violation which is supported by other tools [40, 44, 45, 9]. Some works have focused on reducing the overheads with hardware support.
HardBound [8] is a hardware/software approach. SoftBound is inspired by HardBound. It provides hardware bounded pointers. It checks bounds for dereferencing and propagates bounds efficiently. Chuang et al. [5] proposed a mechanism for accelerating meta data checks. They provide ISA extensions and additional hardware support to eliminate overheads due to access to meta-data. Chen et al. [3] proposed a mechanism for fine-grain instruction monitoring that supports operations similar to that of Mudflap. This system works for general instruction-grain monitoring tools. These previous works focused on accelerating meta-data use with specialized hardware supports.

1.2 Exploiting parallelism

As described in Section 1.1, while meta-functions ensure reliability and security for programs, they incur large runtime overheads because of fine-grained instrumentation to programs. Consequently, there have been many works to reduce runtime overheads by limiting systems’ coverages or by being supported by hardware. In addition, because multicore systems become ubiquitous and there is abundant parallelism in meta-functions, prior studies have investigated exploiting parallelism between the application and its security meta-functions.

Patil and Fischer [37] focused on parallelizing checking of pointer and array accesses on multiprocessors. However, due to the large overheads of communication and synchronization on their target systems, their approach was to replicate all code that generated pointer dereferences and execute it in a separate process. For work that could not be replicated, like program inputs or outputs, the two processes synchronized. They were able to achieve concurrent execution of pointer-use checking with low execution time overhead, but they replicated a large fraction of the program’s execution on an additional processor. Nightingale et al. proposed Speck [33], which exploits speculative parallelism between the application being protected with its security meta-functions. Shadow profiling [28] and SuperPin [53] execute original application with instrumented application slices in parallel. Shetty et al. [41] and Kharbutli et al. [20] parallelize memory management functionality for increased security. Plakal et al. [38] proposed parallelizing garbage collection using program slices on multiprocessors. However, they did not implement the slices, rather they only evaluated the potential.

One alternative way to exploit parallelism between the application and its meta-functions is to use thread-level speculation (TLS). TLS allows frequent data sharing at low cost, but without a guarantee that the speculative work will be useful. Examples of work in bug detection with TLS include [34, 57, 19]. Oplinger and Lam [34] investigated the use of speculative threads for software reliability. Oplinger and Lam were the first to point out the value of using speculative threads to hide the overheads of data monitoring and checking code, like the kind Mudflap uses for pointer-use checking. They considered monitoring code in general and investigated
techniques for optimizing it to run in parallel with the application. All studies employ thread-
level speculation to hide the overhead of program checking. It has been applied in security by
Speck [33], as discussed previously.

1.3 Problems

In Section 1.2, we reviewed various studies focusing on exploiting parallelism to reduce runtime
overheads due to fine-grained meta-functions. Although all the studies show their efficiency
and potentials on multicore systems. They still suffer from a variety of problems. In this
section, we point out three important problems which must be solved for exploiting parallelism
for fine-grained meta-functions on a chip multiprocessor.

**Problem 1: Granularity of data communication** While a meta-function can be an ar-
bitrary function, we focus on fine-grained meta-functions. Because we focus on offloading meta-
functions from a main program, fine-granularity becomes more critical. In order to concurrently
run meta-functions and main program in parallel on a chip multiprocessor, meta-functions code
is extracted from a program and scheduled on helper processes or thread contexts. In addition,
at runtime, a large amount of data communication between main and helper processes/threads
on extra cores is required because fine-grained meta-functions take inputs from main program.
Although meta-functions code is extracted from a program and scheduled on helper processes or
thread contexts, code for data communication between main and helpers is inserted into main
program and helper processes/threads. That is, code for data communication becomes major
bottleneck and incurs large runtime overheads instead of meta-functions code. Therefore, to
reduce runtime overheads due to data communication, how to decrease the granularity of data
communication must be considered.

**Problem 2: Load balance** Given the large runtime overhead introduced through meta-
functions, multiple helpers are required to successfully parallelize them. For example, as shown
in Table 1.1, the overhead in bzip2 with only one helper thread can’t be significantly reduced.
However, in order to use multiple helpers as efficient as possible, load balance must be care-
fully considered. That is, the meta-functions should be distributed evenly among multiple
helpers. Figure 1.1(a) shows a scenario that unbalanced workload causes stalls among multi-
ple threads. In this scenario, main thread and some helper threads stall until a helper thread
having large workload finishes its job. In Figure 1.1(a) and (b), Q1 ,Q2, and Q3 represent
single-producer/single-consumer communication queues between main and helper1, between
main and helper2, and between main and helper3, respectively. In Figure 1.1(a), instruction D
(send1) in main thread is blocked because Q1 is already full. In order to perform D, helper1
must execute instruction A’(recv1) to make Q1 available again. However, until meta-function in
helper1 is completed, A’ can not be executed. As a result, instruction E (send2) and F (send3)
Figure 1.1: Bottlenecks due to workload imbalance.
can not be executed until instruction D is completed. In addition, helper2 and helper3 stall because instruction E’ and F’ are blocked until helper2 and helper3 receive the data through Q2 and Q3, respectively. Figure 1.1(b) shows that how long main, helper2, and helper3 stall because of unbalanced workload among helpers.

**Problem 3: Delay of data communication** As described in Problem 1, code for data communication becomes major bottleneck and incurs large runtime overheads instead of meta-functions code. Therefore, we pointed out granularity of data communication as an important problem. However, there is another problem due to data communication. Because queues are a commonly used programming construct in multithreaded programs for data communication, we use queues for data communication in the work. For coarse-grained data communication, software queues are efficient programming construct. However, to support fine-grained data communication, software queues are too expensive. Because software queues consist of too many instructions, enqueue and dequeue operations are inefficient and the total latency is pretty high. In addition, they use shared memory to support data communication for multithreaded programs. Therefore, unless software queues are implemented carefully, the delay through the queue is not low enough to support concurrent execution. As an alternative to software queues, hardware queues have been proposed in some studies to support fine-grained data communication between cores because the total latency is low and the delay through the queue is low. However, they are unattractive because hardware queues require a large design cost to the processor core. In addition, it is not easy for programmers to use hardware queues because they require understanding underlying architectures. They also require modification to the existing operating systems because their state must be preserved across context switches.

1.4 Major Proposals of This Thesis

The work seeks to develop helper computing technology for enhancing reliability and security of computer systems. In helper computing, relatively autonomous “helper” threads or processes execute extra code on behalf of the application on separate processors or thread contexts. In the past, the use of helper threads was constrained to prefetching and branch prediction. In this work, we propose exploring a new and novel use of helper computing for improving software reliability and security. With helper computing, reliability and security functionalities that are normally performed as parts of the application code are offloaded to the helper thread/process. This enables sophisticated functionalities to be computed in parallel with the application without slowing down the application much. We already pointed out three important problems disturbing our helper computing technology such as granularity of data communication, load balance, and delay of data communication. Therefore, we propose two techniques in this work to solve the problems. First, in order to solve the problems regarding
granularity of data communication and load balance, we propose Automatic Parallelization for Fine-Grained Meta-Functions on a CMP. Second, we propose HAQu: Hardware-Accelerated Queueing for Fine-Grained Threading on a CMP, which is a technique to reduce fine-grained queueing overheads in multithreaded programs as well as to support OS transparency. It will be the solution for Problem 3.

1.4.1 Automatic Parallelization of Fine-Grained Meta-Functions on a Chip Multiprocessor

Software's increasing reliance on meta-functions present a promising source of parallelism. The predominate approach in performing metafunctions is to interleave their execution with the main application code, through direct augmentation of the metafunction code into the application code. Such an interleaved execution approach invariably increases the critical path of the main application, sometimes increasing it by orders of magnitudes.

The trend in microprocessor design is toward many-core systems. The performance of a single core in a many-core system will increase at a slower speed than in the past, and in some cases will stagnate. In order to exploit the performance of such a system, programmers need to exploit parallelism at various levels in their applications. Only recently has parallelism been exploited for performing metafunctions, such as memory management [20, 50], and bug detection [46, 33]. However, these techniques mainly focused on long-running, coarse-grained meta-functions.

In this work, we focus on automatic extraction of fine-grained meta-functions for parallel execution on multi-cores. Our goal is the development of a compiler which can efficiently extract meta-function level parallelism from programs in a way that provides low latency execution and is scalable on multi-core systems available today or in the near future. One of the primary challenges for fine-grained meta-functions are the abundant instructions inserted throughout the main application. Reducing this fine-grained interleaved overhead is difficult to accomplish while still providing scalability. Our strategy is the generation of customized helpers which follow the call context of the main program and replicate as much state as possible in a way that does not require communication. Using a smart task partitioning strategy, meta-function code and some of its dependences are migrated to the customized helper. This approach results in low overhead in the main thread since meta-functions and their support code need not be executed there. To provide scalability, we generate multiple customized helpers using static and dynamic load-balancing heuristics.

In our work, we make the following contributions:

- We propose a novel parallelization methodology using Custom Helpers for fine-grained meta-function parallelization.
To provide scalability, we describe static and dynamic heuristics to generate multiple helpers and balance their load.

We evaluate our framework on Mudflap, a pointer-use checking tool implemented in GCC. We run our experiments on an architectural simulator modeling an aggressive 8-core CMP with support for fast queueing operations. We show that our automatic technique is efficient and scalable; With static heuristics, it achieves an average $1.64 \times$ speedup using 8 cores. Compared to a manually optimized version of the Mudflap framework, our approach is only 19% slower. With dynamic heuristics, it achieves an average $1.96 \times$ speedup using 8 cores. Our automatically parallelized code using dynamic load balance is competitive, on average, to the manually parallelized version on a simulated 8-core system. Furthermore, our custom helpers reduce the overhead in the main thread by $5.3 \times$ compared to serial Mudflap.

1.4.2 HAQu: Hardware-Accelerated Queue

Chip multiprocessors (CMPs) are now ubiquitous, but achieving high performance on a wide range of applications is still a big challenge. It is important to consider new architectural features that can expand the programmability and performance of CMPs. In this work, we consider such a feature by adding architectural support for a single-producer, single-consumer queue.

Queues are a commonly used programming construct in multithreaded programs for synchronization and communication. They are effective when the granularity of parallelism is large enough to amortize the cost of enqueueing and dequeueing. But software queues tend to be too expensive to support fine-grained parallelism. For this reason, hardware queues have been proposed to reduce the overhead of communicating between cores. However, hardware queues tend to be unattractive to designers. They come at a large design cost as they require modifications to the processor core and need a custom, global interconnect. They also pose difficulties for the operating system because their state must be preserved across context switches. These challenges grow with ever increasing processor virtualization.

To solve these problems, we propose a hardware accelerated queue, or HAQu (read like haiku). HAQu adds hardware to a CMP that accelerates operations on software queues. Rather than adding a custom hardware queue, our design implements fast queueing through an application’s address space with operations that are compatible with a fully software queue. Our design provides accelerated and OS-transparent performance in three general ways. (1) It provides instructions for enqueueing and dequeueing which significantly reduce the overhead of fine-grained threading. These instructions are critical for achieving fast queueing since they eliminate the high instruction count involved in queueing operations. (2) Operations on the
queue are designed to leverage low-level details of the on-chip caches and coherence protocol to provide fast communication. (3) Furthermore, the full state of the queue is stored in the application’s address space, thereby providing virtualization, queues in nearly unlimited quantity, and queues of a large size. On the whole, HAQu is no worse for programmability than a software queue.

We have evaluated our design in three contexts. First, we evaluate our mechanism using a micro-benchmark designed to show the peak performance of our architecture compared to state-of-the-art queueing algorithms in software. Our system is able to achieve a 6.5× speedup over an efficient software implementation proposed by Lee et al [23]. Secondly, we show how our design can facilitate fine-grained parallelization by parallelizing Mudflap, a pointer-checking utility built into GCC. Using HAQu, we can afford to parallelize fine-grained, inlined pointer checking codes. Using 16 cores, we achieve an average speedup of 1.8×, calculated using geometric mean, and a max speedup of 3.3× on a set of SPECint applications. Finally, we demonstrate the potential of HAQu for automatic parallelization using decoupled software pipelined (DSWP). On a small set of SPECint applications, we show that our mechanism enables fully automatic parallelization.

In summary, this work makes the following contributions:

- We propose HAQu a hardware accelerated queue. HAQu is unique in its ability to accelerate fine-grained queueing operations on a CMP in a way that is highly programmable and OS transparent. We have described in detail its overall design and microarchitecture.

- We describe two case studies in fine-grained parallelization using HAQu: fine-grained parallelization of pointer-checking codes and decoupled software pipelining. We discuss optimizations necessary to achieve high performance in this systems.

- We evaluate our proposed queue in three contexts: on micro-benchmarks that show its performance potential, on a parallelized implementation of Mudflap, and on decoupled software pipelining.

1.5 Report Organization

The rest of the report is organized as follows. Chapter 2 presents automatic parallelization of fine-grained meta-functions on a chip multiprocessor. Chapter 3 presents HAQu, Hardware-Accelerated Queueing for fine-grained threading on a chip multiprocessor. Chapter 4 shows evaluation of the proposed schemes. Chapter 5 concludes this work.
Chapter 2

Automatic Parallelization of Fine-Grained Meta-Functions on a CMP

2.1 Fine Grained Meta-function Parallelization

Meta-functions are extra functionality added to a program to augment its behavior in some useful way. While they can be an arbitrary function, we restrict their definition, somewhat, to cover common uses and to limit the scope of our study. We assume that meta-functions have the following properties. (i) They are placed at a program point, \( p \), to compute some desired property at that location. (ii) They take as input State\(_p\) that is extracted from the main program state and MetaState\(_{p_in}\) that is information calculated specifically by the meta-functions. A critical simplifying assumption is that MetaState is disjoint from State. (iii) They produce as output an optional error state, Error\(_p\), and optional updates to the meta-function state, MetaState\(_{p_out}\). Note that none of these meta-functions update State; we do not allow meta-functions to update program state in order to narrow the scope of this work.

