HARDWARE-LEVEL FINE-GRAINED THREAD MIGRATION

by

Mieszko Lis

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2014

Signature redacted

Author .....
Department of Electrical Engineering and Computer Science
August 18, 2014

Signature redacted

Certified by .....
Srinivas Devadas
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Signature redacted

Accepted by .....
Leslie A. Kolodziejski
Chairman, Department Committee on Graduate Theses

HARDWARE-LEVEL FINE-GRAINED THREAD MIGRATION

by

Mieszko Lis

Submitted to the Department of Electrical Engineering and Computer Science
on August 18, 2014, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

ABSTRACT

Although thread migration has long been employed to satisfy load-balancing goals in operating systems for symmetric multiprocessing hardware, the high cost of OS-mediated migration has made more fine-grained applications impractical. With only a few cores per processor, and high overheads due to moving threads across processors and loss of cache affinity, assigning threads to specific processor cores for long periods has remained the default strategy for ensuring maximum performance.

Massive-scale single-chip multiprocessors dramatically alter this picture. On-chip data transfer latencies—even across a 100+-core chip—rarely exceed tens of cycles, making the potential cost of thread migration as low as executing several instructions. At the same time, all cores are placed on the same die and typically share one last-level cache distributed on chip, obviating cache affinity concerns.

In this dissertation, we explore the limits of fine-grained thread migration by developing an autonomous mechanism for migrating threads implemented entirely in hardware. We then employ migration to implement the unified shared memory abstraction without a cache coherence protocol—a particularly demanding application that requires fast and fine-grained thread movement—and show that performance is competitive with traditional shared memory mechanisms. Finally, we describe a real-world implementation of both concepts in a 110-core single-chip multiprocessor in 45nm ASIC technology.

Thesis Supervisor: Srinivas Devadas
Title: Professor of Electrical Engineering and Computer Science

ACKNOWLEDGEMENTS

Myong Hyon Cho and Keun Sup Shim joined me in the madness of building the EM² ASIC from design to tapeout. I never would have thought it possible that such a large ASIC could have been done by a team of three in such a short time, and, indeed, I have never seen anyone work as hard as they did; without their hard work dedication there would have been no chip. I could not have asked for better colleagues and friends.

In addition to helping with the final stages of the tapeout process, Ilia Lebedev built a compiler for our custom stack-based architecture and designed an interface board for mounting and testing the EM² chip. He was always ready to help, be it with the project or with anything else.

I am grateful to the Peh group, and especially Owen Hsin, for sharing their expertise and advice. When we took upon us the task of building an ASIC, we had no idea how little we actually knew about the subject, and would certainly have failed without their helping hand.

The early work on thread migration for shared memory was a collaboration with Omer Khan, and I fondly remember the intellectually stimulating days we spent poring over diagrams and data.

Many thanks to Charles Herder, who out of sheer kindness helped us with testing the EM² chip. He freely offered advice and patiently explained every step, betraying not only a kind heart but also a rare talent for teaching.

My academic pedigree I owe to my supervisor, Srinivas Devadas. His example has helped me mature as a researcher, and, although I yet have far to travel, my tastes and style bear the imprint of his guidance. His ability to engage with students as research equals created the best research environment I could have imagined, and in this too I hope to follow his example.

I am grateful to the members of the Computation Structures Group who have come and gone over the years, for supporting a great environment for professional and social interaction. They have contributed both as colleagues and as friends, as the need arose.

Finally, I can never repay the debt of gratitude I owe to my family, who have never failed in unquestioningly offering their support.

CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

In this dissertation, we design a mechanism for fast, fine-grained thread migration targeted for 100+-core general-purpose multicores, and develop a migration-based shared-memory scheme. We then demonstrate the feasibility of these mechanisms by realizing them in a 110-core general-purpose multicore processor implemented in 45nm ASIC technology.

To provide background for this discussion, this chapter explores the issues surrounding of the advent of multicores in the last decade, and discusses existing approaches to shared memory and thread migration.

1.1. THE DAWN OF THE MULTICORE ERA

The past decade has brought a revolution in computer architecture: computer processors, instead of running on ever faster clocks, have evolved into designs that place multiple processing cores on a single chip. This rather radical change has occurred because while transistors continue to get smaller and smaller, their power requirements no longer reduce accordingly, and practical consideration limit processors to roughly 100 watts of power (despite the development increasingly intricate cooling designs, dissipating very much more from a thumbprint-sized area has not proved commercially viable).

Why is this? Until recently, CMOS transistor technology has followed what has become known as Dennard scaling (Dennard et al., 1974). As the linear dimensions of the transistor are halved, the number of transistors in a given area (N) quadruples; Dennard et al. observed that, at the same time, the dynamic capacitive load (C) and supply voltage (V) halve and the maximum switching frequency (f) doubles. Power density per unit area depends on all three of those,

$$P_{density} = NCV^2f,$$

so halving transistor dimensions keeps this quantity constant:

$$\text{new } P_{\text{density}} = (4N)\left(\frac{C}{2}\right)\left(\frac{V}{2}\right)^2(2f) = NCV^2f.$$

More than anything, this property has been responsible for the exponential growth in processor performance from the late 1970s until about the mid-2000s.

About a decade ago, however, the corresponding reductions in the supply voltage V have slowed dramatically. This is mainly because a supply voltage that is too close to the transistor's threshold voltage allows some current to flow through even if the transistor is switched *off*—i.e., the circuit "leaks" current (and dissipates power) even if no computation is being performed. In addition, While accepting a lower reduction in voltage keeps leakage in check, however, it means that the V^2 term in the power density equation is reduced by a substantially lower factor, and one must also forgo increases in operating frequency to keep power density constant.

Even though we can no longer run processors at faster and faster frequencies, transistor dimensions still continue to shrink. Designers have taken advantage of this by placing multiple processor cores on a single die: in effect, instead of making each processor core significantly faster, the available transistor area has been used to place multiple cores on a single chip. At the time of this writing, 4 to 8 cores are common in commodity general-purpose processors, and even low-power applications such as mobile phones feature processors with 2 to 4 cores.

1.2. SHARED MEMORY AND THREADS

Unfortunately, while a typical sequential program suitable for a single-core processor will (roughly) run twice as quickly if the operating frequency of the computing system doubles (with the processor still executing a sequence of instructions), it will not automatically take advantage of a processor with several cores, which execute independent streams of instructions in parallel.

To manage this concurrency, two kinds of models have developed: (a) a *message passing* model where the processor cores effectively run separate programs that share no data and must actively communicate, and (b) a *shared memory* model where the cores run different parts of the *same* program, reading and writing the same memory space.[1] While message passing allows the programmer to achieve optimal performance by manually controlling all

[1]Some architectures, such as GPUs, achieve massive parallelism by restricting the efficiently implementable program space to stream-like kernels, and are beyond the scope of this dissertation.

intercore communication, it comes at the very high cost in terms of programmer effort and time, and for most applications shared memory is by far the preferred approach (Sodan, 2005).

As in the message-passing paradigm, in a shared-memory scheme different processor cores run potentially different instruction sequences in parallel. The difference lies in how threads communicate. Message-passing programs have separate copies of all their variables in each core, so that if one of the cores changes the value of a variable (say x), no other cores will directly observe the change; to coordinate the program's threads, the programmer must *explicitly* add code that transmits messages among the program's threads. Shared-memory threads, on the other hand, access the same copy of each variable: if one thread changes x from 23 to 37, then all other threads immediately and automatically observe the change. With shared memory, inter-thread communication occurs *implicitly*, and the underlying hardware must enable a core to automatically observe any changes to the memory space effected by the other cores in the system.

1.3. CACHES AND THE TROUBLE WITH SHARED MEMORY

Implementing shared memory in a modern multi-core processor is, however, far from straightforward, because modern processors implement *caches* to mitigate main memory latency.

The main memory in today's computers consists of DRAM; while this inexpensive and area-efficient solution allows for multi-gigabyte memory sizes, however, access times are slower—by roughly two orders of magnitude—than the time a processor core takes to execute a single instruction. Caches are very fast but small memories that keep copies of the most recently accessed data very close to the processor, with access times of a few clock cycles. To achieve such short access times, caches must be close to the processor core they serve,[2] and so caches tend to be private to each core.

Private caches, however, cause a problem for shared memory, because if multiple caches are allowed to hold copies of a variable x, they must somehow all agree as to the value of x. This is usually accomplished with complex *cache coherence protocols* that coordinate the caches in the different cores to ensure this kind of agreement (see Chapter 2 for related research). Despite recent advances, however, cache coherence protocols remain difficult to implement and verify, and do not easily scale to large numbers of cores (Arvind et al., 2008; DeOrio et al., 2008; Zhang et al., 2010); as a result, a number of recent high-core-count processors have eschewed a hardware coherence mechanism (e.g., Mattson et al., 2010).

Existing techniques—hardware cache coherence protocols, distributed shared cache, software cache coherence, and so on—used to implement shared memory all have one thing in

[2]Wire delays do not scale as transistors do: while the capacitance C decreases as wires get smaller, the resistance R goes up, and the delay RC remains roughly constant per unit distance.

common: they all bring data to the core that wishes to compute on it. A core contribution of this dissertation is turning this idea upside down: we instead migrate the computation from one core to another, bringing it to the data it wants to access.

1.4. GETTING THREADS IN AND OUT OF THE CPU

How does one move a running thread from one processor core to another? It turns out that operating systems—even running on single-core processors—have long done something very similar. One reason for this is time-multiplexing, a way to achieve the illusion of running multiple threads (or processes) on a single CPU. In this technique, after running for a single time slice (a period dependent on the task's priority, but usually ranging from a few to a few hundred milliseconds), the operating system (OS) may evict the currently running thread from the processor core and load in another thread, an event called a context switch. Another reason is to put the processor in a power-saving sleep state: to accomplish this, the currently running thread is stored in a special memory region and the processor itself is powered down until it is woken up.

In either case, the complete state of the currently running thread must be moved out of the CPU and stored in memory. At a minimum, this state consists of the program counter (PC)—which holds the memory address of the next instruction to be executed—as well as all of the general-purpose and special-purpose registers, flags, control registers, and input/output/interrupt information, but may include other cached state to avoid recomputation when the thread is woken up again.

Because the timescales involved are fairly long compared to processor clock cycles—on the order of milliseconds—there has been little need for thread loading and unloading tasks to be optimized for speed, and as a result they have largely been implemented in software. For the purposes of this dissertation, we need to move *running* threads among the cores in timeframes not exceeding tens of cycles, and existing solutions are far too slow; we therefore develop a technique to move threads entirely in hardware.

1.5. THE CONTRIBUTIONS OF THIS DISSERTATION

In this dissertation, we develop an autonomous mechanism for fast, fine-grained thread migration in multicore processors, and outline the protocol-level innovations and microarchitectural details required to implement it entirely in hardware. Capitalizing on the efficiency of this implementation, we then design a simple, scalable shared memory mechanism, leveraging migration to match state-of-the-art performance levels. Finally, we describe the real-

ization of these concepts in the Execution Migration Machine (EM^2), a 110-core chip multiprocessor implemented in 45nm ASIC technology.

Specifically, the contributions of this work are:

1. the design and implementation of the first autonomous, deadlock-free migration scheme implementable in hardware;
2. the concept of *partial context migration*, which significantly reduces thread migration costs by transferring only a useful part of the thread context;
3. an implementation of the sequentially-consistent shared memory abstraction using purely migration;
4. an optimized implementation using both migration and remote cache access that is competitive with the state of the art;
5. a detailed description of the first implementation of hardware-level thread migration and migration-based shared memory in a 110-core CMP;
6. an exploration of what computation patterns benefit from thread migration; and
7. a detailed performance comparison vs. an "ideal" cache coherence baseline implemented in RTL.

This thesis constitutes the final part of a trinity of dissertations centered around execution migration and the EM^2 chip, following those of Cho (2013) and Shim (2014). Designing and building such a large chip is by necessity a group activity, and, as such, there is a fair amount of overlap among the three.

Specifically, the contribution of the deadlock-free migration protocol described in Chapter 3 is shared with Cho (2013), while the design and implementation of migration prediction described in Chapter 5 are due to Shim (2014). Lebedev (2013) describes compilation and optimization techniques for the stack machine architecture detailed in Appendix A.

In the following sections, after reviewing related work, we develop a design for fast, fine-grained thread migration that can be realized in hardware, and describe how to implement shared memory using thread migration.

CHAPTER 2

RELATED WORK

This chapter delineates the significance of the main contributions of the present dissertation and places them in the context of previous research.

2.1. THREAD MIGRATION

In symmetric multiprocessor (SMP) systems and chip multi-processors (CMPs), process and thread migration has long been employed to provide load and thermal balancing among the processor cores and across separate processors. Typically, migration is a direct consequence of thread scheduling and is performed by the operating system (OS) at timeslice granularity; although this approach works well for achieving long-term goals like load balancing, the relatively long periods, expensive OS overheads, and high communication costs have generally rendered other applications impractical.

The advent of non-uniform memory access (NUMA) architectures in the 1990s (e.g., the MIT Alewife machine, Agarwal et al., 1995)—as well as the subsequent appearance of on-chip memory controllers in commodity processors during the following decade—sparked a renewed interest in thread migration. While the unified shared memory abstraction (provided via a shared bus or a distributed cache coherence protocol) made references to any address possible for any processor, accesses to regions of memory physically adjacent to a given processor were significantly faster; thus, performance improvements could potentially be achieved if threads could be moved to follow the data they operated on. Approaches like Computation Migration (Hsieh et al., 1993), Ariadne (Mascarenhas and Rego, 1996), Millipede (Itzkovitz et al., 1998), Active Threads (Weissman et al., 1998), thread migration in D-CVM (Thitikamol and Keleher, 1999), and the O^2 scheduler (Boyd-Wickizer et al., 2009) are in the end limited by the costly overheads of a software implementation. Mobile Multithreading (Dorojevets and Strukov, 2002) demonstrated significant speedup potential if migration

overheads could be kept extremely low, and highlighted the need for the fastest thread migration mechanism possible.