We will assume that most meta-functions fall into two categories: validating meta-functions and updating meta-functions. A validating meta-function checks a property of MetaState\(_{p_in}\) without modifying it and determines if State\(_p\) contains an error. This is essentially a complex check of some dynamically invariant property. An updating meta-function uses State\(_p\) to transition MetaState\(_{p_in}\) to MetaState\(_{p_out}\) and may detect an error related to the update operation. For example, in pointer-use checking, an important validating meta-function is testing that a dereference occurs to memory that has been allocated and not freed; an example of an updating meta-function is tracking when an object is allocated or freed so that the database of known memory objects is kept up-to-date.
Table 2.1: Characterization of Mudflap overheads normalized to an optimized baseline without it.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2</td>
<td>11.1</td>
<td>19.2</td>
<td>51.8</td>
<td>14.2</td>
<td>25.3</td>
<td>1988.2</td>
<td></td>
</tr>
<tr>
<td>gap</td>
<td>6.6</td>
<td>9.7</td>
<td>21.1</td>
<td>5.4</td>
<td>53.6</td>
<td>1646.0</td>
<td></td>
</tr>
<tr>
<td>gzip</td>
<td>3.9</td>
<td>6.2</td>
<td>52.0</td>
<td>0.0</td>
<td>26.7</td>
<td>2011.6</td>
<td></td>
</tr>
<tr>
<td>mcf</td>
<td>5.6</td>
<td>15.7</td>
<td>55.1</td>
<td>0.0</td>
<td>34.1</td>
<td>1945.4</td>
<td></td>
</tr>
<tr>
<td>parser</td>
<td>7.1</td>
<td>11.0</td>
<td>15.9</td>
<td>22.3</td>
<td>39.7</td>
<td>1871.3</td>
<td></td>
</tr>
<tr>
<td>twolf</td>
<td>6.1</td>
<td>10.0</td>
<td>12.7</td>
<td>12.3</td>
<td>53.3</td>
<td>1664.5</td>
<td></td>
</tr>
<tr>
<td>vpr</td>
<td>5.4</td>
<td>8.2</td>
<td>25.6</td>
<td>22.7</td>
<td>33.1</td>
<td>1994.7</td>
<td></td>
</tr>
<tr>
<td>Geo.Mean.</td>
<td>6.2</td>
<td>10.7</td>
<td>28.7</td>
<td>13.7</td>
<td>36.5</td>
<td>1868.7</td>
<td></td>
</tr>
</tbody>
</table>

There are several costs associated with inserting meta-functions into a program. First and foremost, the meta-function must be executed as part of the main program, thereby adding considerable runtime overhead. This instruction payload can increase runtime by an order of magnitude. Second, the inlining of code and function calls worsen the compiler’s ability to optimize the main program. As a result, the efficiency of the main application is much worse than it would be without meta-function instrumentation. The first two effects are clearly discernible from direct inspection of the generated code. Thirdly, the additional code competes with the main program code for critical processor resources, like issue slots, space in the cache memory hierarchy, or space in branch predictor history tables.

Table 2.1 shows some key overheads from Mudflap, a pointer-use checking tool. All percentages are shown relative to a binary without any Mudflap support. This data was collected using the same experimental setup described in Section 4.1. The fraction of updating and validating meta-functions combined is approximately 42% (Check + Other); this fraction represents time spent in the computational region of meta-functions. Additional inlined-code makes up another 36% (Hash) of the program. The total increase in instructions, on average, is 10.7×.

A naive way to parallelize meta-function workloads is by re-writing the meta-functions to explicitly leverage parallelism. Such a meta-function level approach to parallelization is straightforward to achieve. In such a parallelization strategy, each meta-function would decide the best way to parallelize it’s task. For example, it may send all of its inputs to waiting helper threads that are ready to receive and process them. At a minimum, it will need to send State_p. With frequent meta-function operations, the main thread will spend considerable time forwarding tasks to helpers. This approach can achieve good performance, but it requires significant programmer effort and cannot exploit redundancy or eliminate inefficiency that may exist across meta-functions within the same program.
2.1.1 Our Parallelization Approach: Custom Helpers

In our approach, we want to extract meta-function code out of the main program and run it in parallel. By automatically extracting meta-function code to helper threads, we can construct highly tuned, custom helpers that match the main program’s needs. Looking at each input to a meta-function provides some insight on how to do this. Consider State\(_p\). The same state may be needed by multiple meta-functions. Through code analysis, the compiler can generate a custom helper which receives the input value once and uses it in each meta-function call. Furthermore, rather than communicating just the inputs to meta-functions, main can communicate values for a code slice that the helper uses to generate the full State\(_p\) for one or more meta-function calls. By sending a only a few inputs for a slice, some work in the main thread can be eliminated and communication with the helper can be reduced. Location\(_p\) can also be handled more efficiently. If a custom helper is generated for a specific region of code, then some inputs can be embedded directly in the helper without any need for communication.

In summary, our compiler will generate a custom helper thread that is highly aware of the main program. It would be possible to generate a custom helper for the entire program (essentially a reduced program clone) or just for parts of it. So that we do not need to choose regions for parallelization, we generate a custom helper for the entire main program. The helper will mimic the behavior of the main program by matching its call stack dynamically. Relationships among multiple meta-functions will be exploited through a task selection algorithm that aggressively moves or replicates code from the main thread into the custom helper.

2.1.2 Providing Scalability

Given the large computational overhead introduced through meta-functions, multiple threads are needed to successfully parallelize them. For example, as shown in Table 2.1, there is no hope of significantly reducing the overhead in bzip2 with only one helper thread. Ideally, performance should scale as additional resources are added; once enough resources are added, the impact on the main program should be minimal. Therefore, we support multiple custom helpers.

In order to support multiple threads, meta-function parallelism must be exposed to the compiler. A general way of expressing and operating on such parallelism is beyond the scope of this work. Currently, we make decisions about which meta-functions can run in parallel at compiler compile-time to narrow the scope of the work. In other words, they are fixed for a given set of meta-functions. To decide which meta-functions can run in parallel, we leveraged profiling information and our understanding of the meta-function implementation as much as possible\(^1\).

\(^1\)This is non-trivial to perform automatically, but the implementer of such meta-functions can determine parallelization properties with reasonable effort.
To provide scalability, multiple custom helpers can be generated for the same function. Each custom thread will receive part of the meta-function workload of a function. In order to use multiple helpers as efficient as possible, load balance must be considered. Therefore, we provide simple static algorithms and show that they work reasonably well. The two heuristics we consider are random assignment and round robin. However, load balance is challenging with only a static algorithm for a variety of reasons. First, meta-functions may have dynamically varying execution times. For example, in order to reduce the time of meta-function execution, memoization is often used to significantly reduce overheads. As a result, the same meta-function can have wildly varying execution times. Second, the execution path of the program is unknown and can only be estimated. To properly balance load, you want the meta-functions that will actually execute to be distributed evenly. Finally, the structure of some program regions may not lend themselves to parallelization over many threads. For example, an inner loop may have one meta-function. If the loop is not unrolled and the trip count is high, then all executions of the inner loop body will be scheduled on the same helper resulting in imbalance. Analogous behaviors occur with many different code structures.

To compensate for all of these effects due to a static algorithm, we provide dynamic algorithms to balance workloads more evenly and show that the dynamic algorithms work better than the static algorithms. Our dynamic algorithms are based on broadcasting. We describe the detailed algorithms in Section 2.2.5.

2.1.3 Summary

In summary, our overall parallelization approach aims to reduce the overhead in the main application by extracting customized helpers that provide low overhead meta-function execution. However, to combat the high overheads of meta-functions themselves, we leverage multiple helpers to scale up to many threads and keep the overhead in the main program low. We will show that this approach is competitive with manual parallelization in our case study.

2.2 Parallelization Algorithm

We have designed a compiler algorithm that extracts and parallelizes meta-functions. In the explanation that follows, we start with the parallelization of a single function that includes meta-functions. Then, we expand it to support generation of the base helper thread which mimics the main thread’s behavior. Next, we describe how to use the base helper thread parallelization algorithm for custom helper thread extraction. Finally, we describe how to scale the algorithm to multiple threads using static and dynamic load balancing techniques.
2.2.1 Helper Function Parallelization Algorithm

**Algorithm 1: CreateHelper**

<table>
<thead>
<tr>
<th>Input:</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>f:</td>
<td>original function</td>
<td></td>
</tr>
<tr>
<td>P:</td>
<td>set of statements to move to helper thread</td>
<td></td>
</tr>
<tr>
<td>R:</td>
<td>set of statements to replicate in helper thread</td>
<td></td>
</tr>
<tr>
<td>tid:</td>
<td>thread id</td>
<td></td>
</tr>
</tbody>
</table>

**Result:** Customized helper function, helper, and modified f as a side-effect.

1. build PDG for f /*CDG + DDG*/ ;
2. $S = P \cup R$ ;
3. $A = \emptyset$ ;
4. foreach $s \in S$ do
5.    $A = A \cup \{m \mid m$ is an ancestor of $s$ in CDG subgraph $\}$ ;
6. end
7. $S = S \cup A$ ;
8. helper, map = Clone($f, S, tid$) /* create clone of $f$ restricted to $S$ */ ;
9. foreach stmt $s' \in$ helper do
10.    foreach use $u \in s'$ do
11.        MaybeInsertCommChannel ($u, f, helper, map$) ;
12. end
13. end
14. remove $P$ from $f$ ;
15. return helper

We base our parallelization algorithm for a single function on the Program Dependence Graph (PDG) [11]. We include all control, register and memory dependence edges in our PDG. The CreateHelper algorithm in Algorithm 1 shows the algorithm for creating the helper function.

The inputs to CreateHelper are the original function in SSA form ($f$), a subset of statements $P$ to include only in the helper, and a subset of statements $R$ to include in both. We assume that extracting $P$ and $R$ to the helper will not result in cyclic communication between the main program and the helper thread; rather, communication flows in a single direction from the main thread to the helper. CreateHelper assumes that $P$ and $R$ have been selected properly, so no such check is repeated here.

In order to move a statement to the helper, a few actions are required. First, we build a PDG [11] (a unified control and data dependence graph) for $f$. We set $S = P \cup R$ so that we have all statements to be included in the helper in one place. Next, all control predicates for each statement in $S$ must be included in the helper (lines 4-7). Once all the nodes in the
helper are computed, a clone of the function can be created with only the selected statements and control predicates present (line 8). For each input or predicate value, a communication channel, in the form of a queue, is added between the function and the helper (lines 9-12). In line 14, instructions in $P$ are removed from $f$.

MaybeInsertCommChannel has some smarts built into it that are not shown here. First, we check if $u$ really needs to be communicated: is $def(u)$ in $f$ but not in $helper$? This test covers three cases. Either the definition has been moved to $helper$, it has been replicated in $helper$, or it is only in $f$. We only communicate $u$ if it is not available in $helper$. If we do need to communicate $u$, we proceed as follows. Since $f$ is in SSA form, we insert a $send(u)$ just after its definition in $f$, and we replace its definition in $helper$ with $u = recv()$. We also keep a record that $u$ was communicated and never insert another channel for it between $f$ and $helper$.

One additional detail worth noting. Clone forces the new helper function to have a signature with no input parameters and no return value. While this is not strictly necessary, it simplifies our implementation since all helpers will have no inputs or outputs.

### 2.2.2 Baseline Helper Extraction

**Algorithm 2: BaseHelperThread**

- **Input**: $f$: original function in SSA form
- **Input**: $P$: set of nodes to move into helper
- **Input**: $R$: set of nodes to replicate in helper
- **Input**: $tid$: custom thread id

```plaintext
foreach stmts ∈ f do
  if s is call stmt then
    $R = R ∪ s$ ;
  end
end
$f' = CreateHelper(f, P, R, tid)$ ;
if $f$ is externally visible then
  insert ‘send(f)’ at entry of $f$;
end
return $f'$;
```

Our baseline helper is a clone of the entire main program that mimics the main application’s call stack. This ensures the helper is very lean and efficient by default. Whenever the main application calls a function the helper should call and execute an associated helper version.

The algorithm is shown in several parts. First, in the BaseHelperThread function (Algo-
gorithm 2), we accumulate all call statements into the $R$ set for parallelization. Next, we generate the helper with those statements extracted using the CreateHelper algorithm. CreateHelper works as previously described but we can now consider more of the details in the Clone function, shown in Algorithm 3. Clone will copy the function exactly, however, it will remove all statements that are not in $S$, and it will replace all call statements that are not meta-functions with a code sequence that will call their associated helper.

Algorithm 3: Clone

<table>
<thead>
<tr>
<th>Input:</th>
<th>$f$: function in SSA form</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input:</td>
<td>$S$: set of statements to be included in clone</td>
</tr>
<tr>
<td>Input:</td>
<td>$tid$: custom thread id</td>
</tr>
</tbody>
</table>

1. $f' = \text{clone of } f$ /* exact replica of $f$ but with a new name */ ;
2. $map = \text{CreateMap}(f,f')$ ;

3. foreach stmt $s \in f$ do
4.     $s' = map[s]$ ;
5.     if $s$ not in $S$ then
6.         remove $s'$ from $f'$;
7.     end
8.     if $s$ is a call && not a meta-function then
9.         ReplaceCall($s,s',tid$) ;
10.    end
11. end
12. $CHHT[f,tid] = f'$ ;
13. remove parameters and make void type ;
14. return $f'$ ;

Replacing the calls turns out to be more involved than it may appear at first. There are several cases which must be supported for this to work for arbitrary programs. First, indirect function calls through pointers can only be resolved at runtime. Therefore, the mechanism for determining which helper to run must occur at runtime (at least some of the time). Second, functions in libraries may not have a corresponding helper. Ideally, this should be prevented by providing a wrapper function for the library call that makes it compatible with our system, but this is not always possible. So, the external library call will not have a corresponding helper, but it could still call yet another function which does have an associated helper. The qsort function is a good example; a function pointer is passed as an argument and is used to help sort the data structure. Such a function passed as an argument is now externally visible. The qsort function may not have a helper, but its function argument likely will. While neither case is very common, both do occur with regularity in applications and must be supported.
Algorithm 4: CHHTLookup

\begin{algorithm}
\KwIn{$funptr$: function pointer to callee in main}
\KwIn{$tid$: custom thread id}
\If{$funptr$} {
  \If{$funptr == 1$} {
    $funptr = \mathrm{recv}()$
  }
  $newptr = \mathrm{CHHT}[funptr, tid]$
  $newptr()$
}
\Else {
  \Repeat{$funptr$} {
    $funptr = \mathrm{recv}()$
    \If{$funptr$} {
      $newptr = \mathrm{CHHT}[funptr, tid]$
      $newptr()$
    }
  }
}
\end{algorithm}

Custom Helper Hash Table. In order to dynamically bind a function with its customized helper, a Custom Helper Hash Table (CHHT) is constructed which creates a mapping between a function and its helper. Since we may generate multiple custom helper threads, it also needs a thread id as input.