More recently, the need for an efficient migration mechanism has resurfaced. Rapid thread migration among cores in different voltage/frequency domains has been used to run less demanding computation phases on slower cores, with the goal of improving overall power-performance ratios (Rangan et al., 2009), to accelerate critical sections by running them on the higher-performance ("fat") cores of a heterogeneous CMP (Suleman et al., 2009), to take advantage of the overall on-chip cache capacity and improving performance of sequential programs (Michaud, 2004), to improve code locality (Chakraborty et al., 2006), and to accelerate locks (Joao et al., 2012). In the area of reliability, fine-grained inter-core thread migration has been proposed to salvage cores that cannot execute some instructions because of manufacturing faults (Powell et al., 2009).

What has been lacking is a suitably efficient implementation of fast, fine-grained thread migration. The most important consideration is that such a protocol must be deadlock-free: if a thread context can be blocked by other contexts during migration, this additional resource dependency can cause the on-chip interconnect to lock up. Secondly, the mechanism must support instruction-granularity migrations: if the next instruction cannot be executed on the current core (e.g., because of a hardware fault), the thread must move immediately. Finally, migrations must have minimal overhead, both to outperform emulation mechanisms for fault recovery and to provide any hope of accelerating memory accesses. Existing studies, however, fail to satisfy one or more of these requirements: they either implicitly rely on expensive, centralized migration protocols that provide deadlock freedom but incur overheads that preclude frequent migrations (Hu et al., 2010; Misler and Jerger, 2010), limit migrations to a core's local neighborhood (Shaw and Dally, 2002), or simply give up on deadlock avoidance and rely on expensive recovery mechanisms (e.g., Melvin et al., 2003).

In this dissertation, we develop a hardware-level thread migration mechanism that meets all of those requirements. First, we describe a provably deadlock-free migration protocol that is amenable to efficient implementation; unlike previous work, migrations are autonomous and do not require handshaking. We then outline the microarchitectural details of a hardware-level implementation of thread migration and show how to migrate only the useful subset of a thread to obtain lower migration latency and reduced on-chip traffic. Finally, we describe how these concepts are implemented in a 110-core chip multiprocessor ASIC.

2.2. SHARED MEMORY

Experience with early multiprocessor systems made it obvious that keeping the illusion of a single unified memory significantly eases programming, and a shared memory abstraction—

with sequential consistency (Lamport, 1979) as the ultimate benchmark—has become sine qua non for general-purpose multiprocessors.[1] To support this abstraction, designers have turned to directory-based cache coherence, in which complex protocols ensure agreement among per-core private caches. For large-scale CMPs, however, the scalability of directories is arguably a critical challenge since the area required for directories and coherence traffic overhead keeps increasing along with the core count. Although some recent work has proposed more scalable directories in terms of area and performance (e.g., Feldman et al., 2011; Sanchez and Kozyrakis, 2012), the design and verification complexity of directories and complex coherence protocols still remain significant and do not easily scale to a large number of cores (Arvind et al., 2008; DeOrio et al., 2008; Zhang et al., 2010).

In addition to providing a consistent memory model, designers of modern multicores must address the physical distribution of cache memory across the chip area, and the consequent variation in cache access times. This body of work has its roots in the non-uniform memory access (NUMA) paradigm as extended to single-die caches (NUCA: Kim et al., 2002; Chishti et al., 2003) and chip multiprocessors (Beckmann and Wood., 2004; Huh et al., 2005). To speed up access times, studies investigated the data movement of data towards the locus of computation in NUMA machines (e.g., Verghese et al., 1996). More recently, similar data placement ideas have been explored in the different tradeoff regime of single-chip multiprocessors. Cho and Jin (2006) have proposed page-granularity approaches that leverage the virtual addressing system and the TLBs to improve locality, CoG (Awasthi et al., 2009) explored moving pages to the "center of gravity" to improve data placement, and the O^2 scheduler (Boyd-Wickizer et al., 2009) made data placement a factor in OS thread scheduling. Other schemes proposed hardware support for page migration support (Chaudhuri, 2009; Sudan et al., 2010); at finer granularity, Zhang and Asanović (2005) addressed locality by keeping L1 cache victims in the local L2 cache. In a similar vein, Reactive NUCA (Hardavellas et al., 2009) explored automatic detection of data access patterns and providing limited replication to accelerate performance on read-only data.

Various alternative designs have the complexity of cache coherence protocols while keeping, relaxing, or even dispensing with the shared memory abstraction. One approach is to disallow automatic data replication, divide the address space among per-core caches, and provide a hardware mechanism to access data cached in remote locations (e.g., Fensch and Cintra, 2008); while the shared memory abstraction is maintained, performance can suffer because remotely cached data cannot be cached locally and incur the remote access penalty every time they are accessed. Another is to transfer the burden of cache coherence from hardware to the operating system and software (Kontothanassis et al., 1997; Zeffer et al., 2006),

[1]For performance reasons, most multiprocessors do not implement sequential consistency directly, but achieve it via a combination of a more efficient relaxed memory model and memory barrier instructions.

or to require the programmer to obey a more disciplined shared-memory model to simplify the hardware coherence mechanism (e.g., DeNovo, Choi et al., 2011). Finally, some designs, such as Intel's 48-core Single-Chip Cloud Computer (Mattson et al., 2010), forgo a hardware coherence mechanism entirely and require the programmer to explicitly manage data consistency.

In this dissertation, we employ fine-grained thread migration to accelerate a remote-cache access scheme similar to that of Fensch and Cintra (2008), providing the unified shared memory abstraction entirely in hardware. As in previous work, our scheme eschews automatic data replication, and, consequently, the cost and complexity of providing automatic cache coherence; this design choice allowed us to trivially scale the implementation from a 4-core testbed to the full 110-core ASIC. Unlike similar approaches, however, our approach can match or exceed the performance of directory-based cache coherence on a range of applications while reducing (sometimes dramatically) on-chip network traffic rates (and consequently dynamic power). In addition, because memory accesses form a significant part of the instruction stream, implementing shared memory places strict demands on the thread migration framework and allows us to stress-test and evaluate our migration mechanism.

CHAPTER 3

A DEADLOCK-FREE THREAD MIGRATION PROTOCOL

We begin our study of thread migration by developing a migration protocol suitable for fast, fine-grained thread movement and amenable to straightforward implementation in hardware. After demonstrating how deadlock arises in a naïve approach to thread migration, we show how to design a low-latency protocol where deadlock never arises.

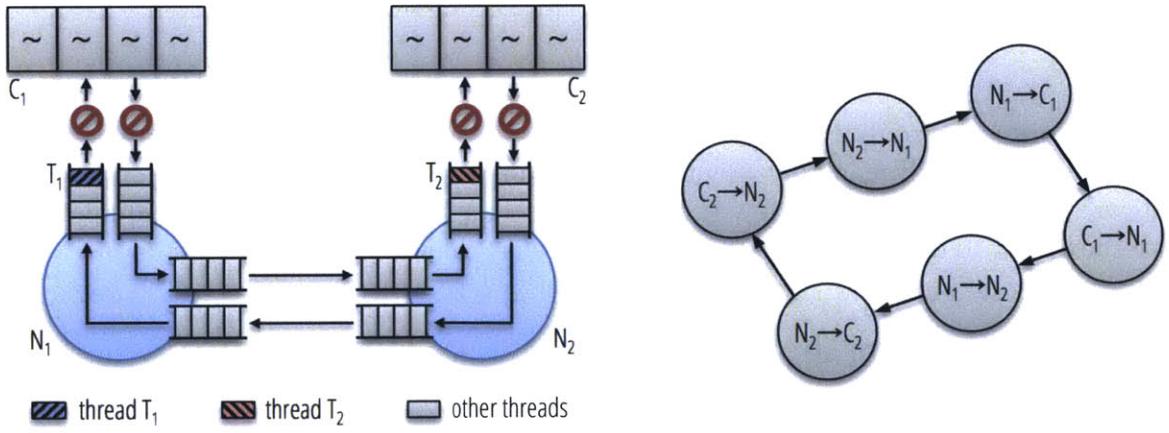
3.1. THE EMERGENCE OF DEADLOCK

Most on-chip network routing protocol studies focus on the network itself and assume that a network packet *dies* soon after it reaches its destination core: for example, the result of a memory load request might simply be written to its destination register. This assumption greatly simplifies deadlock analysis because the dead packet no longer holds any resources that might be needed by other packets, and only *live* packets are involved in deadlock scenarios.

When a routing protocol is used for thread migration, however, this assumption no longer holds. The packet carries an execution context, which moves to an execution unit in the core and *occupies* it until it migrates again to a different core.[1] The need to evict such execution contexts—which may not always be possible—introduces a scenario where contexts arriving over the network are blocked, and creates additional dependencies that conventional on-chip network deadlock analysis does not consider.

For example, suppose a migrating thread T_1 in Figure 3-1a is heading to core C_1. Although T_1 arrives at routing node N_1 directly attached to C_1, all the execution units of C_1 are occupied

[1]We could, of course, avoid this with centralized migration scheduling or a scheme where a thread requests space from the remote core and waits for the reservation to be confirmed before migrating; both approaches, however, require round-trip requests and conflict with our goal of supporting migration with the lowest possible latency.



(a) the network buffers situation

(b) the channel dependency graph

Figure 3-1: Deadlock in a naïve thread migration protocol where threads simply swap. Thread T_1 wishes to enter core C_1, but all contexts are occupied by other threads (~). One of those must be evicted, but the network buffers are filled up with other migrating threads and will not clear until T_2 enters core C_2. Thread T_2, however, has the same problem, and ultimately depends on T_1 entering its destination. Because of the cyclic dependency, no thread can make progress, and deadlock ensues.

by other threads (~), and one of them must migrate to another core for T_1 to make progress. But at the same time, thread T_2 has the same problem at core C_2, so the contexts queued behind T_2 are backed up all the way to C_1 and prevent a C_1 thread from leaving. So T_{a1} cannot make progress, and the contexts queued behind it have backed up all the way to C_2, preventing any of C_2's threads from leaving, and completing the deadlock cycle. Figure 3-1b illustrates this deadlock using a channel dependency graph (CDG; see Dally and Towles, 2003) where nodes correspond to channels of the on-chip network and edges to dependencies associated with making progress on the network.

We call this type of deadlock a *protocol-level deadlock*, because it is caused by the migration protocol itself rather than the network routing scheme. Previous studies involving rapid thread migration typically either do not discuss protocol-level deadlock, implicitly relying on a centralized deadlock-free migration scheduler (Hu et al., 2010; Misler and Jerger, 2010; Shaw and Dally, 2002), using deadlock detection and recovery (Melvin et al., 2003), employing a cache coherence protocol to migrate contexts via the cache and memory hierarchy, effectively providing a very large buffer to store contexts (Rangan et al., 2009), or employing slow handshake-based context swaps (Powell et al., 2009). All of these approaches have substantial overheads, motivating the development of an efficient network-level deadlock-free migration protocol.

3.2. DEADLOCK IN PRACTICE

We have shown that using an on-chip network for thread migration introduces additional channel dependencies that may in some circumstances result in deadlock. But do those conditions actually occur in practice? In this section, we use a synthetic migration benchmark running on a network-on-chip simulator to answer that question.

To this end, we consider the naturally arising SWAP protocol, implicitly assumed by several works. In this scheme, whenever a migrating thread T_1 arrives at a core, it evicts the thread T_2 currently executing there and sends it back to the core where T_1 originated. Although intuitively one might expect that this scheme should not deadlock because T_2 can be evicted into the slot that T_1 came from, this slot is not actually reserved for T_2 and another thread might migrate there more quickly, preempting T_2; it is therefore not guaranteed that T_2 will exit the network and deadlock may in fact arise.[2]

In order to examine how often the migration system might lock up, we used a synthetic migration benchmark where each thread keeps migrating between the initial core where it was spawned and a *hotspot* core. (Since migration typically occurs to access some resource

[2]Although adding a handshake protocol with extra buffering can make SWAP deadlock-free (Powell et al., 2009), the resulting scheme is too slow our goal of fast, frequent migrations.

Core and migration	
core architecture	single-issue, two-way SMT
thread context size (relative to flit size)	4 flits
number of threads	64
number of hotspots	1, 2, 3 and 4
migration interval	100 cycles
On-chip network interconnect	
network topology	8-by-8 mesh
routing algorithms	dimension-order wormhole routing
number of virtual channels	2 and 4
network buffer size (relative to context size)	4 per link or 20 per node
context queue size (relative to context size)	0, 4, and 8 per core

Table 3.1: The simulation setup for synthetic migration benchmarks.

at a core, be it a functional unit, a lock, or a set of memory locations, such hotspots naturally arise in multithreaded applications.) We used varying numbers (one to four) of randomly assigned hotspots, and 64 randomly located threads that made 1000 migrations to destinations randomly chosen among their originating core and the various hotspots every 100 cycles. We used the cycle-level network-on-chip simulator HORNET (Lis et al., 2011), suitably modified with a migration system, to model a 64-core system connected by a 2D mesh interconnect. Each on-chip network router had enough network buffers to hold 4 thread contexts on each link with either 2 or 4 virtual channels; we also examined the case where each core has a context queue to hold arriving thread contexts when there are no available execution units. We assumed Intel Atom-like x86 cores with execution contexts of 2Kbits (Rangan et al., 2009) and enough network bandwidth to fit each context in four or eight flits. Table 3.1 summarizes the simulation setup.

Figure 3-2 shows the percentage of runs (out of 100) that end with deadlock under the SWAP scheme. Without an additional context queue, nearly all experiments end in deadlock; worse yet, even though context buffering can *reduce* deadlock, deadlock still occurs at a significant rate for the tested configurations.