The $CHHT\text{Lookup}$ function shown in Algorithm 4 is used at runtime to find the dynamic mapping of function to its custom helper. There are several cases that are required for correctness. We will consider each case in turn.

Direct and Indirect Function Calls. To support direct and indirect calls, at each call site we must determine the mapping from callee to helper thread. We can accomplish this with the $CHHT\text{Lookup}$ function by passing the address of the callee as an argument. In the case of the indirect call, the computation of the callee's address must be sent to the helper as a result of lines 9-12 in $CreateHelper$. For direct calls, the callee's address can be encoded directly in the helper. The relevant lines of code are 5-6 in Algorithm 4, and 9 in Algorithm 5.

Externally Visible Calls. Since we cannot modify all of the call sites of externally visible functions, we must communicate with the helper from inside the function rather than at the call site. At such a call site that we can analyze, we replace the call with $CHHT\text{Lookup}$ with a value of 1 as $funptr$ (6-7 in Algorithm 5). This will force $CHHT\text{Lookup}$ to wait until it receives one function pointer (line 2) from the called function; then it uses it to perform the

---

2To create the $CHHT$, the compiler builds a sequence of code to be executed at the beginning of the program that initializes the $CHHT$ and inserts all known mappings.
Algorithm 5: ReplaceCall

Input: s: original stmt in f
Input: s': cloned stmt in f'
Input: tid: custom thread id
Result: Replaces callee in s' with call to a helper function

callee = Callee(s) ;
if callee is external then
    replace s' with call to 'CHHTLookup(0,tid)' ;
    insert 'send(0)' after s ;
else
    if callee is externally visible then
        replace s' with call to 'CHHTLookup(1,tid)' ;
    else
        replace s' with call to 'CHHTLookup(callee,tid)' ;
end
end

lookup and call the corresponding helper function.

When BaseHelperThread analyzes the externally visible function, it inserts the 'send(f)' code (lines 7-9 in Algorithm 2).

External Calls. If an externally defined function (like sort) is called and there is no wrapper function available, then it may call externally visible functions which do have helpers. In this case, we replace the external call with a call to CHHTLookup with a 0 as its funptr argument. This will force the helper to wait until it receives a function pointer from an externally visible function (lines 7-14 in Algorithm 4). However, because it may never call such a function or it may call such functions repeatedly, we must identify when CHHTLookup terminates by sending a null pointer after the external call ends (2-4 in Algorithm 5).

Optimized Call Sequence. This mechanism works for all function calls, but it is sometimes inefficient. With additional information about the function, these costs can be avoided. For example, in the case of a static function in C, we need not call CHHTLookup at runtime to determine the mapping. Instead, we can hardcode the helper function’s location directly into its caller and allow function inlining to further optimize the code. We do not show it in our algorithm, but we do take this additional step in our full implementation.

2.2.3 Custom Helper Extraction

In Custom Helper Extraction, meta-functions and their support code are extracted from a function in the main thread and scheduled in the customized helper thread. We assume that a function, f, is already annotated with all meta-functions and supporting code, and we assume
that meta-function calls can be identified and inserted into a set, $M$, of statement nodes. Using only $M$ as the basis for helper extraction would yield an inefficient extraction with all meta-function inputs communicated explicitly to the helper. Instead, we take a smarter approach and look for additional statements to include in $M$ that will reduce the communication overhead between $f$ and the helper.

**Algorithm 6: CustomHelper**

<table>
<thead>
<tr>
<th>Input: $f$: function in SSA form</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input: $M$: set of meta-function nodes to move into helper</td>
</tr>
<tr>
<td>Input: $tid$: custom thread id</td>
</tr>
<tr>
<td>Result: Custom helper function</td>
</tr>
</tbody>
</table>

 repeat 
 1 foreach $s \in M$ do 
 2     foreach use $u \in s$ do 
 3         if all uses of $u$ are in $M$ then 
 4             $M = M \cup \text{def}(u)$ ; 
 5         end 
 6     end 
 7 end 
9 until $M$ converges ; 
10 return $\text{BaseHelperThread}(f, M, \emptyset, tid)$ ;

Our algorithm is shown in Algorithm 6. Using SSA form, we efficiently iterate over every statement $s$ in $M$. For each use $u$ in $s$, if $u$ is only used by statements in $M$, then the definition of $u$ can be added to $M$ as long as it adds no extra communication (or even reduces it). The extraction phase finishes once all members of $M$, both new ones and ones added along the way, have been considered. This algorithm will see to it that any data only computed for the meta-function is included in $M$. Finally, we use the $\text{CreateHelper}$ function to parallelize the statements in $M$ by setting the input argument $P = M$. Note, this algorithm will never create cyclic communication between $f$ and the helper.

Interestingly, a valuable benefit of this technique is the extraction of context sensitive constant values. For example, many meta-functions take code location as an argument. For a given custom helper, code location amounts to a constant value which will easily be extracted by the algorithm since it is only used in the meta-function.
void foo()
{
    int x, y;
    x = 1;
    /* Static function call */
    x = goo(x);
    /* Predicate node */
    if(x==1)
    {
        /* Stmts for meta-functions */
        y = x + 3;
        meta_check(y, __FILE__, __LINE__);
    }
}

(a) Original version

void foo()
{
    int x, y;
    /* Stmt only in Main */
    x = 1;
    x = goo(x);
    send(x);
}

(b) Main thread version

void foo.helper()
{
    int x, y;
    /* Replace static function call */
    CHHTLookup(goo,1);
    /* Communication for data */
    x = recv();
    if(x==1)
    {
        y = x + 3;
        meta_check(y, __FILE__, __LINE__);
    }
}

(c) Helper thread version (tid is 1)

Figure 2.1: A simple example of our Helper Extraction.
2.2.4 Example

Figure 2.1 shows a simple example of our Helper Extraction. Figure 2.1(a) shows the original function. Using our algorithms, we generate functions for the main thread (Figure 2.1(b)) and the helper thread (Figure 2.1(c)), respectively. This example illustrates how our algorithm replaces a call, communicates a predicate node, and places the computation of some inputs only in the helper thread. Statement "\(x = \text{goo}(x)\)" is a direct call which must be preserved in the main thread and replaced in the helper with \(\text{CHHTLookup}\), as shown in Fig. 2.1(c). Next, consider the predicate node with the expression \((x==1)\). Since a meta-function depends on this predicate, it must be preserved, and our algorithm is smart enough to communicate \(x\) and compute \((x==1)\) in the helper. Note that the main thread sends the predicate value using \("\text{send}(x)\)" and the helper thread receives the predicate value using \("x=\text{recv}()\)". Both operations occur at the same program point as the original definition of \(x\). Finally, operations that support constant inputs of meta-functions are only placed in the helper. \(_\text{FILE}_\) and \(_\text{LINE}_\) are two such examples. However, since \(y\) is an input, we must communicate \(x\) in order to compute \(y\). In this case, our algorithm is smart enough not to send \(x\) twice.

2.2.5 Supporting Multiple Helpers

First, to generate \(N\) custom helpers, we use a static load balancing algorithm to create \(N\) unique partitions from the \(M\) set. Then, for each tid and \(M_{tid}\) from 1 to \(N\), we use the \(\text{CustomHelper}\) algorithm to generate a custom helper. We have experimented with two static load balancing techniques to create the \(N\) partitions of \(M\): round robin and random. For our round robin approach, we traverse the CFG in DFS order and assign meta-functions to a partition using round robin. This is a simple approach that we have found to work well. We also considered random assignment. We use the same basic algorithm as round robin but place meta-functions into threads randomly. Random assignment can avoid imbalance caused by systematic interactions between the code structure and round robin assignment. More sophisticated approaches are possible, but we have found that these are competitive in our current implementation.

Second, to generate \(N\) custom helpers, we use a dynamic load balancing algorithm. The \(M\) set is replicated to all the custom helpers instead of partitioning. We have experimented with dynamic load balancing techniques. We dynamically assign work into each helper thread. At compile time, our framework extracts meta-function code from the main program and schedule them on all the customized helper threads. At runtime, we dynamically allow only one helper thread to execute the meta-function. Figure 2.2 shows the generated main function and helper functions for a loop with a large number of trip count using dynamic load balancing techniques. As seen in Figure 2.2(a), main thread sends data to all the helper threads. As seen in Fig-
loop( ) loop( )
{ { data = recv(1); data = recv(1); idx = HASH(data); idx = HASH(data); if (idx == 1) if (idx == 1) {
    meta_check(data, ...); meta_check(data, ...); }
}

(a) Main (b) Helper 1 (tid = 1)

loop( ) loop( )
{ { data = recv(2); data = recv(3); idx = HASH(data); idx = HASH(data); if (idx == 2) if (idx == 3) {
    meta_check(data, ...); meta_check(data, ...); }
}

(c) Helper 2 (tid = 2) (d) Helper 3 (tid = 3)

Figure 2.2: Generated main and helper functions with Dynamic Load Balancing using HASH.
Figure 2.2(b), (c), and (d), meta-functions from the main program are extracted and schedule on all the customized helper threads. However, we use a simple $HASH$ function to select only one helper thread to execute the meta-function. With dynamic load balancing using $HASH$ function, we expect similar benefits of address indexing of manual parallelization described in Section 2.3.2.

```
loop( )
{
  data = x;
  /* Broadcasting for dynamic load balancing */
  cnt1 = cnt1%3 + 1;
  if (cnt1 == 1)
  {
    meta_check(data, ...);
  }
  send(1,data);
  send(2,data);
  send(3,data);

  ... = x[idx];
}

(a) Main
```

```
(b) Helper 1 (tid = 1)
loop( )
{
  data = recv(1);
  cnt1 = cnt1%3 + 1;
  if (cnt1 == 1)
  {
    meta_check(data, ...);
  }
}
```

```
(c) Helper 2 (tid = 2)
loop( )
{
  data = recv(2);
  cnt2 = cnt2%3 + 1;
  if (cnt2 == 2)
  {
    meta_check(data, ...);
  }
}
```

```
(d) Helper 3 (tid = 3)
loop( )
{
  data = recv(3);
  cnt3 = cnt3%3 + 1;
  if (cnt3 == 3)
  {
    meta_check(data, ...);
  }
}
```

Figure 2.3: Generated main and helper functions with Dynamic Load Balancing using counters.

Figure 2.3 shows another dynamic load balancing technique using local counters. As seen in Figure 2.3(a), main thread sends data to all the helper threads. As seen in Figure 2.3(b), (c), and (d), meta-functions from the main program are extracted and schedule on all the customized helper threads in the same way as shown in Figure 2.3. However, we use a local counter
(cnt1, cnt2, and cnt3) for each helper thread instead of a simple HASH function to select only one helper thread. Because a local counter of each helper thread is incremented in a same way, we dynamically allow only one helper thread to execute the meta-function. With dynamic load balancing using local counters, we expect similar benefits of round robin of manual parallelization described in Section 2.3.2.

While we expect to balance work more evenly on each helper, we also insert more instructions in main thread by sending data to all the helper threads. With fine-grained communications among all the threads, it will add more overheads into main. In addition, by extracting all the meta-function code and scheduling them on all the customized helper threads, it will add more communication for predicate nodes among the threads because predicate nodes should be communicated with all the helper threads. With static load balancing, because each meta-function is extracted and scheduled to a specific helper thread, predicates nodes were not communicated with all the helper threads. In Section 4.1, we show trade-offs between static load balancing and dynamic load balancing due to these behaviors. Furthermore, we provide a couple of techniques to make dynamic load balancing techniques more efficient in Section 4.1.

### 2.3 Case Study: Mudflap

Mudflap [9] is a set of passes and libraries integrated into GCC [49] to facilitate pointer use checking. The Mudflap pass in GCC transparently adds error checking code at the dereferencing of any pointer to validate the memory access against a database of allowed memory regions in the stack, heap, static variable section, and others.

In this section, we consider how to support Mudflap parallelization in two ways. We will discuss how Mudflap maps into our automatic parallelization framework and discuss some synergistic benefits we discovered. Also, we will describe our strategy for manually parallelizing it. In the evaluation, we will compare these two approaches.

<table>
<thead>
<tr>
<th>Meta-function Name</th>
<th>Inputs</th>
<th>Insertion Point</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_mf_register</td>
<td>Pointer to buffer, size, location</td>
<td>Any memory allocation point</td>
<td>Record allocated memory region in a database</td>
</tr>
<tr>
<td>_mf_unregister</td>
<td>Pointer to buffer, location</td>
<td>Any memory deallocation point</td>
<td>Remove previously allocated region from database</td>
</tr>
<tr>
<td>_mf_check</td>
<td>Pointer, ref. size, location</td>
<td>All memory references</td>
<td>Validate reference is for a registered region of memory</td>
</tr>
<tr>
<td>inlined software cache lookup</td>
<td>Pointer, size</td>
<td>All memory references</td>
<td>Checks a software cache to see if reference has already been validated</td>
</tr>
</tbody>
</table>
int *x; int size = 2;
...
x = WRAP_MALLOC(sizeof(int)*size);
/* the real malloc is wrapped to register
the allocated region with libmudflap */
...
if( __MF_CACHE_MISS(x+0,4) )
  __mf_check(x+0,4);
x[0]=3;
if( __MF_CACHE_MISS(x+2,4) )
  __mf_check(x+2,4); /* Error! */
x[2]=3;
WRAP_FREE(x); /* unregister */
...
if( __MF_CACHE_MISS(x+1,4) )
  __mf_check(x+1,4); /* Error! */
..=x[1];

Figure 2.4: Example code with Mudflap annotation.

2.3.1 Mapping Mudflap to our Framework

Table 2.2 shows the meta-functions used by the Mudflap algorithm. _mf_register is an updating meta-function added at every memory allocation point to record a new memory region that can be accessed by the program. Such memory allocation points include malloc, stack allocation, and global variables. _mf_unregister is an updating meta-function that is inserted at all deallocation points. And, _mf_check is a validating meta-function that checks if a memory reference falls within an allocated region. Figure 2.4 provides an example of a program annotated with Mudflap’s meta-functions.

We have studied the behavior of Mudflap to determine the appropriate strategy for parallelizing these meta-functions. Since _mf_register and _mf_unregister are relatively infrequent and since the database they create is relatively small, they are always replicated across all helpers. _mf_check, on the other hand, is very frequent and must be balanced across the helpers to provide scalable performance improvement. Table 2.1 from Section 2.2 backs up these decisions. GCC provides a library, called libmudflap, that implements these metafunctions. We modified this library to support a parallel implementation. In particular, we changed the code so that a database could be created and maintained per thread.

Since the C library is not compiled with Mudflap annotations, library functions that manipulate pointers must be handled as a special case. Libmudflap already provides wrapper functions for many libC functions that allocate memory. For the case of malloc and free, a wrapper function performs the register or unregister, respectively, on the allocated or freed region. We modified all such wrapper functions manually to provide a customized helper and added them in the CHHT.
Benefits for Mudflap. Our framework is expected to provide several useful benefits for Mudflap. Looking at the inputs for each metafunction, it is clear that two of the inputs are likely constants: size and location. These inputs will be extracted directly into the custom helper. The pointer must be communicated, but if it is used in multiple meta-functions it may only need to be communicated once. Finally, the inlined software cache lookup is directly inserted into the program to prevent redundant calls to `mf_check`. These inlined codes add significant instruction and control flow overhead and all of these can be moved to the helper.

Our framework also provided some unexpected benefits. Local aggregates need to be registered with Mudflap. However, when a helper is created, we initially copy all local variable declarations into the helper. Since the helper’s copy is not the same memory location as the one in the main program, the local copy in the helper need not be registered with Mudflap. But, we found it beneficial to do so. By registering the helper’s local aggregate, many meta-functions could be executed using the local aggregate without ever receiving input from the original function. An out-of-bounds check in the helper on a copy is equivalent to a check in the original function. This optimization reduced communication and, thereby, lightened the
load on the main program. (N.B. We still register the address of the original aggregate from the main program just in case its address is ever taken.)

2.3.2 Manual Parallelization

In order to evaluate how well our automatic technique works, we also manually parallelized Mudflap for comparison. In this implementation, we just re-wrote the meta-functions to exploit parallelism explicitly.

The meta-functions implement a pipelined architecture as shown in Figure 2.5. For each meta-function in Mudflap, we created a parallel version which packs up its input and enqueues them to Queue0. Each meta-function inserts a special header at the beginning of its input so that it can be parsed properly by the Middle Thread. Also, we modified \_\_mf\_check to include the software cache lookup. This modification significantly reduces the code injected into the main program. Also, to make \_\_mf\_check a little more efficient, we omitted the debugging location information. For every pointer use check, we insert this alternative implementation of \_\_mf\_check.

The Middle Thread parses the contents of Queue0 and decides how to handle the meta-function. It must send \_\_mf\_register and \_\_mf\_unregister requests to all helpers. For \_\_mf\_check requests, it applies a load balancing heuristic and sends the request to one of the Helper Threads for completion. Because the performance of the system is very sensitive to the processing rate of the Middle Thread, the load balancing heuristics are kept simple and fast. We considered two dynamic load balancing techniques: address-based balancing and round robin.

**Parallelization 1: Address indexing.** Because the Helpers use a software cache to filter out redundant checks, we prefer to send redundant checks to the same core so they will be filtered quickly. We use a simple hash function on the input address to select a queue.

**Parallelization 2: Round-robin.** We also evaluate the Middle thread using a round-robin method to select a helper thread. Using this approach, allocation is more uniform, and the Middle thread is less likely to block on a full queue.

Our round robin algorithm sends 8 consecutive operations to the same Helper. By sending 8 at a time, we exploit locality in the access stream; consecutive checks are more likely to refer to the same object in the Mudflap database. We determined a chunk size of 8 through experimentation.

2.4 Related Work

**Helper Threads.** Many works have improved single-thread performance by using helper threads [25, 42, 50, 54, 55]. In [25, 42], to tolerate memory bottlenecks, the helper threads perform pre-execution with idle hardware resources. Xekalakis et al. [55] combined helper
threads with Thread-Level Speculation and Runahead execution. Tiwari et al. [50] used a helper thread to accelerate memory management.

**Parallelizing Security Meta-functions.** While many security protection mechanisms have been proposed, few have investigated exploiting parallelism between the application and its security meta-functions, which is the focus of this proposal. Nightingale et al. proposed Speck [33], which exploits speculative parallelism between the application being protected with its security meta-functions. Shetty et al. [41] and Kharbutli et al. [20] parallelize memory management functionality for increased security. Plakal et al. [38] proposed parallelizing garbage collection using program slices on multiprocessors. However, they did not implement the slices, rather they only evaluated the potential.

**Thread-Level Speculation.** One alternative way to exploit parallelism between the application and its meta-functions is to use thread-level speculation (TLS). TLS allows frequent data sharing at low cost, but without a guarantee that the speculative work will be useful. Examples of work in bug detection with TLS include [34, 57, 19]. All studies employ thread-level speculation to hide the overhead of program checking. It has been applied in security by Speck [33], as discussed previously.

**Memory Bugs.** Detection of memory bugs also can be performed statically by utilizing explicit model checking [29, 47] and program analysis [4, 10, 14]. As software becomes more complex and diverse, users increasingly rely on run-time bug detection by instrumenting the code with checks, such as in Purify [40], Valgrind [44, 45], Intel thread checker [18], DIDUCE [15], Eraser [43], CCured [31], and others [2, 7, 24, 36, 37]. Such instrumentation typically introduces large performance overheads because instrumented memory references (loads and stores) are executed often, and execution of the instrumentation code is interleaved with normal program execution.

Hardware support for detecting memory bugs or to support debugging has been proposed, such as iWatcher [57], AccMon [56], SafeMem [39], DISE [6], and the scheme proposed by Oplinger et al. [34]. All three schemes (iWatcher, AccMon, and DISE) interleave application execution with bug checking, address matching, or debugging functions. In contrast, our helper computing mechanisms attempt to exploit parallelism between the helpers and the main application execution.

**2.5 Summary**

Due to the importance of reliability and security, prior studies have proposed inlining meta-functions into applications for detecting bugs and security vulnerabilities. However, because these software techniques add fine-grained instrumentation to programs, they often incur large runtime overheads by reducing the effectiveness of code optimization on the original program.
and by adding large overheads to compute the meta-function workload. In this work, we consider an automatic thread extraction technique for removing these fine-grained checks from a main application and scheduling them on helper threads. The proposal provided 1) Baseline Helper Extraction and 2) Custom Helper Extraction using Helper Function Parallelization Algorithm. Baseline Helper Extraction makes all helper threads follow the same call sequence of main thread. Custom Helper Extraction extracts all the meta-functions code and their support code from the main program and schedule them on multiple helper threads. With Baseline/Custom Helper Extraction, meta-functions and their support codes were eliminated from the critical path. Also, we insert data communication code into the main and helpers. Our automatic parallelization framework inserts data communication efficiently to support concurrent run of the main program with meta-functions. In addition, we proposed static and dynamic load balancing techniques to improve our automatic parallelization framework. With our load balancing algorithms, we expect to reduce overheads due to unbalanced workloads among helper threads.
Chapter 3

HAQu: Hardware-Accelerated Queueing for Fine-Grained Threading on a Chip Multiprocessor

3.1 Main Idea: Hardware Accelerated Queue

3.1.1 Motivation

Queues that support fine-grained parallelism on chip-multiprocessors must meet several criteria. We have identified four that would make a queue acceptable for a wide range of uses. Throughout this article, we restrict our discussion to single-producer/single-consumer queues. First, (Criterion I) enqueue and dequeue operations must be made as efficient as possible. Since these operations may occur frequently, it’s necessary that the total latency is low. Second, (Criterion II) the delay through the queue (i.e. the time between an enqueue and dequeue of the same item) must be low enough that concurrent execution proceeds efficiently. Third, (Criterion III) they should be no worse to program than software queues. For example, there should be enough queues available to support a wide variety of applications, and their synchronization constraints should be no harder to understand than for fully software implementations. Finally, (Criterion IV) the queue must work seamlessly on an unmodified operating system (OS).

The current state-of-the-art implementations for queueing are fully in software. Figure 3.1 shows an implementation of a Lamport queue designed for sequential consistency; part (a) shows the queue definition, and part (b) and (c) show the enqueue and dequeue operations respectively. The enqueue and dequeue operations shown here are non-blocking and lock-free; a blocking implementation is trivial to construct using these operations. For weaker consistency models, a fence is needed to properly read and update the queue state.
struct queue {
    Item *head;
    Item *tail;
    uint size; // = SIZE;
    Item buffer[SIZE];
}

typedef struct queue * queue_p;

(a)

int enqueue(queue_p q, Item val) {
    Item *next = NEXT(q->head);
    if (q->tail == next)
        return BLOCK;
    *head = val;
    // memory barrier
    q->head = next;
    return SUCCESS;
}

(b)

int dequeue(queue_p q, Item *val) {
    Item *tail = q->tail;
    if (tail == q->head)
        return BLOCK;
    // memory barrier
    *val = *tail;
    q->tail = NEXT(tail);
    return SUCCESS;
}

(c)

Figure 3.1: Lamport's queueing operations single-producer, single-consumer queue on a sequentially consistent processor.
While these implementations are not well suited for fine-grained parallelism, they do meet most of the criteria above. In particular, they provide low latency for communicating through the queue because they often exploit the fast on-chip interconnect. This latency can be optimized through careful consideration of on-chip caches and the coherence protocol. They meet Criterion III trivially because they are implemented in software. Criterion IV is met trivially as a result of Criterion III; since the queues are stored in virtual memory, they are trivially virtualized by any OS. Software queues, however, do not provide enqueue and dequeue operations that are efficient enough for fine-grained parallelism; so, they do not meet Criterion I. Even with efficient non-blocking and lock-free implementations for enqueue and dequeue, the cost of these operations tend to be high — 10 instructions per queue operation.

Hardware queues have appeared in a variety of proposals [41, 35, 21]. Due to the similarities, we will treat them the same in this discussion. These designs support Criterion I and Criterion II. Using special instructions or special hardware, data is inserted directly into the hardware queue, and this makes enqueue and dequeue highly efficient. In fact, the latency of enqueue or dequeue can be hidden by out-of-order execution. Furthermore, the storage for the queue is provided by special buffers and an interconnect between two or more cores. This provides very low latency communication. However, these mechanisms are costly, and there are practical concerns over their programmability and the limits on the number of queues that can be supported (Criterion III). Furthermore, these hardware queues have not been designed with the OS in mind, and often do not consider the challenges of context virtualization. For example, how is the state of the queue preserved when one of the threads, either the producer or consumer, is context switched. Furthermore, allocation of global queueing resources may need to be managed by the operating system. Given the complexity of a modern OS, hardware that requires such changes in the OS is considerably less desirable.

Neither software nor proposed hardware mechanisms meet all of the criteria for fine-grained threading on current CMPs. However, the shortcomings of these software and hardware mechanisms point the way to a hybrid design that leverages strengths from each one.

3.1.2 Our Proposal: Hardware-Accelerated Queue

The Hardware-Accelerated Queue (HAQu) aims to exploit the critical benefits of hardware queueing while retaining the many desirable features of software queues. In a sense, we start with a software implementation and speedup the features that are critical for faster and more efficient fine-grained threading. Using hardware to accelerate queueing is not new to this proposal. The VAX [16] also supported hardware accelerated queues (see comparison in Section 3.4). However, we will show that HAQu provides a simple, lean implementation that integrates well into modern superscalar architectures; and it can provide much higher throughput and lower
latency compared to optimized software implementations.

HAQu provides two new instructions for enqueue and dequeue operations to meet Criterion I. These new instructions keep the code footprint low, and their implementation will work much like software enqueue and dequeue operations. However, because they are implemented in hardware, we incorporate detailed knowledge of the memory consistency model, coherence protocol, and on-chip resources to provide a highly optimized enqueue or dequeue operation.

HAQu stores the queue’s state in the application’s address space. As a result, Criterion III and Criterion IV nearly trivially are met. This means that we can support a large number of queues efficiently, and we can design HAQu to provide synchronization semantics that programmers expect. Also, since the state of the queue is held in virtual memory, a context switch poses almost no problems since the queue need not be saved or restored. Furthermore, HAQu can leverage the existing on-chip interconnect and cache coherence protocol for transferring data from producer to consumer. This usually provides low enough latencies to meet Criterion II.

Arriving at an efficient implementation is not as straightforward as it may seem. Criterion I and Criterion II are difficult to obtain without leveraging queue specific information in the hardware. We find that some information about the queue must be tracked in hardware. However, we can do so only if we preserve OS transparency and programmability concerns. Our architecture for HAQu balances the competing concerns of achieving a high-throughput implementation (Criterion I and Criterion II) while still providing an implementation that can be deployed on conventional CMPs that support a wide range of applications (Criterion III and Criterion IV) and programmability concerns.