The synthetic benchmark results also illustrate that susceptibility to deadlock depends on migration patterns: when there is only one hotspot, the migration patterns across threads are usually not cyclic because each thread just moves back and forth between its own private core and only one shared core; when there are two or more hotspots and threads have more

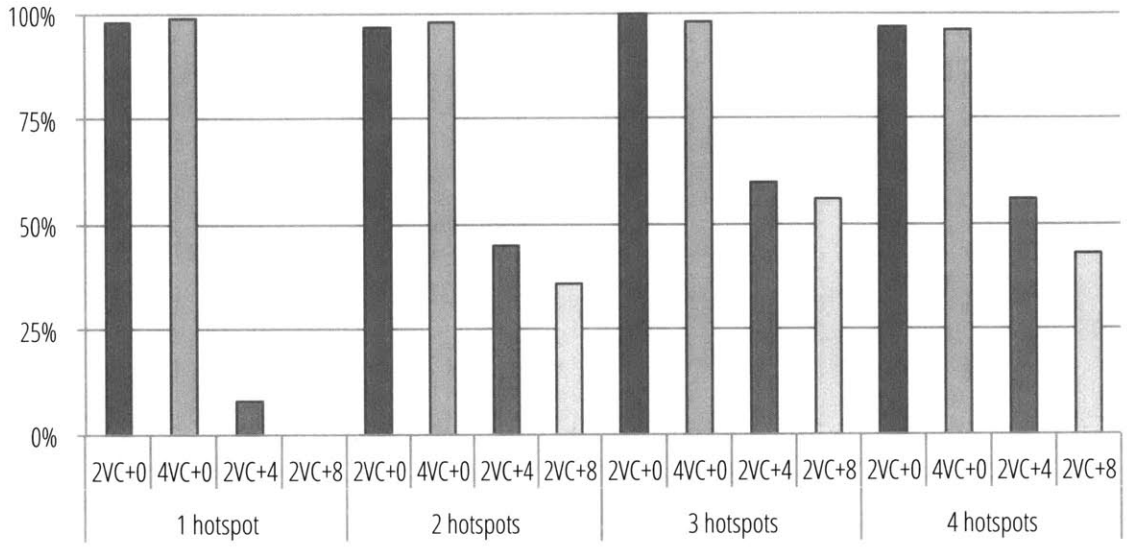


Figure 3-2: The percentage of random synthetic migration patterns that deadlock under the SWAP scheme under various configurations: 2VC+4, for example, corresponds to 2 virtual channels and a context queue of 4 contexts.

destinations, on the other hand, their paths intersect in more complex ways, making the system prone to deadlock. Most importantly, while small context buffers prevent deadlock with some migration patterns, they do not ensure deadlock avoidance.

3.3. A DEADLOCK-FREE PROTOCOL

To support fine-grained thread migration, therefore, we need to design a migration protocol that is deadlock-free from the ground up. Our scheme, called the Exclusive Native Context (ENC) protocol, takes a network-based approach to allow autonomous thread migration while providing deadlock freedom.

With ENC, threads may begin a migration autonomously, without coordinating with any other core or thread. When such a thread arrives at its destination core and the destination core has no available contexts, the new thread may evict one of the thread contexts already executing in the destination core; ENC provides the evicted thread context a *safe path* to another core on which it will never be blocked by other threads that are in transit concurrently.

To provide the all-important safe path for evicted threads, ENC uses a set of policies in core scheduling, routing, and virtual channel allocation.

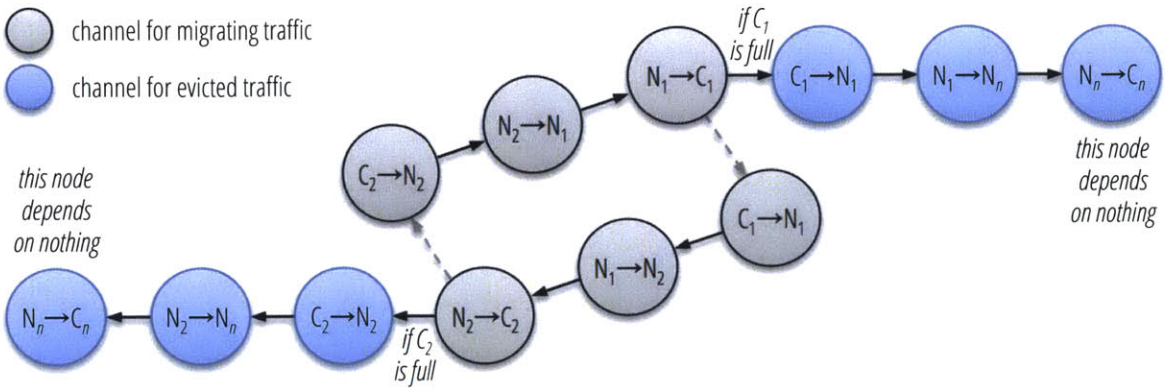


Figure 3-3: The acyclic channel dependency graph of ENC.

Each thread is considered a *native thread* in one particular core, which reserves storage space (a register file and other associated context state) for the thread. Other threads cannot use this reserved resource even if it is not being used by the native context; therefore, a thread will always find an available resource every time it arrives at the core where the thread is considered native. We will refer to this core as the thread's *native core.*

If an arriving thread is not a native thread at the destination core, it may be temporarily blocked by other non-native threads currently on the same core.[3] Eventually, the new thread evicts one of the executing non-native threads and takes the released resource. (Recall that a thread never evicts a *native* context from the destination core because the resource is usable only by the native thread.)

Dedicating resources to native contexts requires some rudimentary multithreading support in the cores. If a thread may migrate to an arbitrary core which may have a different thread as its native context, the core needs to have an additional register file (i.e., a *guest context*) to accept a non-native thread because the first register file is only available to the *native* context. Additionally, if a core has multiple native contexts, there must be enough resources to hold all of its native contexts simultaneously so no native thread is blocked by other native threads. Because of the multiple contexts, an efficient fine-grained migration-based architecture will require some level of multithreading in order to prevent performance degradation when multiple threads compete for the resources of the same core.

We first describe a basic, straightforward version of ENC, which we call ENC0, and then

[3]To prevent livelock, a thread is not evicted unless it has executed at least one instruction since it arrived at the core it currently occupies; that is, an existing thread may be evicted by a new thread only if it has made some progress in its current visit in that core.

describe a better-performing optimized version.

The basic ENC algorithm (ENC0)

Whenever a thread wishes to move from a non-native core to a destination core, ENC0 first sends the thread to its native core which has a dedicated resource for the thread. If the destination core is not the native core, the thread will then move from its native core to the destination core. Therefore, from a network standpoint, a thread migration either *ends* at its native core or *begins* from its native core. Since a thread arriving at its native core is guaranteed to be unloaded from the network, any migration process is in a fully unloaded state (and therefore momentarily occupies no network resources) somewhere along its path.

To keep the migrations deadlock-free, however, we must also ensure that thread migrations destined for a native core actually get there without being blocked by any other migrations; otherwise the native-core movements might never arrive to be unloaded from the network. The most straightforward way of ensuring this is to use two sets of virtual channels, one for to-native-core traffic and the other for from-native-core traffic. Thus, ENC0 doubles the number of virtual channels required by the underlying network routing protocol: for example, if for dimension-order routing, which requires only one virtual channel to prevent network-level deadlock, ENC0 requires two virtual channels.

The full ENC algorithm

Although ENC0 is simple and straightforward, it incurs the overhead of introducing an intermediate destination for each thread migration: if thread T wishes to move from core A to B, it must first go to N, the native core for T. In some cases, this overhead might be significant: if A and B are close to each other, and N is far away, the move may take much longer than if it had been a direct move.

To reduce this overhead, we observe that indirections through the native core are not *always* required to keep the protocol deadlock-free. Specifically, migrations that evict other threads will in fact eventually complete *provided* that the evicted thread is guaranteed to leave its context. This suggests a distinction between *migrating* traffic and *evicted* traffic: the former consists of threads that wish to migrate on their own because, for example, they wish to access resources in a remote core, while the latter corresponds to the threads that are evicted from a core by another arriving thread.

The optimized version of the algorithm, which we simply call ENC, sends a thread to its native core only when it has been *evicted.* This prevents a chain of evictions: even if the evicted thread wishes to go to a different core to make progress (e.g., return to the core it was

evicted from), it must first visit its native core, get unloaded from the network, and then move again to its desired destination. Unlike ENC0, however, whenever a thread migrates on its own accord, it may go directly to its destination *without* visiting the native core. (Like ENC0, ENC must guarantee that evicted traffic is never blocked by migrating traffic; as before, this requires two sets of virtual channels.)

Based on these policies, the ENC migration algorithm can be described as follows. Note that network packets always travel within the same set of virtual channels.

1. If a native thread has arrived and is waiting on the network, move it to a reserved native context and proceed to Step 3.
2. (a) If a non-native thread is waiting on the network and there is an available guest context, move the thread to this context and proceed to Step 3.

   (b) If a non-native thread is waiting on the network and all the non-native contexts are full, choose one thread from among the threads that have finished executing at least one instruction on the core[4] *and* the threads that want to migrate to other cores. Send the chosen thread to its *native* core on the virtual channel reserved for evicted traffic. Then, advance to the next cycle (no need for Step 3).
3. Among the threads that want to migrate to other cores, choose one and send it to the desired destination on the virtual channel reserved for migrating traffic. Then, advance to the next cycle.

This algorithm breaks the cycle of dependency of migrating traffic and evicted traffic: Figure 3-3 illustrates how the cyclic dependency from Figure 3-1b is broken (C_n denotes the native core of the evicted thread, and N_n its attached router node).

There is a subtlety when a migrating thread consists of multiple flits and the core cannot send out an entire context all at once. For example, the core may find no incoming contexts at cycle 0 and start sending out an executing context T_1 to its desired destination, but before T_1 completely leaves the core, a new migrating context, T_2, arrives at the core and is blocked by the remaining flits of T_1. Because T_1 and T_2 are on the same set of virtual channels for migration traffic, a cycle of dependencies may cause a deadlock. To avoid this case, the core must inject migration traffic only if the whole context can be moved out from the execution unit, so that arriving contexts will not be blocked by incomplete migrations; this can easily be implemented by monitoring the available size of the first buffer on the network for migration traffic or by adding an additional outgoing buffer sufficient to hold one thread context.

[4]This prevents livelock.

Both ENC0 and ENC are provably deadlock-free under deadlock-free routing because they eliminate all additional dependencies due to limited context space in cores. We confirmed this using the same synthetic benchmarks used in Section 3.2 (data not shown). We also simulated an incomplete version of ENC that does *not* consider the partial-context subtlety described above and sends out a migrating context as soon as it is possible to push out its first flit: while ENC0 and ENC had no deadlocks, the incomplete version caused deadlock. This illustrates that fine-grained migration is very susceptible to deadlock and migration protocols need to be carefully designed.

CHAPTER 4

MIGRATION-BASED SHARED MEMORY

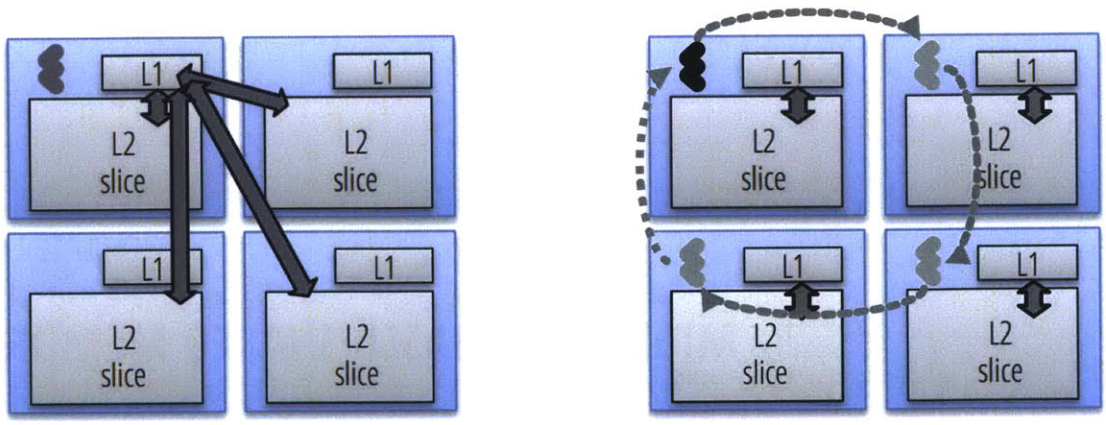
In this chapter, we introduce a unified shared memory model based on fine-grained thread migration. Implementing shared memory support on top of migration allows us to evaluate the migration framework using standard benchmarks (where memory instructions may cause migrations). Equally important is the fact that memory access is a very fine-grained phenomenon that directly affects application performance: this presents very strict performance demands on the migration framework that will drive the development of design.

4.1. A BASIC SHARED MEMORY SCHEME

To provide all cores with access to the entire address space, our migration-based memory model treats the per-core caches as a single distributed shared cache: the per-core caches are responsible for caching non-overlapping ranges of the address space, which together cover all of addressable memory. Because data are not automatically replicated—each address may be cached only at its unique *home core*—there is no need for a protocol to ensure coherence among multiple copies of data, and memory consistency is easy to reason about.

Dividing the address space among the per-core caches is a feature of existing shared memory implementations (e.g., Fensch and Cintra, 2008), and is common with on-chip last-level caches in production multicores. In those designs, whenever a core wishes to access an address assigned to a remote cache, the hardware satisfies the request by sending a remote-access message to the target cache over the on-chip interconnect; any results (load results, store acknowledgements) are sent back over the interconnect to the original core via another message. We will refer to these shared memory models as remote access (RA) architectures.

Our migration-based shared memory scheme, which we call Execution Migration (EM), eschews remote cache accesses in favor of migrating the thread of execution to the core where the memory resides and continuing execution there, in effect turning the remote cache access



(a) bringing data to computation

(b) bringing computation to data

Figure 4-1: Traditional shared-memory paradigms (such as cache coherence or remote access) bring data to the locus of computation, while the migration-based shared memory architecture we describe here brings the computation to the data it operates on.

into a local cache access by running the thread on the remote core (see Figure 4-1). In essence, instead of moving *data* to feed a given computation, we move the *computation* to consume its data. In what follows, we first consider a migration-only variant called EM0, and, after evaluating its pros and cons, develop a more optimized version.

Under EM0, if a thread is already executing at the destination core, it must be evicted and migrated to a core where it can continue running. In accordance with the ENC protocol (see Chapter 3), we assume that each core contains at least one *native context* (for threads originating on that core) and at least one *guest context* (for threads originating elsewhere). In addition, to prevent deadlock, we follow ENC in requiring that an evicted thread must travel to its native core over a virtual network that is only used for evictions.

Figure 4-2 illustrates the life of a memory access in the EM0 shared memory implementation. Briefly, when a core C running thread T executes a memory access for address A, it must

1. compute the *home* core H for A (e.g., by masking the appropriate bits);
2. if H = C (a *core hit*),
   (a) forward the request for A to the cache hierarchy (possibly resulting in a DRAM access);
3. if H ≠ C (a *core miss*),

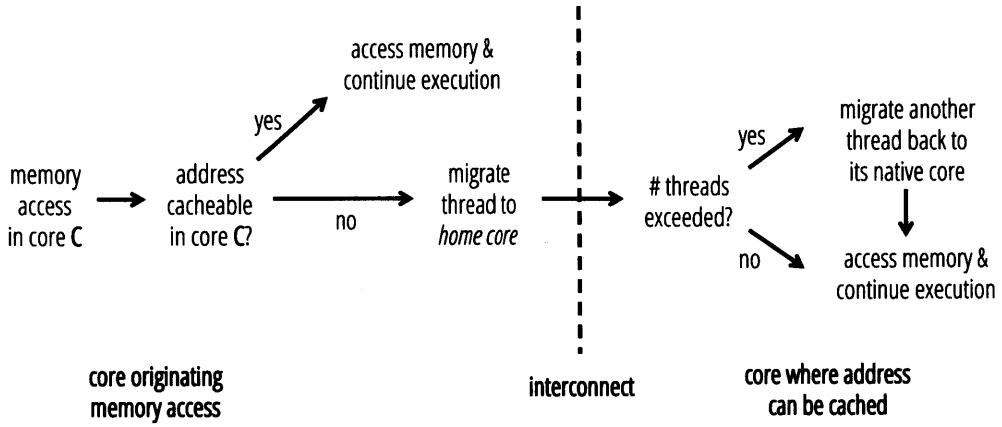


Figure 4-2: The life of a memory access under the EM0 migration-based shared memory scheme.

(a) interrupt the execution of the thread on C (as for a precise exception),

(b) migrate the microarchitectural state to H via the on-chip interconnect:

   i. if H is the native core for T, place it in the native context slot;

   ii. otherwise:

      A. if the guest slot on H contains another thread T′, evict T′ and migrate it to its native core N′

      B. move T into the guest slot for H;

(c) resume execution of T on H, requesting A from its cache hierarchy (and potentially accessing the next-level cache or backing DRAM).

Unlike distributed shared memory architectures based on remote cache access, where repeated accesses to even the same remote location must incur the penalty of a remote-access round trip to ensure memory consistency, migration-based shared memory can potentially take advantage of spatiotemporal locality: the thread, having moved to where the memory is located, can continue to access a sequence of nearby locations using local cache requests.

Compared to traditional directory-based cache coherence, the migration-based approach has advantages in simplicity and scalability: since the state of each cache line does not depend on all potential sharers (i.e., every core in the system), the protocol transitions and state remain the same regardless of the number of cores in the system. In addition, as any shared memory approach that eschews replication, it also has the advantage of larger effective on-chip cache capacity: because only one copy of any given address may be present on chip,

there is more room to store other data. Finally, it avoids the round-trips and indirections (via the directory as well as other sharers) inherent in directory-based coherence protocols; while this is often outweighed by the disadvantages that can result from not replicating data, it can be a significant effect if the memory access patterns include large amounts of read-write sharing.

On the other hand, migration-based shared memory potentially suffers from a number of disadvantages. When spatiotemporal data access locality is poor, the overheads of migrating the entire thread—both in terms of network traffic and because of the added latency of loading and unloading threads from a core's execution context—outweigh the benefits and a message-based remote access architecture will perform better. When, on the other hand, cache pressure is low and read-only sharing dominates, the larger effective cache capacity and absence of indirections do not make up for the lack of data replication permitted by a cache coherence protocol. In the next section, we examine these tradeoffs.

4.2. ANALYTICAL MODEL

To develop intuition about the relative strengths and weaknesses of EM0, this section formulates analytical models based on *average memory latency* (AML). This model abstracts away multi-actor system interactions to focus on the average cost of a single memory access in terms of experimentally measurable quantities like off-chip memory and on-chip cache access delays, fractions of the various memory requests in programs, or cache miss rates for each request type. By clearly identifying the various sources of latency contributing to each shared memory paradigm, the AML model allows us to explore not only the avenues of optimization available in each protocol, but also to bound the maximum potential of each technique. In the remainder of this section, we describe the AML models for the MSI variant of directory cache coherence (DirCC), remote cache access (RA), and the migration-based protocol (EM0).

Interconnect latency

We adopted a uniform network model in which, on average, each type of core-to-core message travels the same number of hops (12 for an 8×8 mesh) and experiences similar congestion (50%). Packets are divided into equal-size (256-bit) flits, and comprise a header flit with several data flits; wormhole routing is assumed, so each packet experiences a one-cycle serialization

Parameter	Value
$cost_{L1\$\ access}$	2 cycles
$cost_{L1\$\ insert/inv/flush}$	3 cycles
$cost_{L2\$\ access}$	7 cycles
$cost_{L2\$\ insert}$	9 cycles
$cost_{dir\$\ lookup}$	2 cycles
$size_{ack}$	32 bits
$size_{address}$	32 bits
$size_{value}$	32 bits
$size_{cacheline}$	512 bits
$size_{thread}$	1088 bits
$cost_{DRAM}$	250 cycles (200 latency + 50 serialization)
flit size	256 bits
$cost_{avg\ net\ dist}$	36 cycles (12 hops × 2 cycles/hop + 50% congestion overhead)
$rate_{read}$, $rate_{write}$	70%, 30%
$rate_{rdI,wrI,rdS}$	40% + 40% + 5%
$rate_{wrS}$	5%
$rate_{rdM}$	10%
$rate_{wrM}$	negligible
$rate_{L1\$\ miss}$	6%
$rate_{L2\$\ miss}$	1%
$rate_{core\ miss}$	2%

Table 4.1: Default parameters for the analytical AML models, from architectural assumptions (top) and simulations over a standard set of benchmarks (bottom).

delay in addition to the latency of delivering the first flit:

$$\text{cost}_{\rightarrow,\,\text{data size}} = \text{cost}_{\text{avg net dist}} + \left\lceil \frac{\text{data size}}{\text{flit size}} \right\rceil,$$

where data size depends on the packet size, and grows for cache lines (64 bytes) and EM0 thread migrations (1088 bits = 32×32-bit registers, a 32-bit instruction pointer register, and a 32-bit status register):

$$\begin{aligned}
\text{cost}_{\rightarrow,\,\text{ack}} &= \text{cost}_{\rightarrow,\,\text{address}} = \text{cost}_{\rightarrow,\,\text{value}} \\
&= (12 \times 2 + 50\%) + \left\lceil \frac{32}{256} \right\rceil = 36 + 1 = 37 \text{ cycles}, \\
\text{cost}_{\rightarrow,\,\text{addr\&value}} &= (12 \times 2 + 50\%) + \left\lceil \frac{32 + 32}{256} \right\rceil = 36 + 1 = 37 \text{ cycles}, \\
\text{cost}_{\rightarrow,\,\text{cacheline}} &= (12 \times 2 + 50\%) + \left\lceil \frac{512}{256} \right\rceil = 36 + 2 = 38 \text{ cycles, and} \\
\text{cost}_{\rightarrow,\,\text{thread}} &= (12 \times 2 + 50\%) + \left\lceil \frac{1088}{256} \right\rceil = 36 + 5 = 41 \text{ cycles.}
\end{aligned}$$

Only for $\text{cost}_{\rightarrow,\,\text{thread}}$, we add an additional 3 cycles corresponding to restarting a five-stage pipeline with a new instruction at the destination core, resulting in $\text{cost}_{\rightarrow,\,\text{thread}} = 44$.

Last-level cache and DRAM latencies

To simplify analysis, we assumed a two-level data cache hierarchy: a per-core cache (L1) plus a shared last-level cache (L2). The L2 cache was modeled as a shared non-uniform access cache (NUCA) distributed in slices divided equally among all the cores, with each slice handling a non-overlapping subset of the address space. Directories (for directory cache coherence) were similarly distributed, with each per-core slice handling the same predefined address range as the L2 cache slice on the same core; therefore requests to L2 did not incur additional network costs above those of contacting the directory or library, but only the L2 access itself and the amortized cost of accessing off-chip memory:

$$\begin{aligned}
\text{cost}_{\text{L2\$ request}} &= \text{cost}_{\text{L2\$ access}} \\
&\quad + \text{rate}_{\text{L2\$ miss}} \times \left(\text{cost}_{\text{DRAM}} + \text{cost}_{\text{L2\$ insert}}\right) \\
&= 7 + 1\% \times (250 + 9) = 9.6 \text{ cycles.}
\end{aligned}$$

First-level cache miss effects

Under EM0 and RA, L1 misses at the home core of the address being accessed result directly in L2 requests, and have identical costs:

$$\begin{aligned}\text{cost}_{\text{L1\$ miss, RA}} &= \text{cost}_{\text{L1\$ miss, EM0}} \\ &= \text{cost}_{\text{L2\$ request}} + \text{cost}_{\text{L1\$ insert}} = 9.6 + 3 = 12.6 \text{ cycles.}\end{aligned}$$

In all L1 miss cases under DirCC, the directory must first be contacted, which may involve network traffic if the relevant directory slice is not attached to the current core. The relevant cache line must be brought to the L1 cache for all types of access, but the protocol actions that must be taken and the associated latency depend on the kind of request (read or write), as well on whether any other L1 caches contain the data. Reads and writes for lines that are not cached in any L1 (i.e., the directory entry is *invalid*) simply query the directory and concurrently access the next level in the cache hierarchy to retrieve the cache line; the same is true for reads of lines cached in *shared* state in some per-core L1, because the directory does not store the actual cache line:

$$\begin{aligned}\text{cost}_{\text{rdI,wrI,rdS}} &= \text{rate}_{\text{core miss}} \times \text{cost}_{\rightarrow,\text{addr}} \\ &\quad + \max(\text{cost}_{\text{dir lookup}}, \text{cost}_{\text{L2\$ request}}) \\ &\quad + \text{rate}_{\text{core miss}} \times \text{cost}_{\rightarrow,\text{cacheline}} + \text{cost}_{\text{L1\$ insert}} \\ &= 2\% \times 37 + 9.6 + 2\% \times 38 + 3 = 14.1 \text{ cycles.}\end{aligned}$$

(The fairly low 2% core miss rate we measured in simulation results from assigning address ranges to L2 slices and directories using a page-granularity first-touch heuristic that keeps data close to its accessors.)

Exclusive access (write) requests to lines that are in *shared* state elsewhere additionally contact the sharer(s) to invalidate their copies and wait for their invalidate acknowledgements; assuming that the messages to all sharers are all sent in parallel, we have:

$$\begin{aligned}\text{cost}_{\text{wrS}} &= \text{rate}_{\text{core miss}} \times \text{cost}_{\rightarrow,\text{addr}} \\ &\quad + \max(\text{cost}_{\text{dir lookup}}, \text{cost}_{\text{L2\$ request}}) \\ &\quad + \text{cost}_{\rightarrow,\text{addr}} + \text{cost}_{\text{L1\$ inv}} + \text{cost}_{\rightarrow,\text{ack}} \\ &\quad + \text{rate}_{\text{core miss}} \times \text{cost}_{\rightarrow,\text{cacheline}} + \text{cost}_{\text{L1\$ insert}} \\ &= 2\% \times 37 + 9.6 + 37 + 3 + 37 + 2\% \times 38 + 3 = 91.1 \text{ cycles.}\end{aligned}$$

Read requests for data that is cached in *modified* state by another L1 cache must flush the

modified line from the cache that holds it, write it back to L2 (to satisfy future read requests from other caches), and only then send the cache line back to the client (note that, in this case, the directory access cannot be amortized with the L2 request because the data to be written must first arrive from the L1 cache):

$$\begin{aligned}\text{cost}_{\text{rdM}} &= \text{rate}_{\text{core miss}} \times \text{cost}_{\rightarrow,\text{addr}} + \text{cost}_{\text{dir lookup}} \\ &\quad + \text{cost}_{\rightarrow,\text{addr}} + \text{cost}_{\text{L1\$ flush}} + \text{cost}_{\rightarrow,\text{cacheline}} + \text{cost}_{\text{L2\$ write}} \\ &\quad + \text{rate}_{\text{core miss}} \times \text{cost}_{\rightarrow,\text{cacheline}} + \text{cost}_{\text{L1\$ insert}} \\ &= 2\% \times 37 + 2 + 37 + 3 + 38 + 9 + 2\% \times 38 + 3 \\ &= 93.5 \text{ cycles.}\end{aligned}$$

Writes to data in modified state are similar but can skip the L2 write (because the data is about to be written and the L2 copy would be stale anyway) and directly send the flushed cache line to the client cache:

$$\begin{aligned}\text{cost}_{\text{wrM}} &= \text{rate}_{\text{core miss}} \times \text{cost}_{\rightarrow,\text{addr}} + \text{cost}_{\text{dir lookup}} \\ &\quad + \text{cost}_{\rightarrow,\text{addr}} + \text{cost}_{\text{L1\$ flush}} + \text{cost}_{\rightarrow,\text{cacheline}} \\ &\quad + \text{rate}_{\text{core miss}} \times \text{cost}_{\rightarrow,\text{cacheline}} + \text{cost}_{\text{L1\$ insert}} \\ &= 2\% \times 37 + 2 + 37 + 3 + 38 + 2\% \times 38 + 3 = 84.5 \text{ cycles.}\end{aligned}$$

To estimate the occurrence rates of these state transitions, we simulated several SPLASH-2 benchmarks (Woo et al., 1995) in the Pin-based multicore simulator GRAPHITE (Miller et al., 2010), and obtained the rates shown in Table 4.1. Given those, the overall L1 miss penalty for DirCC becomes:

$$\begin{aligned}\text{cost}_{\text{L1\$ miss, CC}} &= \text{rate}_{\text{rdI,wrI,rdS}} \times \text{cost}_{\text{rdI,wrI,rdS}} \\ &\quad + \text{rate}_{\text{wrS}} \times \text{cost}_{\text{wrS}} + \text{rate}_{\text{rdM,wrM}} \times \text{cost}_{\text{rdM,wrM}} \\ &= 85\% \times 14.1 + 5\% \times 91.1 + 10\% \times 93.5 + 0\% \times 84.5 \\ &= 25.9 \text{ cycles.}\end{aligned}$$