3.2 System Architecture

To explain the overall architecture of our proposal, we will begin with a software queue and modify it to support enqueueing and dequeueing operations in hardware. There are three main components to this discussion: the allocation and layout of the queue in memory, enqueue and dequeue operations, and making these operations efficient for fine-grained threading.

3.2.1 Allocation and Layout

Our queue will reside in memory but will be operated on directly using hardware. Therefore, the layout of the queue is determined a priori\(^1\). For now, we will assume a layout similar to Figure 3.1(a). This is not an optimized configuration but is sufficient for the following discussion\(^2\). A client will allocate a queue using a memory allocator like malloc (in C) and

\(^1\)This is a limitation of our proposal, but a handful of layout configurations could be supported and configured with firmware with little additional cost.

\(^2\)An optimized implementation will pad the head and tail to fit on their own lines to prevent false sharing.
prepare it for use. The head, tail, size, and buffer fields of the queue are located at fixed offsets from the beginning of the queue. We presume that the queue data structure and its initialization can be provided through a runtime library. Furthermore, since the buffer is the last field of the struct, the queue can be of arbitrary size.

### 3.2.2 Enqueue and Dequeue Operations

A key goal of HAQu is reducing the cost of enqueue and dequeue operations by adding two new instructions to carry out these tasks. A naive design directly implements Lamport’s algorithm [22] in hardware.

Consider an enqueue operation. When an enqueue instruction is executed, it is passed the base address of the queue as an input argument. Using the base address, the head and tail fields of the data structure are read. From this point on, hardware implements Lamport’s algorithm.

Figure 3.2 shows a possible timeline for the execution of queue operation. Since enqueue and dequeue operations are similar, we only describe the enqueue operation in detail. In cycle 1, the head and tail are loaded into temporary registers (these are not architectural registers or part of the register file). For this example, we assume that both memory operations hit in the L1 cache and return 2 cycles later. In cycle 3, we perform the test to validate that the queue is not full. In this timeline, we assume it was not and proceed with cycle 4 to store the value in the queue. We will refer to this store as the enqueueing-store since it actually places the value in the queue. (Similarly, we refer to the operation in cycle 4 for dequeue as the dequeueing-load.) At this point, we have to wait for the store to complete before updating the head to the next position. Note, this store ordering is required for Lamport’s algorithm to function properly and without races.

The enqueue and dequeue operations are shown in Figure 3.3. Since we target a MIPS architecture in our evaluation, these instructions are shown using MIPS-like mnemonics and implementation nuances. We support variants of these operations for each size of memory load or store supported in MIPS to ensure efficient code generation. Each enqueue instruction takes two input arguments: (1) the base address of the queue, and (2) the data to be enqueued. Dequeue operations only require the base address of the queue as input. Non-blocking instructions return an error code in their destination register.

### 3.2.3 Making Fine-Grained Queueing Efficient

So far, our design does not consider the interaction of queue operations with the cache coherence protocol or on-chip caches. It is critical to do so in order to achieve high throughput using our queue. Consider the implications of the proposed timeline in Figure 3.2 in a state-of-the-art CMP when used for fine-grained threading. First, if enqueueing and dequeueing operations are
Figure 3.2: Enqueue (a) and dequeue (b) operations. Assumed cache hits with a dual-ported, 2-cycle access L1 cache.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Enqueue Operations</th>
<th>Dequeue Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>t0 = MEM[head]</td>
<td>t0 = MEM[head]</td>
</tr>
<tr>
<td>1</td>
<td>t1 = MEM[tail]</td>
<td>t1 = MEM[tail]</td>
</tr>
<tr>
<td>2</td>
<td>Access cache</td>
<td>Access cache</td>
</tr>
<tr>
<td>3</td>
<td>NEXT(t0) != t1</td>
<td>t0 != t1</td>
</tr>
<tr>
<td>4</td>
<td>MEM[t0] = val</td>
<td>val = MEM[t1]</td>
</tr>
<tr>
<td>5</td>
<td>Wait for completion</td>
<td>Wait for completion</td>
</tr>
<tr>
<td>6</td>
<td>MEM[head] = NEXT(t0)</td>
<td>MEM[tail] = NEXT(t1)</td>
</tr>
</tbody>
</table>

(a) (b)

Figure 3.3: Enqueue and dequeue instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>benq</td>
<td>Rs, QAddr Enqueue Rs into QAddr, block on queue full.</td>
</tr>
<tr>
<td>bdeq</td>
<td>Rd, QAddr Dequeue from QAddr to Rd, block on queue empty.</td>
</tr>
<tr>
<td>enq</td>
<td>Rt,Rs,QAddr Enqueue Rs into QAddr, Rt returns status.</td>
</tr>
<tr>
<td>deq</td>
<td>Rt, Rd,QAddr Dequeue from QAddr to Rd, returns status in Rt.</td>
</tr>
</tbody>
</table>
happening concurrently, then the head and tail variables for a queue are being read frequently by two different cores. This results in frequent transitions between exclusive state and shared state for the cache line holding each variable. This will significantly lengthen the time needed to decide if an enqueue or dequeue can proceed. Second, even in the case where the threads are not operating concurrently, the queue operations that can occur per unit time are determined by the amount of ILP that can be extracted from these instructions. The amount of parallelism is limited by the number of ports on the cache, the cache access time, and by the degree of ILP provided in the microarchitecture for enqueue and dequeue operations. Overall, these effects could serve to drastically reduce the throughput of our mechanism. To improve the efficiency of our mechanism, we adopt a few key measures: relaxed memory ordering for queue operations, reduced accesses to shared variables, and pipelined queue operations.

**Relaxed Memory Ordering for Queue Operations**

To enhance the parallelism available in the microarchitecture, we will assume a system with weak memory ordering, and we allow queueing instructions to be re-ordered dynamically as long as they operate on different queues. Since each queue operation actually involves multiple memory accesses, we are allowing more aggressive reordering of memory operations than would be available in a software implementation. This scenario is shown in Figure 3.4(a). The first enqueue and dequeue operations on Q1 must be properly ordered, otherwise FIFO ordering may not be preserved. This is indicated through the ordering arrows. However, the operations on Q2 can be re-ordered with respect to those on Q1, and this is shown by the lack of arrows to the enqueue or dequeue on Q2. For queueing operations on different queues that must occur in a given order, a memory fence is necessary to guarantee their order.
struct queue {
  Item *head;
  Item *tail;
  uint size; // = SIZE;
  Item buffer[SIZE];
}

typedef struct queue * queue_p;

int hw_enqueue(queue_p q, Item val) {
  if (!QLT.find(q)) {
    QLT[q].PC = Consumer;
    QLT[q].LTail = q->tail;
    QLT[q].LHead = q->head;
    QLT[q].QSize = q->size;
    QLT[q].Cnt = 0;
  }
  Item *next = NEXT(QLT[q].LHead);
  if (next == QLT[q].LTail) {
    QLT[q].LTail = q->tail;
    if (next == QLT[q].LTail)
      return BLOCK;
  }
  *next = val;
  QLT[q].LHead = next;
  QLT[q].Cnt++;
  if (QLT[q].Cnt==CHUNK_SIZE) {
    // memory barrier
    q->head = QLT[q].LHead;
  }
  return SUCCESS;
}

{int,Item} hw_dequeue(queue_p q) {
  if (!QLT.find(q)) {
    QLT[q].PC = Producer;
    QLT[q].LTail = q->tail;
    QLT[q].LHead = q->head;
    QLT[q].QSize = q->size;
    QLT[q].Cnt = 0;
  }
  if (QLT[q].LTail == QLT[q].LHead) {
    QLT[q].LHead = q->head;
    if (QLT[q].LTail == QLT[q].LHead)
      return {BLOCK, 0};
  }
  // memory barrier
  Item val = * QLT[q].LTail;
  QLT[q].LTail = NEXT(QLT[q].LTail);
  QLT[q].Cnt++;
  if (QLT[q].Cnt == CHUNK_SIZE) {
    // memory barrier
    q->tail = QLT[q].LTail;
  }
  return {SUCCESS, val};
}

Figure 3.5: Algorithm for reducing accesses to shared variables.
Furthermore, the relative order of other memory operations with respect to queueing operations are also relaxed. In Figure 3.4(b), a flag is set to 1 before the enqueue in Thread A, and the flag is read in Thread B after the corresponding dequeue. In this case, there is no guarantee that Thread B observes the value 1. However, this can be enforced using a fence before the enqueue in Thread A and after the dequeue in Thread B, as shown in Fig 3.4(c).

If fences are needed between queueing operations, they must be inserted into the code. If a programmer is directly coding using HAQu, then she is responsible for inserting the fence. However, a parallelization framework or language may use these queues, and in that case, the compiler or a runtime library can provide automatic fence placement.

**Reduced Accesses to Shared Variables**

We modify our queueing algorithm so that we no longer need to access shared variables on each enqueue or dequeue operation. This simplification can eliminate the need to access the cache in advance of the enqueueing-store or dequeueing-load.

Our optimized enqueuing and dequeuing algorithms are shown in Figure 3.5 and are inspired by Lee et al [23] (abbreviated as P.Lee from here on). We adopt an innovation that avoids accesses to the shared head and tail through use of a local copy of the head and tail that is private to each thread. In our algorithm, these private variables are located in a hardware table, called the Queue Local Table (QLT), associated with each queue. (In P.Lee’s approach, these values would be held in general purpose registers; we are simply moving them to dedicated registers.) The QLT and its fields are shown in Fig 3.5(b). Consider the case of enqueuing in Figure 3.5(c). Most of the time, the head of the queue is tracked using \( \text{QLT}[q].\text{LHead} \) (LHead for short), which is the head of the queue as stored in the QLT for queue, \( q \). As long as LHead has not caught up with LTail, there is room to enqueue and the value of LHead is correct to use. Afterwards, the enqueueing-store is performed and LHead is updated.

We can also avoid the final update to the head of the queue. In this case, we can delay the update of head until enough items are enqueued so as to amortize the cost of coherence transactions on head and tail. The \texttt{CHUNK\_SIZE} denotes the number of enqueues needed and this value is dependent on the block size of the cache and, ideally, is a multiple of the cache block size. We will discuss the interaction of this policy with the memory consistency model in Section 3.2.4. Also, we discuss a potential deadlock caused by this policy in Section 3.3.

The dequeueing operation is analogous to that of the enqueue. However, it is worth pointing out that the private copy of LHead and LTail are not shared by the producer and consumer. If a single core were both enqueuer and dequeuer, a separate table entry for enqueueing and dequeueing operations would be needed.
Pipelined queue operations

We decompose queue operations into three stages: *address generation, queueing-op, and the index-update*. Figure 3.5 shows the stages for the enqueue and dequeue operations between parts (b) and (c). In address generation (AG), the location of the *enqueueing-store or dequeueing-load* is calculated. With the availability of the *LHead* and *LTail*, generating multiple addresses in a single cycle is feasible. This enables high throughput issue of enqueue or dequeue operations. The *queueing-op* (QOp in the figure) performs the operation on the queue’s buffer in memory. It is treated like a regular load or store for dequeueing and enqueueing, respectively. Consequently, this stage is executed by the processor as efficiently as it would a load or store operation and under the same constraints. The QOp phase must wait until the proper stage in the pipeline to execute the load or store operation. Finally, the *index-update* stage (IU) occurs last to bring the state of the *head or tail* up to date. If multiple queue operations on the same queue are ready to retire in a single cycle, the IUs can be merged into a single update.

3.2.4 Ensuring Programmability and OS Transparency

To achieve high performance, we have significantly altered the behavior of the queue as compared to its software version. For example, we buffer the head and tail in hardware and allow them to be reused by multiple queue instructions. Since these values are now cached in hardware, we must be careful to preserve a clear programming model and one that remains OS transparent. The key questions we need to answer are: (1) when and for how long can we keep the local head and tail values buffered, and (2) under what circumstances is it allowable to delay the index-update operation.

To better explain the problems, Figure 3.6 shows a few scenarios that are particularly important for the functioning of our mechanism. Fig 3.6(a) shows the case where a queue is used to communicate from Core 1 to 2, and then after some time, it reverses direction. At any point in time, there is only one producer and consumer, but there is a transition point between these modes of operation. By the time the producer and consumer have switched roles, both must have the same view of the queue state and that should reflect all prior queue operations. But, if we are not careful, two cores may be holding simultaneously a QLT entry for either consuming or producing, which would violate our assumptions; and, worse, we have compromised the programmability of the mechanism. Finally, Fig 3.6(c) shows the case of a context switch. Initially, Thread A is running on Core 1, but, eventually, it is context switched to Core 2. Thread A should have a consistent view of the queue after moving to Core 2. If this case fails to work, we have sacrificed OS transparency. The common requirement among these cases is that all cores must retain a consistent view of the queue data structure.

To make the cases in Fig. 3.6(a) and (c) function properly, we conservatively must guarantee
that all hardware state is flushed to memory before the producer or consumer switches cores.

Because we cannot automatically detect these cases in hardware, we rely on a memory fence to force all information in the QLT to memory. On a weakly ordered system, we require that any pending index updates must be completed before a fence or synchronizing instruction can retire. Furthermore, no new queue operations can begin until the fence completes. Figure 3.6(b) and (d) show the proper orderings after synchronizing these events properly. Note that in Fig. 3.6, an additional barrier operation (labeled as a sync) is needed to force ordering across threads, but this would also be required in an equivalent program using a software queue implementation; so, our requirements are no worse than the software queue case.

Because we basically relied on a memory barrier (fence) of existing operating systems to provide OS transparency, it is important to show that current existing operating systems use a memory barrier during context switch. Therefore, we describe why current existing operating systems use a memory barrier during context switch.

Many architectures require memory barrier to enforce memory ordering as described in [26].
In addition, memory barriers must be inserted during context switch because we can’t guarantee that all the cores will see memory operations in the same order when a thread migrates from a core to another [26]. Figure 3.7 shows an example. Figure 3.7(a) shows memory operations on a single core. Figure 3.7(b) and (c) show that thread A migrates from Core 1 to Core 2 after ST C is issued. Because Core 2 can’t see the recent memory operations in the same order as Core 1, thread A can be broken after migrating to Core 2 as seen in Figure 3.7(b). That is, because LD C and ST C are reordered on Core 2, thread A loads stale value through LD C. Therefore, context switch requires memory barriers to guarantee correctness. Figure 3.7(c) shows that memory barrier prevents a program from being broken by enforcing memory order during context switch. In Figure 3.7(c), memory barrier allows thread A to migrate from Core 1 to Core 2 after ST C is completed on Core 1. As a result, thread A is successfully context switched.