Overall average memory latency

Under DirCC, the overall average memory latency (AML) comprises the L1 cache access and the pro-rated cost of the L1 miss:

$$\begin{aligned}\text{AML}_{\text{CC}} &= \text{cost}_{\text{L1\$ access}} + \text{rate}_{\text{L1\$ miss, CC}} \times \text{cost}_{\text{L1\$ miss, CC}} \\ &= 2 + 6\% \times 25.9 = 3.56 \text{ cycles.}\end{aligned}$$

For EM0, the AML incorporates the cost of the L1 request (be it a hit or a miss) and the thread migration delay in the event of a core miss:

$$\begin{aligned}\text{AML}_{\text{EM0}} &= \text{cost}_{\text{L1\$ access}} + \text{rate}_{\text{L1\$ miss}} \times \text{cost}_{\text{L1\$ miss, EM0}} \\ &\quad + \text{rate}_{\text{core miss}} \times \text{cost}_{\rightarrow,\text{thread}} \\ &= 2 + 6\% \times 12.6 + 2\% \times 44 = 3.63 \text{ cycles.}\end{aligned}$$

Under RA, the core miss overhead involves a round-trip message that depends on the type of access (read or write):

$$\begin{aligned}\text{cost}_{\text{core miss, RA}} &= \text{rate}_{\text{read}} \times \left(\text{cost}_{\rightarrow,\text{addr}} + \text{cost}_{\rightarrow,\text{value}}\right) \\ &\quad + \text{rate}_{\text{write}} \times \left(\text{cost}_{\rightarrow,\text{addr\&value}} + \text{cost}_{\rightarrow,\text{ack}}\right) \\ &= 70\% \times (37 + 37) + 30\% \times (37 + 37) = 74 \text{ cycles,}\end{aligned}$$

giving a total average latency of:

$$\begin{aligned}\text{AML}_{\text{RA}} &= \text{cost}_{\text{L1\$ access}} + \text{rate}_{\text{L1\$ miss}} \times \text{cost}_{\text{L1\$ miss, RA}} \\ &\quad + \text{rate}_{\text{core miss}} \times \text{cost}_{\text{core miss, RA}} \\ &= 2 + 6\% \times 12.6 + 2\% \times 74 = 4.23 \text{ cycles.}\end{aligned}$$

Although the analytical AML model is necessarily simplified (interactions among multiple accesses are not considered, additional coherence states such as E or O are not modeled, etc.), it can nevertheless provide intuition about when EM0 performs well and when it performs poorly. We next vary some of the model parameters from their defaults in Table 4.1, and examine how this affects the AML for each implementation (Figure 4-3).

We first examined the effect of different L1 cache miss rates on performance. Varying $\text{rate}_{\text{L1\$ miss}}$ has the most striking effect under the DirCC model, which incurs relatively expensive coherence protocol costs (of two or more interconnect round-trips plus directory and often invalidation overheads) for every L1 miss (Figure 4-3, top left); while the other three protocols also degrade with increasing miss rates, the differences are not as dramatic.

Of course, L1 cache miss rates under these protocols are not *directly* comparable: on the one hand, DirCC can make multiple copies of data and is more frequently able to hit the local cache; on the other hand, EM0 and RA do not pollute their L1 caches with multiple copies of the same data, and feature higher total L1 cache capacities. Nevertheless, comparisons within each protocol are valid, and we can safely use the figure to examine how the protocols tend to *degrade* when cache miss rates grow.

In all protocols, keeping core miss rates low via efficient data placement is critical to

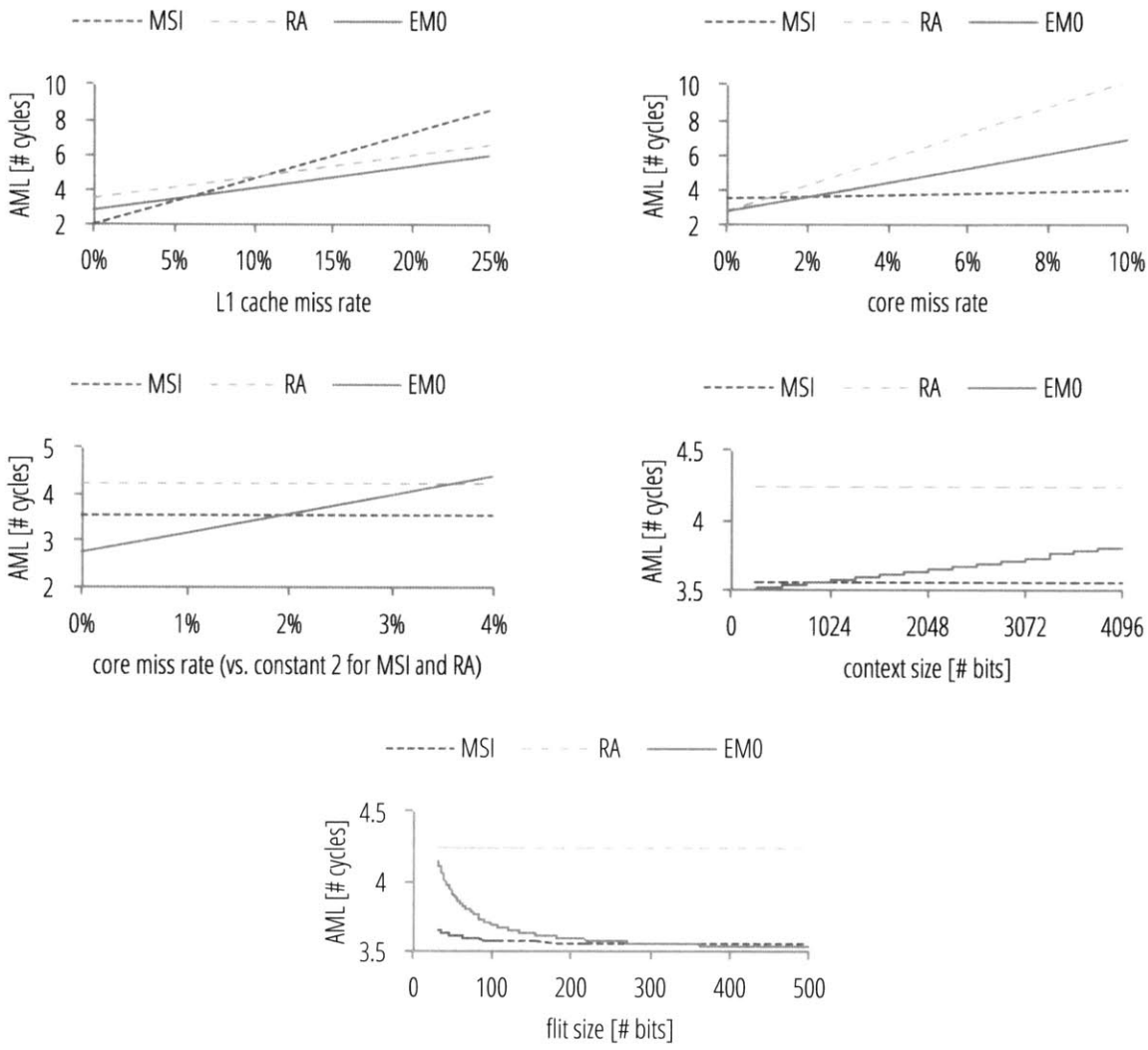


Figure 4-3: Average memory latency (AML) of a single memory access (as estimated by the analytical models of DirCC, RA, and EM0) demonstrates where EM0 performs well/poorly.

accessing the shared NUCA L2 cache efficiently. In EM0 and RA, however, a core miss may occur even if the data resides on-chip in some L1 cache (because each address may only be cached at a specific *home core*). Figure 4-3 (top right) varies $\text{rate}_{\text{core miss}}$ and illustrates that EM0 and RA perform well as long as the core miss rate is low (below about 2%); when core miss rates are high, however, the high rate of migrations (for EM0) and remote accesses (for RA) means that the ability of DirCC to replicate data allows it to dominate.

Again, comparing core miss rates directly can be somewhat misleading, especially for RA and EM0. This is because under RA threads do not move from their original cores, and each access to a remote core's cache constitutes a core miss; under EM0, however, the thread migrates to the relevant home core on the first access and subsequent accesses to the same core are no longer core misses but an access to the thread's original core would now become a core miss. The effect, then, depends on the amount of spatiotemporal locality in the access pattern: with high locality, EM0 core miss rates will be substantially lower than the RA equivalent. To examine the potential, we varied $\text{rate}_{\text{core miss}}$ for EM0 *only*, keeping it at the default of 2% for the other protocols (Figure 4-3, center left): as expected, lower core miss rates favor EM0, while with higher core miss rates EM0 migration overheads cause its performance to deteriorate.

Finally, the execution contexts migrated in EM0 are substantially larger than both coherence messages and cache-line sizes; this scheme is therefore particularly sensitive to both architectural context size and network bandwidth. To examine the context size variations, we directly varied $\text{size}_{\text{context}}$ (Figure 4-3, center right); to simulate different network bandwidths, we varied the size of a flit that can be transferred in one cycle (Figure 4-3, bottom). Both illustrate that EM0 is sensitive to network bandwidth: for both very large contexts and very low on-chip network bandwidths, EM0 migration latencies become too large to offer much benefit over the round-trips required by the other protocols.

Overall, we have identified two significant weaknesses of EM0: one is its dependence on consistently high spatiotemporal locality (i.e., on keeping core miss rates low), and the other is its sensitivity to migrated context sizes. We will ameliorate the first weakness in the following section by amending the shared memory model itself, and address the second limitation when we discuss a migration-based architecture in Chapter 6.

4.3. AN OPTIMIZED SHARED MEMORY SCHEME

To investigate how much spatiotemporal locality can be expected from standard benchmarks, we simulated several SPLASH-2 benchmarks (Woo et al., 1995) in GRAPHITE (Miller et al., 2010) under the EM0 protocol. Figure 4-4 shows the percentage of memory instructions that result in migrations: the resulting migration rates, at nearly 40%, are prohibitive. To understand

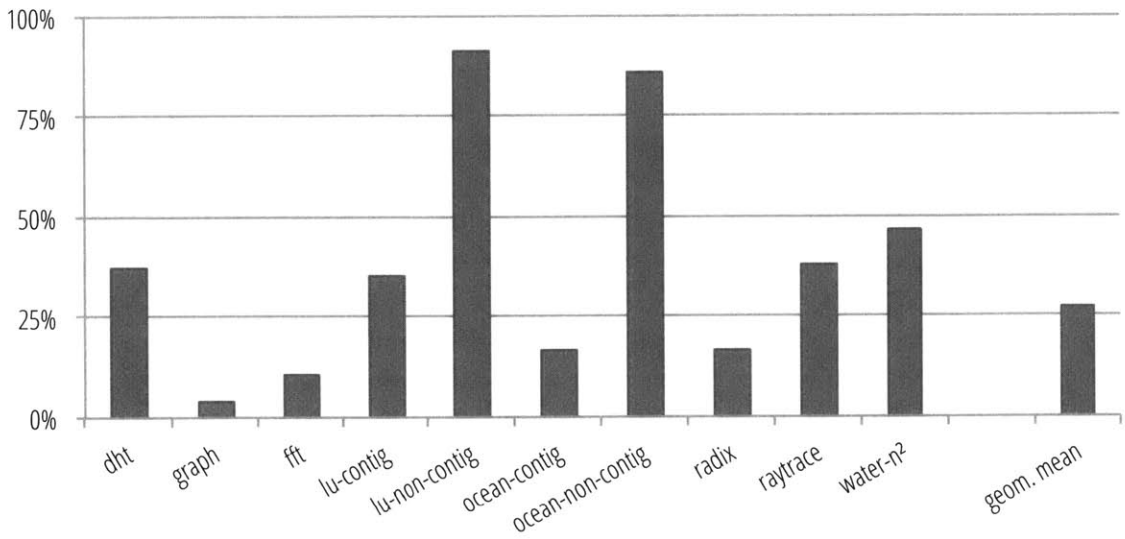


Figure 4-4: Shared memory based on the basic EM0 protocol incurs high migration rates on a set of benchmarks.

the source of this high migration rate, we then counted the number of times an application *contiguously* accessed the same region of memory (i.e., memory assigned to a single per-core cache); the results appear in Figure 4-5. Applications tend to exhibit varying degrees of locality: some applications consist mainly of medium to long (≥ 5) streaks of references to the same region, while some have a high proportion of shorter streaks (Figure 4-5, top). Indeed, locality varies dramatically even within a *single* application, with accesses divided bimodally between very short streaks and fairly long runs (Figure 4-5, bottom).

Since these results were obtained from applications compiled using a stock C compiler and therefore were not specifically optimized to exploit fine-grained spatiotemporal locality,[1] they illustrate a lower bound on the fine-grained locality that can be exploited in applications: cautious reordering of memory accesses by the compiler (or, potentially, hardware) would likely increase the amount of locality available. In addition, even in applications designed to take advantage of cache coherence, some locality optimization effects come "for free" as a side effect of other memory layout optimizations (cf. the locality differences in the contiguous and non-contiguous versions of LU and OCEAN in Figure 4-5).

As we saw in the previous section, EM0 works very well when memory accesses come in long contiguous runs and core miss rates are low, but performs poorly for the case where only

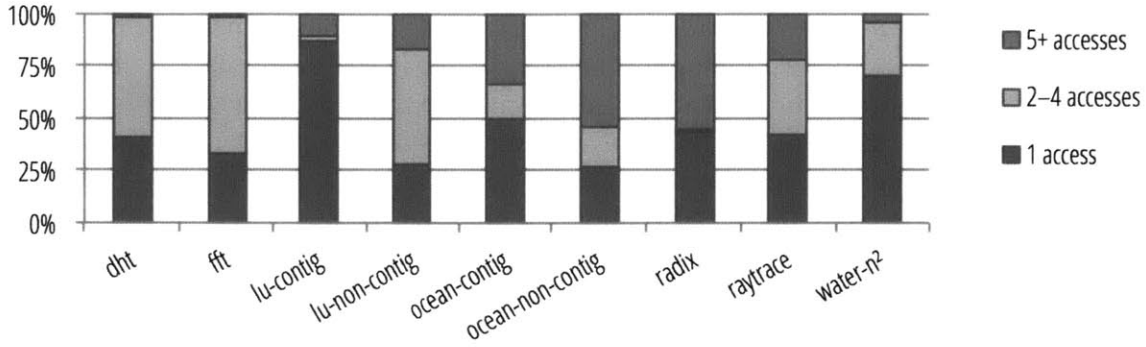
[1]The compiler transformations required to implement such optimizations, as well as their correctness under various memory models, are beyond the scope of this dissertation.

one or two accesses are made to the same region. This is because the overheads of packing a running thread context onto the interconnect and unpacking it at the destination core are not amortized with only a few contiguous accesses. To optimize these short-run-length patterns, therefore, we will turn them into round-trip remote cache accesses (RA): although several of these are slower than one migration followed by local accesses, they become faster if a migrated thread accesses only one address and then migrates back.

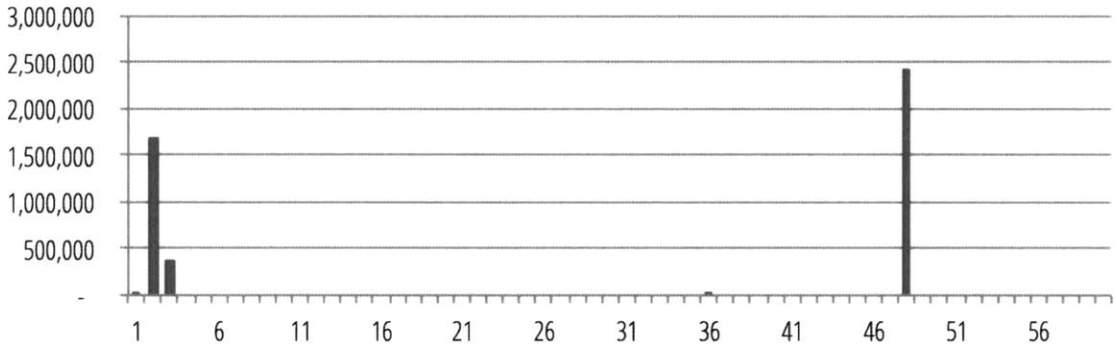
The resulting hybrid shared memory scheme, which we denote EM, is shown in Figure 4-6. Specifically, the protocol for accessing address A by thread T executing on core C is as follows:

1. compute the *home* core H for A (e.g., by masking the appropriate bits);
2. if H = C (a *core hit*),
   (a) forward the request for A to the cache hierarchy (possibly resulting in a DRAM access);
3. if H ≠ C (a *core miss*), and instruction will execute as a remote access,
   (a) send a remote access request for address A to core H,
   (b) when the request arrives at H, forward it to H's cache hierarchy (possibly resulting in a DRAM access),
   (c) when the cache access completes, send a response back to C,
   (d) once the response arrives at C, continue execution.
4. if H ≠ C (a *core miss*), and the instruction will cause a migration,
   (a) interrupt the execution of the thread on C (as for a precise exception),
   (b) migrate the microarchitectural state to H via the on-chip interconnect:
      i. if H is the native core for T, place it in the native context slot;
      ii. otherwise, if the guest slot on H contains another thread T′, evict T′ to its native core N′;[2] next, move T into the guest slot for H;
   (c) resume execution of T on H, requesting A from its cache hierarchy (and potentially accessing DRAM).

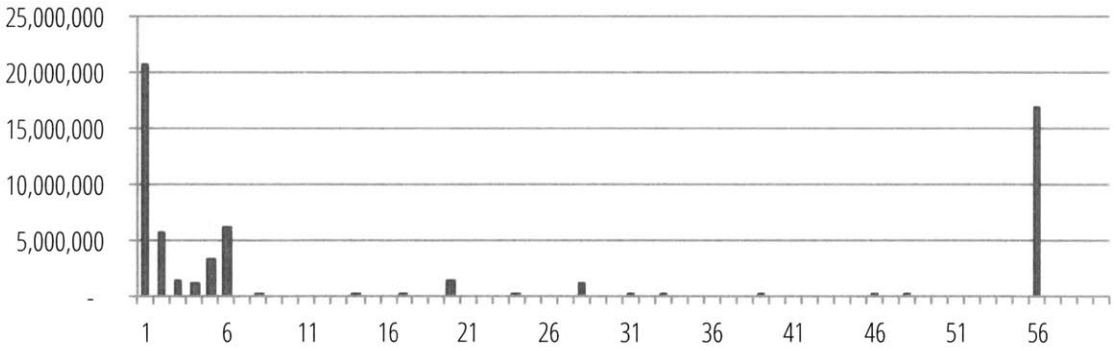
[2]Evictions must wait for any outstanding remote accesses to complete in addition to waiting for DRAM → cache responses.



(a) breakdown of non-local memory access run lengths



(b) LU-CONTIGUOUS memory references binned into run lengths: nearly 2.5 million accesses were part of contiguous 48-access sequences



(c) OCEAN-CONTIGUOUS memory references binned into run lengths: e.g., nearly 17 million accesses were part of contiguous 56-access sequences

Figure 4-5: The number of times a computation contiguously accesses the same memory region varies greatly across benchmarks (top) and even within a single benchmark (bottom). GRAPHITE simulation, 256 cores.

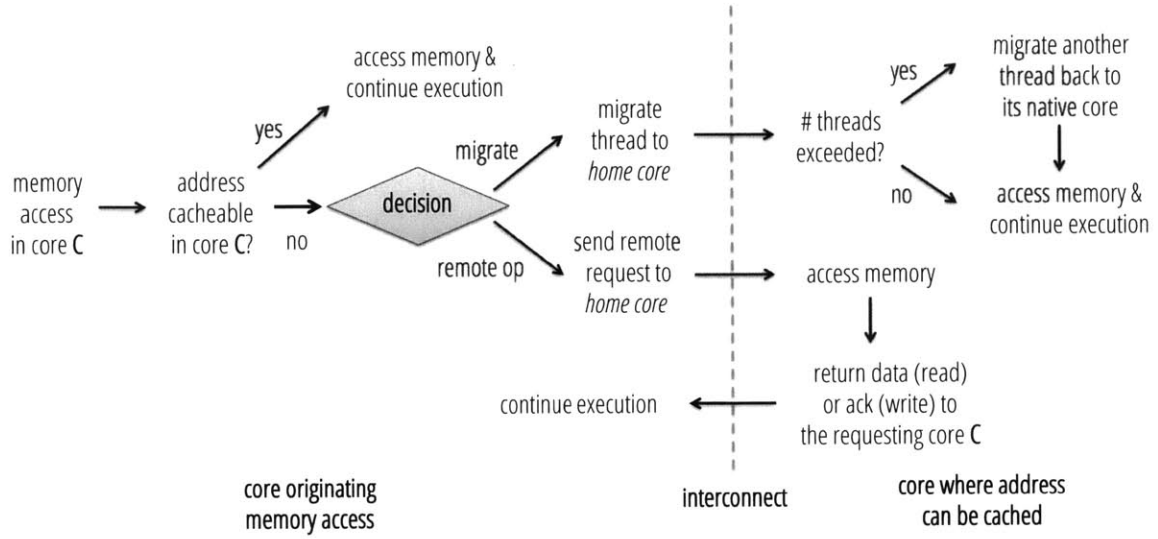


Figure 4-6: The life of a memory access under the optimized EM migration-based shared memory scheme.

Although the hybrid EM protocol can potentially handle both long and short runs of contiguous accesses, its effectiveness hinges on the *decision procedure* that determines whether a particular memory request should result in a migration or a remote cache access. We will return to this subject in Chapter 5.

4.4. VIRTUAL MEMORY AND OS IMPLICATIONS

Although the architecture we describe here (see Chapter 6) follows the accelerator model and, lacking virtual memory support, does not require a full operating system, fine-grained migration can be equally well implemented in a full-fledged CPU architecture. Virtual addressing at first sight potentially delays the local-vs-remote decision by one cycle (since the physical address must be resolved via a TLB lookup), but in a distributed shared cache architecture this lookup is already required to resolve which tile caches the data (if the L1 cache is virtually addressed, this lookup can proceed in parallel with the L1 access as usual). Program-initiated OS system calls and device access occasionally require that the thread remain pinned to a core for some number of instructions; these can be accomplished by migrating the thread to

its native context on the relevant instruction.[3] OS-initiated tasks such as process scheduling and load rebalancing typically take place at a granularity of many milliseconds, and can be supported by requiring each thread to return to its native core every so often.

[3]In fact, our ASIC implementation uses this approach to allow the program to access various statistics tables.

CHAPTER 5

MIGRATION PREDICTION

GIVEN that applications exhibit a mix of single accesses and longer streaks of accesses to the same memory region, our next task is to devise a method for deciding whether a given memory access should be executed as a remote cache access or as a thread migration. Because this decision must be taken quickly enough to take advantage of short bursts of locality (on the order of 20–50 accesses), it must be implemented in hardware.

5.1. LEARNING AND PREDICTING LOCALITY

The hardware module we contemplate here must *predict* ahead of time whether or not a given memory reference will begin a long streak of locality. The key observation here is that memory locality often arises from loops, and as such is correlated with a specific instruction. This is the same phenomenon that underlies branch prediction mechanisms, and, indeed, our strategy for detecting it will be similar: for each instruction, we will count the number of "contiguous" accesses that follow, and, if these exceed a threshold, we will learn that this instruction should migrate.

At a high level, the prediction mechanism operates as follows:

1. when a program first starts execution, each memory access to a remote cache defaults to the baseline RA mechanism (round-trip access);

2. as execution continues, the core monitors the home core information for each memory access, and remembers the PC of the first instruction of every multiple-access sequence to the same home core;

3. if the length of the sequence exceeds a threshold, the instruction address is either considered *migratory* (and inserted into a predictor table), or non-migratory (and poten-

tially removed from the predictor;

4. the next time a thread executes the instruction, it migrates to the home core if it is a migratory instruction (a "hit" in the predictor table), and performs a remote access if it is a remote-access instruction (a "miss" in the predictor).

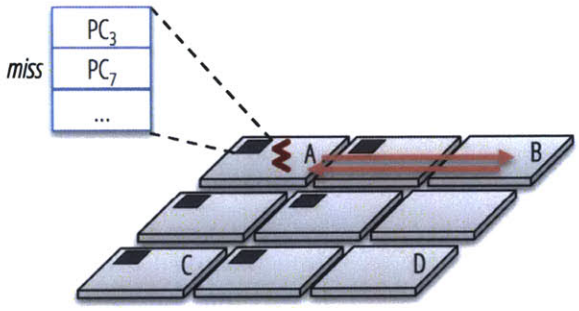
To accomplish this, each thread tracks three pieces of information about the current run of memory accesses: (a) *home*, which tracks the home core ID where the current memory address can be cached, (b) *depth*, which indicates how many times so far this thread has contiguously accessed memory at the current *home* location, and (c) *start PC*, which remembers the PC of the very first instruction in the current sequence that accessed *home*. The prediction mechanism is parametrized by the depth threshold θ, which determines how many contiguous accesses must be made for an instruction to be considered migratory. Migratory instruction PCs are stored in a table—similar to a branch predictor table—that is consulted to determine whether or not a given memory instruction should migrate.

More precisely, when a thread T executes a memory instruction for address A whose PC is P, it must first find the home core H for A; then,

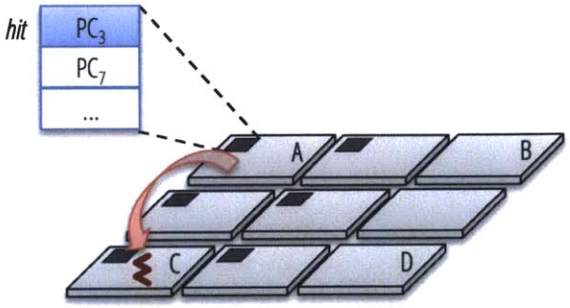
1. if home = H (i.e., memory access to the same home core as that of the previous memory access),
   (a) if depth < θ,
      i. increment depth by one; then if depth = θ, start PC is considered a migratory instruction and inserted into the migration predictor table;
2. if home ≠ H (i.e., a new sequence starts with a new home core),
   (a) if depth < θ,
      i. start PC is considered a remote-access instruction, and removed from the migration predictor table;
   (b) reset the entry (i.e., home = H, PC = P, depth = 1).

Table 5.1 shows an example of the detection mechanism when $\theta = 2$. Suppose a thread executes a sequence of memory instructions, I_1 through I_7.[1] The PCs of I_1 to I_7 are PC_1 through PC_7, respectively, and the home core for the memory address that each instruction accesses is specified next to each PC. When I_1 is first executed, the mechanism considers it a possible start of a contiguous sequence, and the tracker variables {*home*, *depth*, *start PC*} are set to {A, 1, PC_1}. Since the home core (B) of the next instruction I_2 is different from the *home* from

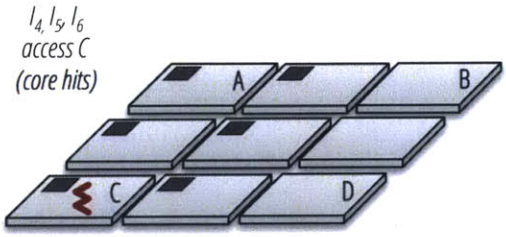
[1]Non-memory instructions do not affect the predictor mechanism and are not shown here.