In addition, we studied context switch codes of Linux with latest stable kernel version
static inline void
context_switch(struct rq *rq, struct task_struct *prev,
       struct task_struct *next)
{
    struct mm_struct *mm, *oldmm;

    prepare_task_switch(rq, prev, next);

    mm = next->mm;
    oldmm = prev->active_mm;

    arch_start_context_switch(prev);

    if (!mm) {
        next->active_mm = oldmm;
        atomic_inc(&oldmm->mm_count);
        enter_lazy_tlb(oldmm, next);
    } else
        switch_mm(oldmm, mm, next);

    if (!prev->mm) {
        prev->active_mm = NULL;
        rq->prev_mm = oldmm;
    }

#ifndef __ARCH_WANT_UNLOCKED_CTXSW
    spin_release(&rq->lock.dep_map, 1, _THIS_IP_);
#endif

    switch_to(prev, next, prev);

    barrier(); // memory barrier

    finish_task_switch(this_rq(), prev);
}  

Figure 3.8: Context switch codes of Linux Kernel 3.2.6 [1].
3.2.6 [1] to see if current existing operating systems use memory barriers in their context switch process. Figure 3.8 shows context switch source code of linux kernel 3.2.6. As seen in Figure 3.8, context switch performs memory barrier through the call `barrier()` before finishing context switch through the call `finish_task_switch()`.

3.3 Implementation

3.3.1 Microarchitecture Description

To support HAQu, we add a Queue Functional Unit (QFU) to each core of a CMP, as shown in Figure 3.9. The QFU is made up of two structures. The Queue Local Table stores information about pending queue operations. In particular, it tracks the local head and tail variables described in Section 3.2.3. Also, it includes the Address Generation Buffer and the Queue Op Buffer (QOB) which contain a record of pending queue operations; we use them to order queue operations, determine the necessary updates to the head and tail variables, and recover from misspeculation. We now provide a brief overview of our micro-architectural mechanism. Since it involves straightforward changes to modern superscalar architectures, we will not discuss the logic in depth.

3.3.2 Pipelined Execution Description

**Address Generation Stage.** When a HAQu instruction is renamed, an entry is created for it in the Address Generation Buffer. The AGB contains all queue instructions (pInst in AGB) in program order. Instructions wait in this buffer until the source registers are available and have been read. Once the queue’s base address has been resolved, the queueing operation can begin.

Address generation proceeds by checking the QLT for a valid entry for the specified queue. If no such entry exists, a new one must be created by loading the head, tail, and size of the queue. We provide the QFU direct access to the cache to obtain the needed values. Once the head and tail are known, a new entry in the QLT is initialized and it is used to generate an address for the queue operation. The entry is set to valid (V bit in table).

**QOp Stage.** At this time, a store or load address can be generated for the enqueue or dequeue, respectively. We assume that we can generate multiple addresses in a single cycle. However, we need not generate more addresses than can be used to access the cache in a single cycle. Once the address is known, the information for the operation is moved to the QOB until it retires and completes. For an enqueue operation, the enqueueing-store cannot execute until it reaches the head of the re-order buffer. A dequeueing-load can execute as soon as its address is known.
Figure 3.9: Hardware Accelerated Queue added to a CMP.
Index Update Stage. When an instruction finally retires, we update several fields of the QLT. The RIndex field of the QLT is updated with the location in the queue it either read or wrote (LTail for enqueue or LHead or dequeue). This serves as a record of the last retired operation on the queue. The dirty (D) bit is set to indicate that it needs to eventually perform an index update. Also, the Cnt field is incremented; if Cnt is more than CHUNK_SIZE, the index update occurs immediately and Cnt is reset. Once the RIndex value is written to memory, the D bit is cleared.

Our policy to delay head and tail updates can introduce deadlock. If we wait until CHUNK_SIZE enqueues occur, the other thread waiting for a head update may be indefinitely stalled. Therefore, we add some additional logic that will force an update of a dirty entry after a fixed interval of time, $t_{update}$. When this timeout is triggered, we steal an idle cache-port and update the dirty entry. We use a separate timer for all entries in the QFU since it is a small table.

Memory Ordering Issues. Only a memory fence alters timing of queueing instructions relative to other memory operations. When a fence is decoded, a signal is sent to the QFU. The fence is then added to the AGB and tracked like other queue operations. The fence will remain in the AGB until all instructions in the QOB complete and all dirty entries in the QLT are flushed to memory and invalidated; and, queueing instructions that are added to the AGB after the fence must stall until the fence completes. Once the fence retires, it is removed from the AGB and queueing operations will begin again.

Note, we implement fence in this way to guarantee the ordering of operations discussed in Section 3.2.4. Indeed, our mechanism will suffer high overheads if fences occur frequently; however, in that case, the queues are unlikely to be the principal bottleneck due to limited re-ordering of memory operations.

Misspeculation and Precise Exceptions. Speculative execution and precise exceptions are properly handled. In the event of a branch misspeculation, the QLT must roll back to the last address assigned before misspeculation. After discarding speculative instructions from the QOB and AGB, the LHead and LTail in the QLT must be brought up to date with any pending non-retired instructions in the pipeline. This can be accomplished by walking the QOB and updating the QLT to the youngest values found there. In the event of an exception on any of the memory operations within the queueing instruction, we require the entire instruction be re-executed. In this way, all structures and resources associated with that instruction can be reclaimed precisely.

3.3.3 Managing QFU Resources

The QLT is a limited resource and can only support a limited number of entries. If the QLT is full with valid entries and a new queue is referenced, an action must be taken to reclaim
resources within the QLT. Fortunately, this case is easy to handle. If there is a non-dirty entry, it can be immediately displaced and claimed for the new queue. If all entries are dirty, then an entry is picked at random to displace (though it is best to choose an entry with no pending operations). To displace a dirty line, we must write back the RIndex to the queue’s head or tail depending on its status as either producer or consumer, respectively.

Because non-dirty lines can be evicted at any time, it is possible than an operation on a displaced queue entry is in the QOB but has not yet retired. In this case, the RIndex update simply writes directly to memory rather than updating the QLT.

3.3.4 Instruction Implementation Details

The queueing instructions are more complex than typical instructions in RISC processors today because they may need to access up to three memory locations: the head, tail, and buffer location. In the worst case, this requires access to three separate pages in memory and could generate three exceptions in the worst case. If the OS cannot guarantee multiple pages stay in memory, this could generate livelock. For OS compatibility and to reduce the potential complexity of these operations, we can require the head and tail to be allocated on the same page. Now, the instruction will access no more than two pages. This is no worse than the case of unaligned loads in current systems which access two pages.

3.4 Related Work

Software Queueing In order to support concurrent lock free queueing, researchers have proposed a variety of queues [27, 22, 12, 13, 23]. They have primarily focused on providing lock-free queueing algorithms which are cache conscious. More discussion of these queues is included in Section 4.2.2. Other work in software queueing has focused on multiple-producer, multiple consumer queues. Currently, HAQu does not support such queues, but it is a challenging problem and a focus of our future work.

Some software-only alternatives do exist for reducing overheads associated with fine-grained queueing; Tiwari et al. [51] used speculation to hide the latency and bulk enqueues to reduce the frequency of queue operations.

Queueing Support in the ISA The VAX [16] supported hardware accelerated queues similar in concept to HAQu. However, their implementation was considerably more complex than HAQu. The VAX instructions INSQHI, INSQTI, REMQHI, and REMQTI allowed insertion or removal at the head or tail of a self-relative doubly-linked list. These instructions were interlocked and could not be interrupted; this prevented other processors from updating the queue at the same time. These complexities made it very difficult to achieve high performance.
and programmers often resorted to implementing their own software queues. Hence, the VAX design failed to meet Criterion I and Criterion II. HAQu intentionally avoids these pitfalls by adopting a hardware design that mimics successful software queueing algorithms. Furthermore, HAQu’s design builds on top of the microarchitectural features that enable precise exceptions and branch mispeculation, thereby ensuring ease of integration in aggressive superscalar designs.

**Hardware Queueing** Hardware queues [41, 35, 21] have been proposed to support low latency between a producer and consumer. Heapmon [41] uses a hardware queue to offload the monitoring of heap memory bugs to a helper core. DSWP [35] leverages a high speed Synchronization Array to support fine grained parallelization. Carbon [21] uses a hardware queue to accelerate dynamic task scheduling. While these works meet Criterion I and Criterion II they do not consider Criterion III and Criterion IV (Section 3.1).

### 3.5 Summary

Queues are a commonly used programming construct in multithreaded programs for synchronization and communication. Queues are effective when the granularity of parallelism is large enough to amortize the cost of enqueueing and dequeueing. But, software queues tend to be too expensive to support fine-grained parallelism. To solve these problems, we proposed HAQu, which is high performing queueing systems on a CMP. In addition, HAQu is programmable and OS transparent systems. With our evaluation on NarrowRing and WideRing, we will show that HAQu achieve high throughput on next generation interconnects (on WideRing). In addition, we expect that HAQu matters even more with higher speed interconnects in the future. With HAQu, we believe that the potential for fine-grained parallelism on a CMP will be more improved.
Chapter 4

Evaluation

4.1 Evaluation of Automatic Parallelization

4.1.1 Experimental Setup

We have evaluated our design on SESC [17], an event-driven performance simulator, that accurately models CMP architectures. Table 4.1 shows the pertinent details of the simulated architecture. The core, L1, and L2 cache are common for all architectures we evaluated. The core is an aggressive 3-issue superscalar processor. We model a cache coherent CMP design with private L1 and L2 caches. Coherence is supported at the L2 cache over a ring interconnect.

For this study, we use highly efficient hardware queues for communication between threads. We assume 1 cycle to read/write from/to the queue. Also, we used 4 KB for each hardware buffer. We used hardware queues rather than software queues to showcase the effectiveness of our parallelization strategy.

Table 4.2 shows the different application binaries we study. For all of our application binaries, we used GCC 4.5 targeted to the Mips32 architecture. All of our compiler algorithms described in this work were implemented as a plugin for GCC. To accurately compare the performance of different binaries, simply timing a fixed number of instructions cannot be used. Instead, “simulation markers” are inserted in the code of each binary, and simulations are run for a given number of markers. After skipping the initialization, a certain number of markers are executed so that the binary graduates from 100 to 500 million instructions.

4.1.2 Baseline Helper Extraction

In this section, we evaluate the scalability of our BaseHelperThread extraction algorithm with several SPECint applications on a 4-core and 8-core CMP. Figure 4.1 shows the speedup over Base. CINT-G.M represents geometric mean normalized to Base over these applications. On
Table 4.1: Architecture simulated. All cycle counts are in processor cycles.

<table>
<thead>
<tr>
<th>8 Core CMP</th>
<th>ROB 126</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency 3.2 GHz</td>
<td>I-window 68</td>
</tr>
<tr>
<td>Fetch width 6</td>
<td>LD/ST queue 48/42</td>
</tr>
<tr>
<td>Issue width 3</td>
<td>Mem/Int/Fp unit 1/2/1</td>
</tr>
<tr>
<td>Retire width 3</td>
<td>Hybrid GShare &amp; Local</td>
</tr>
<tr>
<td>Branch predictor:</td>
<td></td>
</tr>
<tr>
<td>Mispred. Penalty 14 cycles</td>
<td></td>
</tr>
<tr>
<td>BTB 2K, 2-way</td>
<td></td>
</tr>
<tr>
<td>Private L1 Cache:</td>
<td>Private L2 Cache:</td>
</tr>
<tr>
<td>Size 64KB, assoc 4</td>
<td>Size 2MB, assoc 8</td>
</tr>
<tr>
<td>Line 64B</td>
<td>Line 64B</td>
</tr>
<tr>
<td>Latency 2 cycles</td>
<td>Latency 9 cycles</td>
</tr>
<tr>
<td>Protocol MESI</td>
<td></td>
</tr>
<tr>
<td>Memory:</td>
<td></td>
</tr>
<tr>
<td>Latency 300 cycles</td>
<td></td>
</tr>
<tr>
<td>Bandwidth 25.6GB/s</td>
<td></td>
</tr>
</tbody>
</table>

average, our proposal performs 8% worse than Base for both 4-core and 8-core CMP configurations. However, gap loses 30% and parser loses 20% while the others only lose between 2~5%. That is, Helper extraction is scalable for 8-cores with only 2~5% loss. gap and parser show a larger slowdown compared to the others because they have a large number of function calls. The calls themselves alone are not expensive, but in order to match the control path of the main thread, they require frequent communication of predicates which add a lot of overhead to main. With this overhead, gap and parser are limited in the parallelism they can support.

4.1.3 Custom Helper Extraction

In this section, we evaluate several characteristics of the parallelized Mudflap meta-functions of several SPECint applications on the 8-core CMP. Figure 4.2 shows the speedup over Mflap. For this evaluation, we use empty _mf_register, _mf_unregister and _mf_check meta-functions to show the quality of parallelization without the effects of runtime load imbalance. In addition, we compare our two static algorithms for load balancing. Looking at the overall geometric mean, our design achieves 4.5× speedup with random assignment and 4.6× speedup with round-robin assignment. This suggests that our extracted helpers are quite efficient. Furthermore, the performance has only been degraded by 26% over the results in Figure 4.1.