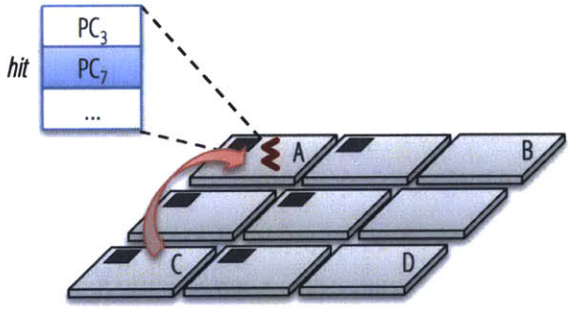
(a) predictor table miss: remote access



(b) predictor table hit: migration



(c) accesses to C become core hits post-migration



(d) another predictor hit: migration

Figure 5-1: The sequence of migrations and remote accesses that result once the predictor has been trained as in Table 5.1.

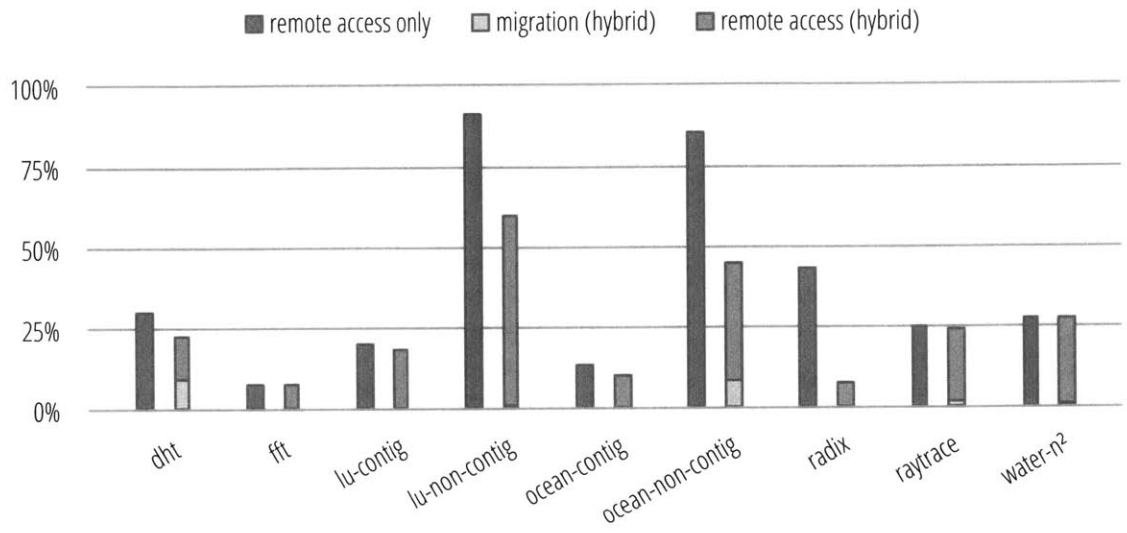


Figure 5-2: Core miss rates for various remote cache access modes. The migration predictor is able to successfully detect long runs of contiguous accesses and turn them into a small number of migrations (dark gray) and remote cache accesses (light gray), reducing the overall core miss rates. GRAPHITE simulation, 256 cores, predictor threshold $\theta = 3$.

the previous instruction, executing I_2 causes the tracker to be reset with I_2 as a sequence start point. At the same time, because the *depth* of the contiguous accesses to home core A has not reached the depth threshold θ, the previously tracked sequence (starting at PC_1) is not considered migratory (and, if it has an entry in the predictor table, its entry is removed). This tracker reset occurs again for I_3. When I_4 is executed, however, it accesses the same home core as I_3 (C) and thus the *depth* field is incremented by one; since *depth* has now reached the threshold $\theta = 2$, *start PC* is classified as a migratory instruction and added to the migration predictor table. For I_5 and I_6 which keep accessing the same home core C, we need not update the entry because θ has already been exceeded. Finally, when I_7 is executed, the predictor again resets the tracker and starts a new sequence of accesses to home core A, starting with PC_7. Next time this instruction sequence occurs, the migration predictor will direct the thread to migrate to core C at I_3, and again to core A at PC_7; Figure 5-1 illustrates this.

5.2. PREDICTION EFFECTIVENESS

To evaluate how the predictor performs in practice, we used GRAPHITE to simulate the same set of benchmarks we used to investigate access locality in Figure 4-5, both in a remote-access-only distributed shared cache system (RA) and a system that implements the hybrid of migration/remote access protocol from Section 4.3 (EM). Overall, the predictor reduces the total core miss rate from 38% on average[2] under RA to 25% on average under EM, a 35% improvement in data locality. The breakdown of each bar also shows that a large fraction of remote accesses have been successfully replaced with a much smaller number of migrations: for OCEAN_NON_CONTIGUOUS, for example, a 86% remote access rate under RA turns into a 45% core miss rate with only a small number of migrations. Importantly, the migration predictor also keeps the overall migration low, with an average migration rate of 3% over all benchmarks.

5.3. PARTIAL-CONTEXT MIGRATION

Finally, we re-examine the sensitivity of migration-based shared memory to the size of the thread context being migrated relative to on-chip interconnect bandwidth (see Section 4.2). While the hybrid EM protocol (Section 4.3) and the basic migration predictor lower migration *rates* and reduce the overall impact of transferring large thread contexts, further reducing that cost would positively affect both on-chip traffic levels (and consequently dynamic power dissipation) and on-chip interconnect congestion.

With this in mind, we turn to examine the possibility of reducing the *size* of migrations. Figure 5-3 shows the number of registers that are read and written under the EM protocol when a thread migrates from its native context: for the benchmarks we simulated, only a subset of the general-purpose registers were used when threads executed outside their native core. This immediately suggests a strategy for further reducing migration sizes: instead of migrating the entire thread context, we should only transfer the subset that will be used at the thread's destination.

At a minimum, this subset consists of the program counter (PC); this allows the remote core to fetch the next instruction to be executed. In practice, however, the computation at the remote core requires some of the registers: for example, if the instruction migrates to access a memory location, it will at a minimum need the register containing the relevant address.

Migrating only a subset of the context leaves the question of what to do with the parts of the thread context that are not migrated. Fortunately, the ENC protocol that we developed

[2]The averages here are geometric means.

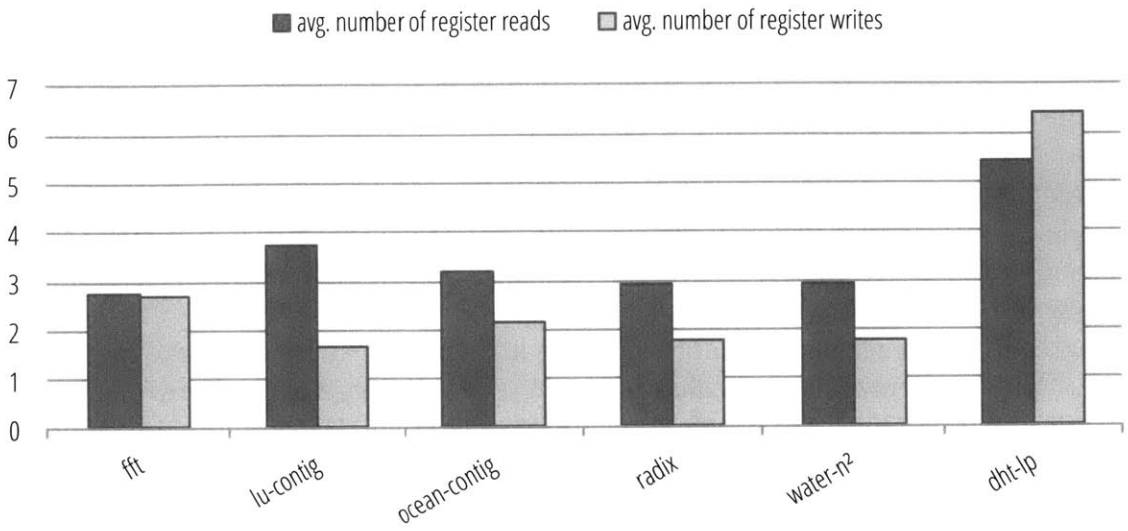


Figure 5-3: The number of general-purpose registers (out of 16) read (dark gray) and written (light gray) after a migration (average over the entire run). Only a few of the registers are used on the remote core. GRAPHITE simulation, 256 cores.

to ensure deadlock-free migrations provides an elegant answer with the exclusive context reserved at each core for the core's native thread(s): a native thread's context can be simply left dormant at its core while the relevant parts of the thread context migrate and execute in another core's guest context. In the next chapter, we consider a core architecture that elegantly supports partial-context migrations, and show how to predict which part of the context should migrate.

memory instruction			present state			next state			action
instr.	PC	home	home	depth	start PC	home	depth	start PC	
I_1:	PC_1	A	—	—	—	A	1	PC_1	reset the entry for a new sequence starting from PC_1
I_2:	PC_2	B	A	1	PC_1	B	1	PC_2	reset the entry for a new sequence starting from PC_2 (evict PC_1 from the predictor if it exists)
I_3:	PC_3	C	B	1	PC_2	C	1	PC_3	reset the entry for a new sequence starting from PC_3 (evict PC_2 from the predictor if it exists)
I_4:	PC_4	C	C	1	PC_3	C	2	PC_3	increment depth by one, insert PC_3 into the predictor table
I_5:	PC_5	C	C	2	PC_3	C	2	PC_3	do nothing (sequence already inserted)
I_6:	PC_6	C	C	2	PC_3	C	2	PC_3	do nothing (sequence already inserted)
I_7:	PC_7	A	C	2	PC_3	A	1	PC_7	reset the entry for a new sequence starting from PC_7

Table 5.1: An example of how the migration predictor learns to migrate (here the depth threshold $\theta = 2$).

CHAPTER 6

EM²: A THREAD MIGRATION ARCHITECTURE

This chapter introduces the Execution Migration Machine (EM²) architecture, a 110-core shared-memory multicore processor that embodies the hardware-level thread migration and migration-based shared memory ideas developed in Chapters 3 and 4.

6.1. SYSTEM ARCHITECTURE

The silicon implementation we describe in this dissertation consists of 110 homogeneous tiles placed on a 10×11 grid and connected via an on-chip network. Each core consists of a stack-architecture processor core (see Section 6.2), instruction and data caches, and on-chip interconnect routers. In lieu of a DRAM interface, our test chip exposes the two networks that carry off-chip memory traffic via a programmable rate-matching interface; this, in turn, connects to a maximum of 16GB of DRAM via a controller implemented in an FPGA.[1]

Tiles are connected by six independent on-chip networks: two networks carry migration/eviction traffic, another two carry remote-access requests/responses, and a further two external DRAM requests/responses; in each case, two networks ensure deadlock-free operation.

The networks are arranged in a 2D mesh geometry: each tile contains six Network-on-Chip (NoC) routers which link to the corresponding routers in the neighboring tiles. Each network carries 64-bit flits using wormhole flow control and dimension order routing. The routers are ingress-buffered, and are capable of single-cycle forwarding under congestion-free conditions.

[1] A practical chip would, of course, have a number of DRAM controllers on-die; since our chip is experimental and focuses on on-chip memory accesses and on-chip network traffic, however, we chose to reduce complexity and lower risk by effectively making the memory controller external.

The memory subsystem consists of a single level (L1) of instruction and data caches, and a backing store implemented in external DRAM. Each tile contains an 8KB read-only instruction cache and a 32KB data cache, for a total of 4.4MB on-chip cache capacity; the caches are capable of single-cycle read hits and two-cycle write hits. The first 86% $\left(=\frac{110}{128}\right)$ of the entire memory address space of 16GB is divided into 110 non-overlapping regions as required by the EM shared memory semantics (see Chapter 4), and each tile's data cache may only cache the address range assigned to it; a further 14% of the address range is cacheable by any tile but without any hardware-level coherence guarantees. In addition to serving local and remote requests for the address range assigned to it, the data cache block also provides an interface to remote caches via the remote-access protocol (Figure 6-1). Memory is word-addressable and there is no virtual address translation; cache lines are 32 bytes.

6.2. A STACK MACHINE CORE

To simplify the implementation of partial context migration, our architecture contains a custom stack-based core (Figure 6-2); in our ASIC implementation, the word width is 32 bits. Although migration can equally well be implemented in a register-based architecture, a stack-based core has several advantages that simplify implementation. Firstly, since the stack is accessed from the top (as opposed to a random-access register file), the values most likely to be used soon tend to percolate to the top of the stack; if the migrating thread, then, migrates only the top part of the stack (down to some depth), it stands a good chance of those entries being the ones that will be used in the remote core. Moreover, the amount of the context to transfer can be easily controlled with a single parameter, i.e., the depth of the stack to migrate (i.e., the number of stack entries from the top of the stack); this is considerably easier to specify (and predict) than what subset of registers should be migrated. In a sense, the programmer has already specified what entries to migrate by keeping them on top of the stack.

To ease programming, our core contains two stacks: a main stack and an auxiliary stack. Most instructions (arithmetic operations, memory loads and stores, etc) operate on the main stack, while the auxiliary stack can be accessed by copying or moving entries to and from the main stack. This is very useful for storing and quickly accessing function call return addresses, memory base addresses, loop termination conditions, and other data that would otherwise have to be stored in memory.

For convenience, both stacks, which consist of in-core registers, are automatically backed by the data memory. That is, when the in-core stack overflows (i.e., data are pushed onto it when the stack is full), the hardware will automatically spill a cache line's worth of data into the data cache, and when the stack underflows (i.e., an attempt is made to read from an empty

stack), the hardware refills it from the appropriate line in the data cache hierarchy.

6.3. MIGRATION MECHANISM

Whenever a thread migrates out of its native core, it has the option of transmitting only the part of its thread context that it expects to use at the destination core. In each packet, the first (head) flit encodes the destination packet length as well as the thread's ID and the program counter, as well as the number of main stack and auxiliary stack elements in body flits that follow. The smallest useful migration packet consists of one head flit and one body flit which contains two 32-bit stack entries. Migrations from a guest context must transmit all of the occupied stack entries, since guest context stacks are not backed by memory.