Figure 4.3 shows the speedups compared to Mflap running the full Mudflap implementation using both the automatic parallelization with static load balancing and the manual paralleliza-
Table 4.2: Configurations and parallelizations.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>No Mudflap,-O3</td>
</tr>
<tr>
<td>Mflap</td>
<td>Base + Mudflap</td>
</tr>
<tr>
<td>A-Random</td>
<td>Automatic Parallelization with static Random Assignment</td>
</tr>
<tr>
<td>A-RoundRobin</td>
<td>Automatic Parallelization with static Round-robin Assignment</td>
</tr>
<tr>
<td>DLB-Hash</td>
<td>Automatic Parallelization with dynamic load balance using Hash</td>
</tr>
<tr>
<td>DLB-Counter</td>
<td>Automatic Parallelization with dynamic load balance using Counter</td>
</tr>
<tr>
<td>M-Index</td>
<td>Manual Parallelization with Address Indexing Assignment</td>
</tr>
<tr>
<td>M-RoundRobin</td>
<td>Manual Parallelization with Round-robin Assignment</td>
</tr>
<tr>
<td>BHE4</td>
<td>Baseline Helper Extraction with 4 threads</td>
</tr>
<tr>
<td>BHE8</td>
<td>Baseline Helper Extraction with 8 threads</td>
</tr>
</tbody>
</table>

Overall, looking at the geometric mean, the automatically parallelized code is only 19% slower than the best manual technique. Furthermore, the automatic technique is performing competitively; mcf, twolf, and vpr achieve their highest performance using A-RoundRobin. mcf, twolf, and vpr achieve their advantage because their workloads are relatively balanced, and they perform even well using a static load balancing approach. This case highlights the importance of automatic meta-function extraction. However, bzip2, gap, gzip, and parser are under performers. gzip suffers from poor load balance; the manual cases achieve speedups within 25% of Base because of their aggressive load balancing strategies. bzip2 also suffers primarily from poor load balance. Finally, gap and parser perform poorly due to a combination of factors: poor load balance and the significant communication overhead needed to support function calls.

4.1.4 Dynamic Load Balancing

In Section 4.1.3, we show that unbalanced workload hurts the performance of automatic parallelization using static load balance in some applications of SPECint. Therefore, we provide and evaluate automatic parallelization using dynamic load balance described in Section 2.2.5 to see the benefits reducing the bottlenecks due to unbalanced workload. In order to estimate overhead due to unbalanced workload on each helper, we measured the number of meta-functions executed in each helper thread. Figure 4.4 and 4.5 show the number of \_mf\_checks executed in each helper thread with various load balancing techniques. Although we measured the numbers in various phases of programs, because they showed similar patterns, we show only one phase of each program and compare each load balancing technique.

Figure 4.4 shows workload of each helper thread in gap. While manual parallelization using round robin (a) has equally distributed workload, manual parallelization using address indexing
Figure 4.1: Scalability of Base Helper Extraction with 4-cores and 8-cores.

(b) and automatic parallelization using static round robin (c) have unbalanced workload. As a result, Figure 4.3 shows that gap achieve better performance with M-RoundRobin than M-index and A-RoundRobin. In this case, we believe that the middle thread of manual parallelization didn’t work well with address indexing. Figure 4.5 shows workload of each thread in gzip. In gzip, M-RoundRobin (a) and M-index (b) balance work very well. However, A-RoundRobin (c) shows unbalanced workload of each thread. As a result, Figure 4.3 shows that gzip performs better with M-RoundRobin and M-index than A-RoundRobin. While helper 7 in Figure 4.5(c) has a large number of _mf_checks compared to other helpers, because mudflap uses software cache to filter out redundant checks, most of _mf_checks of helper 7 are filtered out. That is, workload of helper 7 is not as large as seen in Figure 4.5(b).

Figure 4.6 shows speedups with static and dynamic load balancing techniques. In the figure, we compared dynamic load balancing techniques, labeled as DLB-hash and DLB-counter, to M-RoundRobin and A-RoundRobin normalized to Mflap. On average, dynamic load balancing using hash achieves 1.96× speedup over Mflap. Dynamic load balancing using counters achieves 1.56× speedup over Mflap.

As seen in Figure 4.6, on average, dynamic load balancing using hash is competitive to M-RoundRobin for manual parallelization. In bzip2, gap, and gzip, automatic parallelization using dynamic load balancing shows speedups over manual parallelization. DLB-hash expecting the benefits of M-Index of manual parallelization works well for bzip2 and gzip. Whereas, DLB-counter expecting the benefits of M-RoundRobin works well for gap. We expected the benefits
Figure 4.2: Speedups of Parallelized Mudflap with empty run-time functions.

because M-RoundRobin works better for gap than M-index of manual parallelization as seen in Figure 4.3.

As described in Section 4.1.3, mcf, twolf, and vpr achieve their highest performance using A-RoundRobin. Because A-RoundRobin evenly assigns workload to each helper thread for mcf, twolf, and vpr, they perform very well. However, DLB-hash and DLB-counter even hurt their performance because DLB-hash and DLB-counter adds inefficient instructions for communication among threads.

Although gzip shows a slowdown with automatic parallelization against manual parallelization, it is ignorable. That is, parser is the only one which shows a slowdown with automatic parallelization against manual parallelization. In Figure 4.6, we show that DLB-hash performs better than A-RoundRobin in parser. That is, DLB-hash balances workloads more evenly than A-RoundRobin. However, we also show that DLB-hash is much worse than manual parallelization. Measuring the number of dynamic instruction, we found that parser added large overheads due to frequent communication for predicate nodes and function calls in main and helpers. Measuring the number of dynamic instructions of main thread in parser, we found that Base Helper Extraction without instrumentation of Mudflap has already the similar number of dynamic instructions to that of manual parallelization. As a result, A-RoundRobin in parser shows a slowdown because of overhead due to communication among threads and unbalanced workload for each helper. Although DLB-hash is able to balance workload more evenly, the added overhead for dynamic load balancing becomes more critical in applications like parser
because it suffers from large overhead due to fine-grained communication. Therefore, in order to get better performance for applications like parser, inserting communications more efficiently is required along with load balance.

### 4.1.5 Optimizations

In this section, we provide and evaluate a couple of parallelization techniques to get better performance for each application. In addition, we also evaluate our framework on a 16-core system to show the scalability of our automatic parallelization framework.

**Pipelining communication** Measuring the number of dynamic instructions for each thread with dynamic load balance, we found that a large number of instructions for communication between main and helpers are additionally increased because our dynamic load balance is based on broadcast. Because more added instructions in main become more critical in some applications,
we pipelined communication instead of broadcast. That is, main sends data to the next helper. After the helper receives the data, it sends the data to a next helper. Helpers send the data to a next helper in a same way. Figure 4.7 shows speeds using pipelining communication. In the Figure, bzip2 and gzip show benefits of pipelining communication. As seen in Section 4.1.4, bzip2, gap, and gzip achieve better performance with dynamic load balance. We are able to expect the benefits because dynamic load balance adds a large number of instructions to main. However, we don’t get as good performance as bzip2 and gzip in gap because workload for each helper in gap is more irregular than the others. Whereas, because only one helper has large workload compared to others in bzip2, it can get more benefits to improve performance with pipelining communication.

**Combining static and dynamic load balance** Parallelizing each application with dynamic load balancing and static load balancing, we achieved the benefits of static and dynamic load balancing respectively. Therefore, we have experimented combining static and dynamic
load balance. Because parser didn’t perform well with automatic parallelization, we evaluate the technique with parser. By profiling the number of execution of each unique meta-function, we found the most executed unique meta-functions. With the profiled data, we use dynamic load balance for the most executed unique meta-functions.; we use static load balance for the rest of all the unique meta-functions. As a result, we achieve 6% speedup over dynamic load balance in parser. The gain is not so big. However, we can expect to achieve good performance with more efficient selection algorithms to choose meta-functions for dynamic load balance.

**On 4-core, 8-core, and 16-core CMP** We evaluate the scalability of our static load balance and dynamic load balance on a 4-core CMP, a 8-core CMP and a 16-core CMP. In Figure 4.8, A-RoundRobin4, A-RoundRobin8, and A-RoundRobin16 represent automatic parallelization with static load balance using round-robin assignment on a 4-core, 8-core, and 16-core CMP, respectively. In Figure 4.9, DLB-Hash4, DLB-Hash8, and DLB-Hash16 represent automatic parallelization with dynamic load balance using Hash on a 4-core, 8-core, and 16-core CMP, respectively. As seen in Section 4.1.2, our BaseHelperThread extraction algorithm works well as we increase the number of cores. In Figure 4.8, we can see the benefits for static load balance. In the figure, bzip2, gap, gzip, and mcf achieve better performance on a 16-core CMP. However, parser, twolf, vpr lose performance a little because it also depends on load balance. However, In Figure 4.9, we can not see the benefits for dynamic load balance by comparing DLB-Hash4, DLB-Hash8, and DLB-Hash16 because dynamic load balance inserts a large number of instructions among threads and the overhead becomes more critical. Furthermore, because dynamic load balance schedules meta-functions to all the helper threads
at compile time, all the control flows including meta-functions are replicated to all the helper threads. As a result, overhead due to communication for predicate nodes becomes more critical. That is, unless unbalanced workload is a dominant factor causing slowdown, dynamic load balance will not work well with more cores. In Figure 4.9, we show the benefit of DLB-Hash16 over DLB-Hash8 in bzip2 because unbalanced workload is a dominant factor causing slowdown of bzip2.

![Figure 4.8: Scalability of automatic parallelization using static load balance with 4-cores, 8-cores and 16 cores.](image)

4.1.6 Overheads

We have also measured some key overheads in our system. Figure 4.10 shows the relative size of the binary for four configurations: Base, Mflap, A-RoundRobin, and DLB-Hash. On average, Mflap is almost $1.71 \times$ bigger than Base. Our technique adds additional overhead due to Custom Helper creation. This overhead is 38% for A-RoundRobin and 100% for DLB-Hash respectively, on average.

We also measured the number of dynamic instructions in the main thread for Base, Mflap, M-RoundRobin, A-RoundRobin, and DLB-Hash. We can see that the parallelized versions do significantly reduce the overhead in the main thread. M-RoundRobin slightly edges out A-RoundRobin for two reasons: (1) the main program only communicates with one other thread which keeps its communication overhead low; and (2) we have eliminated debugging information from the \_mf\_check meta-function for manual parallelization. In contrast, A-RoundRobin must send inputs for \_mf\_register and \_mf\_unregister to all custom helpers. Furthermore, other data, like control predicates, also gets sent to multiple helpers. Currently, we do nothing to
reduce this overhead, but we plan to address these communication inefficiencies in future work.

4.2 Evaluation of HAQu

4.2.1 Experimental Setup

To evaluate HAQu, we have integrated our design into SESC [17], an event-driven performance simulator, that accurately models CMP architectures. Table 4.3 shows the pertinent details of the simulated architecture. The core, L1, and L2 cache are common for all architectures we evaluated. The core is an aggressive 3-issue superscalar processor. We model a cache coherent CMP design with private L1 and L2 caches. Coherence is supported at the L2 cache over a ring interconnect. HAQu is supported with a 16 entry QLT per core, and $t_{update}$ value of 500 cycles.

We support two different link bandwidths because our proposal is sensitive to interconnect design. The NarrowRing runs at 25.6GB/s per link and represents the capabilities of a currently available CMP; this is the default. We also model a futuristic interconnect, WideRing, with 102.4GB/s; we believe such a design could be on the market in the very near future.

For two studies, we will compare HAQu to an aggressive hardware queue. The hardware queue uses HAQu’s instructions but maps them into a high speed buffer connecting two cores. We assume 1 cycle to read/write from/to the queue. Also, we used 4 KB for each hardware buffer and allow the same number that HAQu can use (which is unlimited in theory but under 16 our studies).

Tables 4.4 and 4.5 show the different application binaries we study. (They will be described...
in more detail later.) For all of our application binaries, we used GCC 4.5 targeted to the Mips32 architecture. To accurately compare the performance of different binaries, simply timing a fixed number of instructions cannot be used. Instead, “simulation markers” are inserted in the code of each binary, and simulations are run for a given number of markers. After skipping the initialization, a certain number of markers are executed so that the binary graduates between 100 and 500 million instructions.

4.2.2 Micro Benchmarks

We wrote three micro benchmarks to evaluate the efficiency of HAQu compared to software queue implementations. The three algorithms selected are Lamport’s [22], FastForward [12], and P.Lee’s [23]. For each one, we kept the queueing data structure as similar as possible and minimized false sharing by placing the head, tail, and data buffer in different cache lines. The key difference in these implementations are the degree of false sharing they allow and how they check for empty and full status. Lamport’s queue suffers much more false sharing than FastForward or P.Lee’s because head and tail are referenced on each enqueue or dequeue operation. P.Lee’s queue is most similar to the HAQu implementation. FastForward differs from the others because it uses the value stored at the next entry to decide if the queue is empty/full. For example, if the next entry is non-zero, it can be dequeued, otherwise it must wait. This eliminates contention for a shared head and tail.

In each run, enqueue and dequeue are executed 500 times in a loop with 200 iterations, and we used 4-byte elements for each queue operation. In total, we transport 400,000 bytes.
Table 4.3: Architecture simulated. All cycle counts are in processor cycles.

<table>
<thead>
<tr>
<th>No. of Cores: 2 to 16</th>
<th>ROB 126</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency 3.2 GHz</td>
<td>I-window 68</td>
</tr>
<tr>
<td>Fetch width 6</td>
<td>LD/ST queue 48/42</td>
</tr>
<tr>
<td>Issue width 3</td>
<td>Mem/Int/Fp unit 1/2/1</td>
</tr>
<tr>
<td>Retire width 3</td>
<td>Hybrid GShare &amp; Local</td>
</tr>
<tr>
<td>Weak Ordering Consistency Model</td>
<td></td>
</tr>
<tr>
<td>Branch predictor: Mispred. Penalty 14 cycles</td>
<td></td>
</tr>
<tr>
<td>BTB 2K, 2-way</td>
<td></td>
</tr>
<tr>
<td>Private L1 Cache: Size 64KB, assoc 4</td>
<td>Private L2 Cache: Size 2MB, assoc 8</td>
</tr>
<tr>
<td>Line 64B</td>
<td>Line 64B</td>
</tr>
<tr>
<td>Latency 2 cycles</td>
<td>Latency 9 cycles</td>
</tr>
<tr>
<td>Protocol MESI</td>
<td></td>
</tr>
<tr>
<td>Memory: Latency 300 cycles</td>
<td>Bandwidth 25.6GB/s</td>
</tr>
</tbody>
</table>

Ring interconnection:
- NarrowRing Link Bwd (8 data bytes/cycle): 25.6GB/s
- WideRing Link Bwd (32 data bytes/cycle): 102.4GB/s

QLT Size: 16 entire
$t_{update} = 500$ cycles

Table 4.4: Mudflap configurations.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>No Mudflap,-O2</td>
</tr>
<tr>
<td>Mflap</td>
<td>Base + Mudflap</td>
</tr>
<tr>
<td>HAQu</td>
<td>Parallelization with HAQu</td>
</tr>
<tr>
<td>HWQ</td>
<td>Parallelization with Hardware Queue</td>
</tr>
<tr>
<td>SWQ</td>
<td>Parallelization with Software Queue</td>
</tr>
</tbody>
</table>

Table 4.5: DSWP configurations with varied queues.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>No DSWP,-O3</td>
</tr>
<tr>
<td>HAQu</td>
<td>Base + DSWP with HAQu</td>
</tr>
<tr>
<td>HWQ</td>
<td>Base + DSWP with Hardware Queue</td>
</tr>
<tr>
<td>SWQ</td>
<td>Base + DSWP with Software Queue</td>
</tr>
</tbody>
</table>
enqueue or dequeue is inlined. For the software queues, memory fences are inserted for a weak ordered consistency model. For all the queues, we use queue buffers holding 1024 elements (4KBytes). For each micro benchmark, simulation begins after the producer and consumer reach a barrier at the beginning of the loop.

Figure 4.12(a) shows the throughput for each queue in terms of enqueues per cycle. Overall, HAQu achieves 12.5× speedup over Lamport’s queue, 7.0× speedup over FastForward, and 6.5× speedup over P. Lee’s. This is possible because it generates an enqueue operation with a single instruction while the other implementations require several, usually dependent, instructions. Figure 4.12(b) shows the dynamic instruction count for each queue; the other queues use 12× to 17× more dynamic instructions. Furthermore, there is little ILP in any of the queues, on average. Figure 4.12(d) shows the L2 miss rate which is mostly coherence misses for these runs, and Figure 4.12(c) shows the IPC. P.Lee has the highest IPC due to few coherence misses, but even it does not reach one instruction per cycle in our simulations. This shows that an aggressive superscalar cannot extract much queue-level parallelism from the optimized software queues, but it can with HAQu.

Fig. 4.12(d) shows some interesting behaviors. Both P.Lee and HAQu have very few read misses, and this is a direct result of caching a local copy of the head and tail and only updating it when necessary. Figure 4.12(d) also shows that HAQu avoids cache thrashing by updating the head and tail only after \texttt{CHUNK\_SIZE} updates. Its behavior is as good as P.Lee’s which it is modeled after. However, on the NarrowRing HAQu is not living up to its full potential. Figure 4.13(a) and 4.13(b) show throughput and IPC for the WideRing interconnect. With additional bandwidth, the speedup of HAQu soars to 25.5× that of Lamport’s queue and 13.0× over FastForward. In addition, the IPC is nearly as good as FastForward. Furthermore, if
Figure 4.12: Micro benchmarks using queues on NarrowRing.
Figure 4.13: Micro benchmarks using queues on WideRing.
we add prefetching support to HAQu, we take advantage of even more bandwidth and boost the speedup over Lamport’s queue to $29 \times$. HAQu is an interesting design on current models (NarrowRing), but it is even more valuable as interconnects scale to support higher bandwidth links.

Figure 4.13(c) and Figure 4.13(d) show the normalized dynamic power and the normalized energy delay product (EDP) respectively. Figure 4.13(c) shows that HAQu consumes a bit more dynamic power compared to the other queues. This is because HAQu is able to do much more work in a given unit of time. Due to its fast execution, HAQu has a significantly better EDP. In a power constrained system, we could match the power of a software queue by running the entire core a little slower but still provide a performance gain.

4.2.3 Pointer-use Checking Acceleration

Mudflap [9] is a pointer-use checking utility integrated into GCC [49]. When a program is compiled with Mudflap, error checking code is inlined at all memory references to validate the memory access against a database of allowed memory regions in the stack, heap, static variable section, and others. These checking codes add considerable overhead to any program that uses them [19]. It would be nice if, instead of adding these codes to the critical path of the program, they could be performed in a helper thread. Using HAQu, we offload these fine-grained checks to another thread. We adopt a pipelined architecture for parallelizing Mudflap as shown in Figure 4.14. Because address-check operations occur frequently, we want to keep the cost of enqueueing as low as possible. Therefore, we enqueue into a single queue which must be emptied quickly by the Middle Thread. The Middle Thread removes the requested operation from Queue0 and then assigns it to one of the Helper Threads for completion. We implemented parallelized versions with a software queue using P.Lee’s algorithm, a hardware queue, and HAQu as shown in Table 4.4.

In this section, we evaluate HAQu’s performance with several SPECint applications running the parallelized Mudflap algorithm on a 16-core CMP. We used P. Lee’s queue for the software queue. Figure 4.15(a) shows the speedup for each application over Mflap running on the NarrowRing design. Overall, SWQ shows the worst performance, while the HWQ achieves a $1.3 \times$ speedup over HAQu. HAQu uses many fewer processor cycles per enqueue, enabling it to use those cycles for the main application.

HAQu uses $24 \times$ fewer dynamic instructions and $2.6 \times$ fewer L2 misses than SWQ. The performance difference between HWQ and HAQu comes primarily from the lower bandwidth of the NarrowRing compared to the high throughput hardware queue. However, given more bandwidth, HAQu is very competitive with HWQ. Figure 4.15(b) shows that performance normalized to Mflap on the WideRing. Here, HAQu can achieve nearly equivalent speeds as
Figure 4.14: Parallelizing Mudflap using Hardware Accelerated Queue.
 Furthermore, the overall speedup compared to Mflap jumps up to $2.2 \times$.

On WideRing, our parallelized Mudflap is only $1.85 \times$ slower than serial execution without Mudflap. We would have liked to reduce this more, but most of the remaining performance gap comes from bottlenecks unrelated to queueing. A key problem is load imbalance among the helper threads. Due to the load distribution policy, one helper thread may be busy for a long time, and its queue may fill up and prevent the Middle Thread from making progress. We studied several load balancing algorithms and found that round-robin works the best, on average. All of our results are based on this policy. However, some applications perform better with dynamic load balancing algorithms.

4.2.4 Decoupled Software Pipelining

Decoupled Software Pipelining (DSWP) [35] is an automatic parallelization technique that partitions a program, usually just a loop nest, into a pipeline. Queues are inserted between
the pipe stages to serve as communication channels for live-ins and live-outs of each stage. To achieve good performance with this parallelization approach, the cost of communication must be small with respect to the work performed in each pipe stage. If fine-grained queueing can be made efficient and common, many loops with fine-grained parallelism could be parallelized using this technique.

We have implemented a fully-automatic, profile-directed DSWP pass in GCC to evaluate the potential of HAQu in a challenging fine-grained threading environment. We have not incorporated any speculation techniques into our implementation; pipe stages must be fully independent: no true, anti, or output dependences can cross stages. We select any loop for parallelization that appears to offer a speedup based on profiling data, and we generate a parallel version with two threads. For all communication between pipe stages, we assign a dedicated queue. After the parallelized region, the main stage must wait for the helper stage to finish. Table 4.5 shows the experimental configurations for DSWP.

Figure 4.16(a) shows speedups of each queue normalized to Base. In the figure, CINT-G.M. means SPECint geometric mean. Overall, HWQ achieves 10% speedup over Base. This speedup comes from mcf which achieves a 57% speedup over Base, and vpr which achieves 25% over base; bzip2 attains a modest speedup around 4%. The other applications except for gap do contain at least one parallelized loop that turns out to be unprofitable, which explains their degraded performance. HAQu is much better than the SWQ, but it achieves an overall slowdown of 10% on this configuration. Compared to the HWQ implementation, HAQu pays a much bigger latency to communicate between cores. This turns out to be very costly in DSWP because the main stage must wait for the helper stage to finish; this places queue latency on the critical path.

In order to distinguish the effects of latency and bandwidth, we also evaluate DSWP on the more aggressive WideRing interconnect. In this system, HAQu is able to leverage the higher bandwidth and break even with serial. mcf attains a 45% speedup, vpr reaches 5%, and bzip2 attains 1%. Furthermore, gzip, twolf, and parser are penalized less for their attempted parallelization than on the NarrowRing. Clearly, DSWP is sensitive to latency and bandwidth, and this poses some challenges for HAQu. However, considering the flipside, without a HAQu-like-mechanism, achieving good performance using a DSWP algorithm is very difficult. With HAQu’s modest architectural requirements, auto-parallelization techniques for challenging integer applications might be attainable on future systems.
Figure 4.16: Speedup of DSWP over Base using 2 cores.
Chapter 5

Conclusion and Future Work

In this dissertation, we developed helper computing technology for enhancing reliability and security of computer systems on a chip multiprocessor. In helper computing technology, helper threads execute meta-function code on behalf of the application. In this work, we pointed out three important problems disturbing our helper computing technology and proposed the solutions for the problems. In this Chapter, Section 5.1 concludes two proposals and Section 5.2 describes future work.

5.1 Conclusion

5.1.1 Automatic parallelization of fine-grained meta-functions

In this work, we consider an automatic thread extraction technique for removing these fine-grained checks from a main application and scheduling them on helper threads. In this way, we can leverage the resources available on a CMP to reduce the latency of fine-grained checking codes. In addition, we explore a variety of techniques to improve performance. We evaluate our parallelization strategy on Mudflap, a pointer-use checking tool in GCC. To show the benefits of our techniques, we compare them to a manually parallelized version of Mudflap. We run our experiments on an architectural simulator with support for fast queueing operations.

First, we provide the automatic parallelization framework to balance workloads for each helper through static load balance and dynamic load balance. On a subset of SPECint 2000, our automatically parallelized code using static load balance is only 19\% slower, on average, than the manually parallelized version on a simulated 8-core system. Furthermore, three applications achieve better speedups using our algorithms than with the manual approach. Also, our approach introduces very little overhead in the main program — it is kept under 100\%, which is more than a 5.3$\times$ reduction compared to serial Mudflap. Our automatically parallelized code using dynamic load balance is even competitive, on average, to the manually parallelized
Second, we evaluate a couple of parallelization techniques such as pipelining communications and combining static and dynamic load balance to achieve better performance. We found that pipelining communications works well for a couple of applications suffering from large overheads due to large number of dynamic instructions executed in main thread and combining static and dynamic load balance works well for a specific application which needs dynamic load balance for a few unique meta-functions.

Third, we evaluate our framework on 16-core systems to see the scalability of our framework. As a result, we found that our automatic parallelization using static load balance is scalable. However, our automatic parallelization using dynamic load balance doesn’t work well unless an application is dominated by unbalanced workloads because adding a large number of instructions to each thread for dynamic load balance becomes more critical than balancing workloads.

In this study, we show that our automatic parallelization framework achieves better performance than manual parallelization. In addition, our framework with a couple of optimizations based on each instrumented programs’ characteristics can achieve better performance.

5.1.2 HAQu: Hardware-Accelerated Queue

In this study, we proposed a hardware accelerated queue, or HAQu. HAQu adds hardware to a CMP that accelerates operations on queues in memory. Rather than adding a custom hardware queue, our design implements fast queueing through an application’s address space with operations that are compatible with a fully software queue. Our design provides accelerated and OS-transparent performance.

We evaluated HAQu on queueing micro benchmarks, the parallelization of Mudflap, and Decoupled Software Pipelining. Compared to FastForward [12] and Lee et al [23], HAQu achieved speedups of 7× and 6.5×. We found that HAQu can deliver high throughput and low latency queueing, and that it carried over to application level studies. The low instruction footprint of each queue operation frees the processor to perform other tasks, thereby enabling fine-grained parallelism. Furthermore, HAQu is able to scale-up and attain high throughput on next generation interconnects. In conclusion, HAQu is a modest architectural extension that can help parallel programs better utilize the interconnect and processing capability of future CMPs.

5.2 Future work

There are several things we suggest to address in the future. First, in order to generalize the benefits of our automatic parallelization framework, we suggest to apply our framework to various software systems. One example can be DIDUCE [15], which is software to detect bugs...
using automatic anomaly detection. However, in order to use our framework, meta-functions must be efficiently parallelized. There are several things to keep in mind. Above all, we recommend to use separate data structures among multiple threads to update and monitor objects at runtime.

Second, in Section 4.1, we showed that our automatic parallelization technique with dynamic or static load balance performed well for all the applications except parser. The reason why it did not work for parser is that our automatic parallelization framework inserts a large amount of data communication among threads. That is, for the applications, our automatic parallelization framework inserts more data communication codes than manual parallelization. Because our framework makes helpers follow same call sequence of main program, if there are many indirect calls executed at runtime, a large amount of data communication will be inserted among threads. In addition, if communicating predicate nodes is frequently needed, it will be costly to use our automatic framework. Therefore, even though detecting the regions and code generation for each thread are challenging, we suggest to distinguish suitable regions for manual parallelization or automatic parallelization and combine manual parallelization and automatic parallelization.

Third, as described in Chapter 3, HAQu is a queueing system accelerated by hardware to support fine-grained threading on a chip multiprocessor. However, HAQu provides single-producer and single-consumer in this work. Although single-producer and single-consumer is good enough for parallel applications based on communication or pipeline, if HAQu supports multi-producer and multi-consumer, it will be more useful because there are many systems requiring multi-producer multi-consumer queues. For example, for the dynamic task scheduling context, multi-producer and multi-consumer queues will be very useful. In order to support multi-producer and multi-consumer, there are several things to consider. The key challenging is enabling multiple producers and consumers to work locally on a chunk without it leading to deadlock/data corruption or other problem.
REFERENCES