Figure 6-3 illustrates how the processor cores and the on-chip network efficiently support fast instruction-granularity thread migration. When the core fetches an instruction that triggers a migration (for example, because of a memory access to data cached in a remote tile), the migration destination is computed and, if there is no network congestion, the migration packet's head flit is serialized into the on-chip router buffers in the same clock cycle. While the head flit transits the on-chip network, the remaining flits are serialized into the router buffer in a pipelined fashion. Once the packet has arrived at the destination NoC router and the destination core context is free, it is directly deserialized; the next instruction is fetched as soon as the program counter is available and the instruction cache access proceeds in parallel with the deserialization of the migrated stack entries. In our implementation, assuming a thread migrates H hops with B body flits, the overall thread migration latency amounts to 1 + H + 1 + B cycles from the time a migrating instruction is fetched at the source core to when the thread begins execution at the destination core. In the 110-core EM^2 implementation we describe here, H varies from 1 (nearest neighbor core) to 19 (the maximum number of hops for 10×11 mesh), and B varies from 1 (two main stack entries and no auxiliary stack entries) to 12 (sixteen main stack entries and eight auxiliary stack entries, two entries per flit); this results in the very low migration latency, ranging from the minimum of 4 cycles to the maximum of 33 cycles (assuming no network congestion).[2]

While a native context is reserved for its native thread and therefore is always free when this thread arrives, a guest context might be executing another thread when a migration packet arrives. In this case, the newly arrived thread is buffered until the currently executing thread has had a chance to complete some (configurable) number of instructions; then, the active guest thread is evicted to make room for the newly arrived one. During the eviction

[2]Although it is possible to migrate with no main stack entries, this is unusual, because most instructions require one or two words on the stack to perform computations. The minimum latency in this case is still 4 cycles, because execution must wait for the I$ fetch to complete anyway.

process the entire active context is serialized just as in the case of a migration (the eviction network is used to avoid deadlock), and once the last flit of the eviction packet has entered the network the newly arrived thread is unloaded from the network and begins execution.

6.4. PARTIAL-CONTEXT MIGRATION PREDICTION

The migration predictors in EM^2 are based on the mechanism described in Chapter 5, with two differences: (a) the predictors also learn the optimal context size to transfer in each migration, and (b) removing entries from the predictor table is accomplished via a negative-feedback mechanism.

To learn how many stack entries to send when migrating from a native context at runtime, the native context keeps track of the *start PC* that caused the last migration. When the thread arrives *back* at its native core, it reports the reason for its return: when the thread migrated back because of stack overflow (or underflow), the stack transfer size of the corresponding *start PC* is decremented (or incremented) accordingly (see Figure 6-4). In this case, less (or more) of the stack will be brought along the next time around, eventually reducing the number of unnecessary migrations due to stack overflow and underflow.

The returning thread also reports the number of local memory instructions it executed at the core it originally migrated to. If the thread returns without having made θ accesses, the corresponding *start PC* is removed from the predictor table and the access sequence reverts to remote access (cf. Figure 5-1).[3] This allows the predictor to respond to runtime changes in program behavior.

[3]Returns caused by *evictions* from the remote core do not trigger removal, since the thread might have completed θ accesses had it not been evicted.

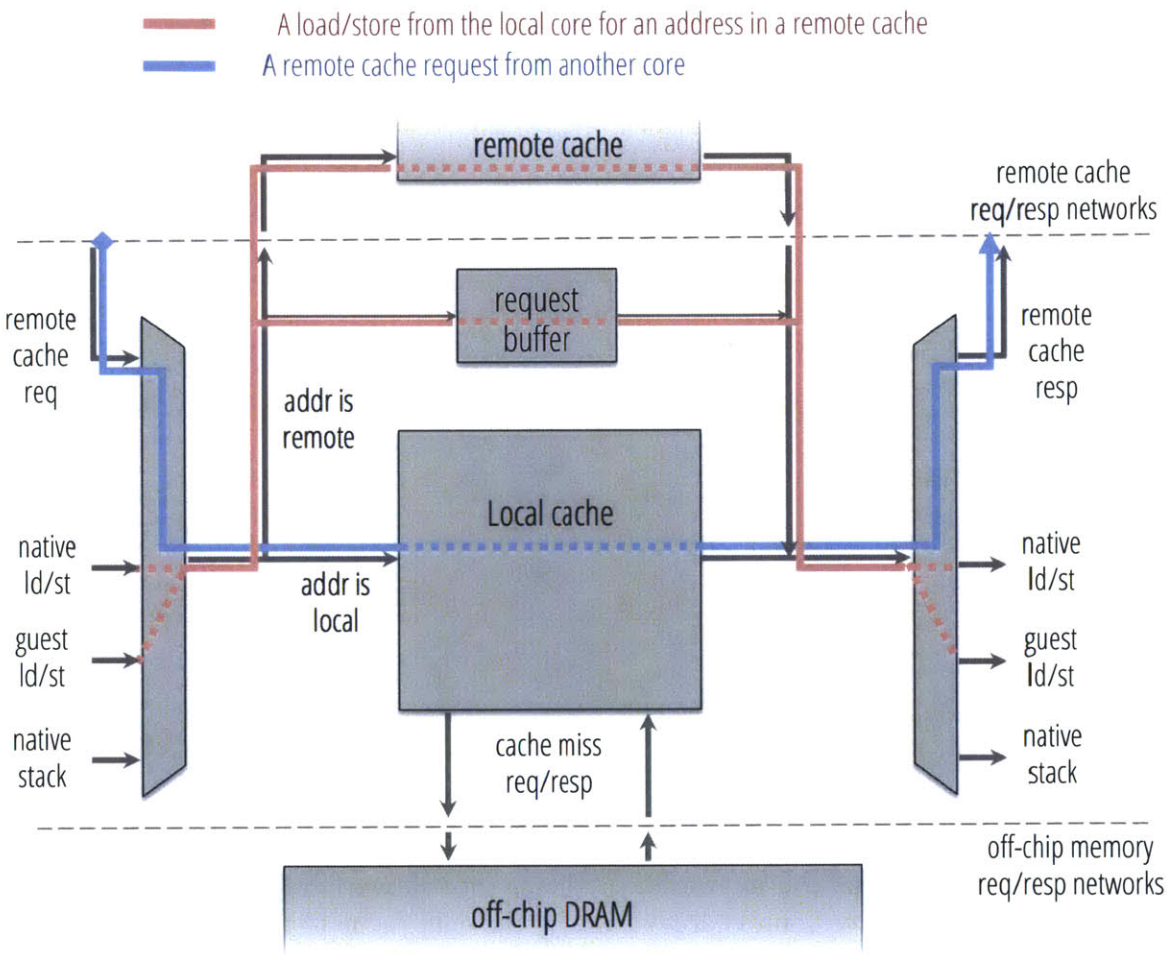


Figure 6-1: The data cache serves local load/store requests from native and guest contexts, stack spill/refill requests from the native context, and external load/store requests from remote cores.

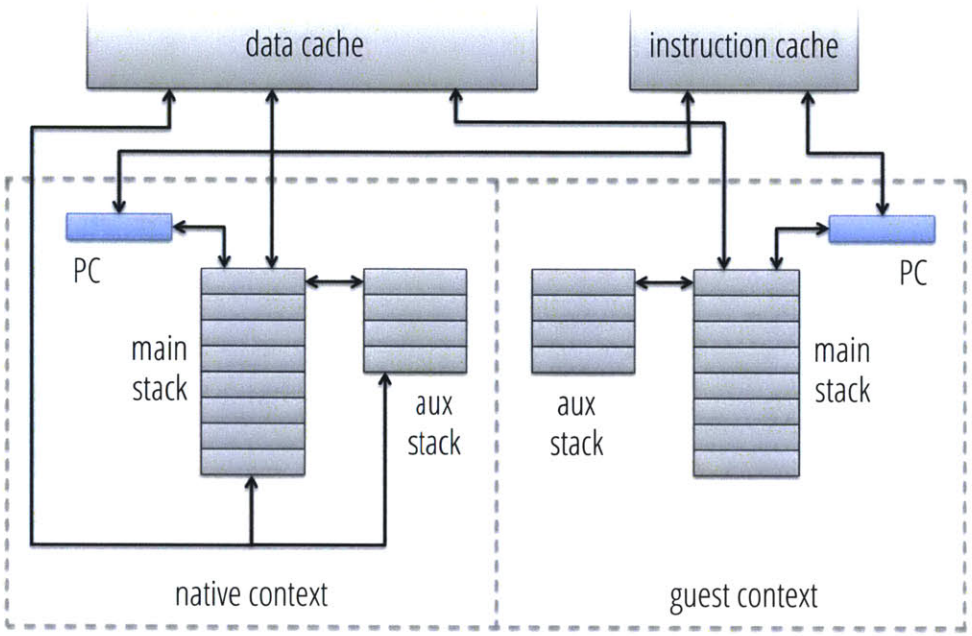


Figure 6-2: The processor core consists of two contexts in an SMT configuration, each of which comprises two stacks and a program counter, while the cache ports, migration network ports (not shown), and the migration predictor (not shown) are shared between the contexts. Stacks of the native context are backed by the data cache in the event of overflow or underflow.

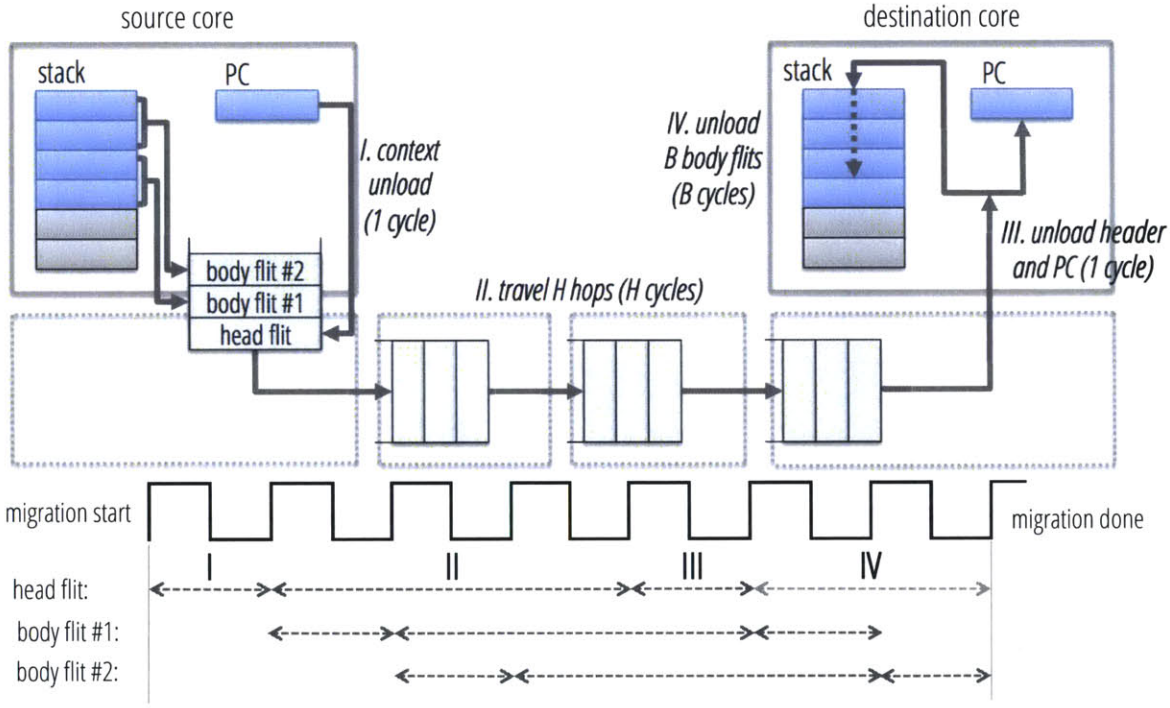
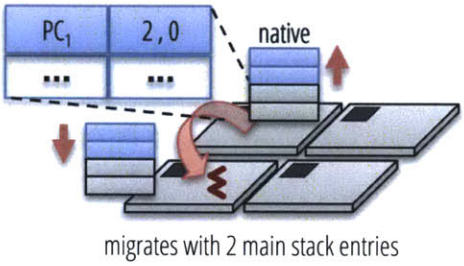
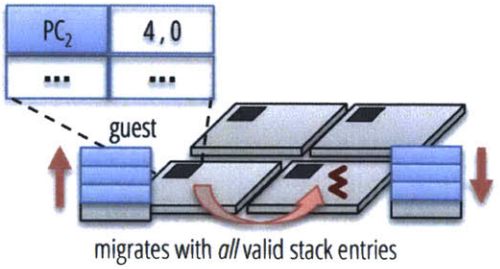


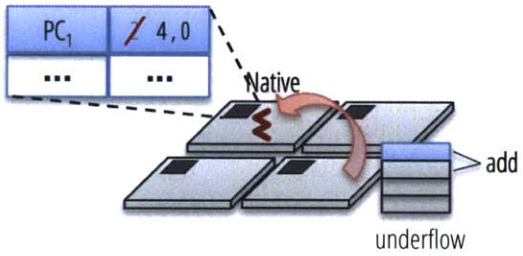
Figure 6-3: Migrations take a minimum of four cycles. (The auxiliary stack is omitted from the diagram for clarity.)



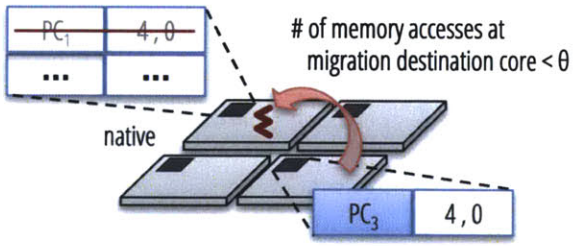
(a) migrations from native cores may transfer a partial thread context



(b) further migrations from guest cores must transfer the full context



(c) if a thread returns because of a stack underflow, it takes along more of the stack next time



(d) if the thread does not make enough accesses after migration, it is removed from the predictor

Figure 6-4: An amended migration predictor that learns how many stack entries to migrate.

CHAPTER 7

ASIC IMPLEMENTATION

In this chapter we describe the implementation, verification, and testing infrastructure used to implement EM² in a 45nm ASIC technology.

7.1. HARDWARE DESIGN LANGUAGE

Bluespec (2011) is a high-level hardware design language based on synthesizable guarded atomic actions (Hoe and Arvind, 2000). In Bluespec, a design is described using *rules*, each of which specifies the condition under which it is enabled (the guard) and the consequent state change (the action). Unlike standard Verilog `always` blocks, rules are *atomic*—in each clock cycle, a given rule will either be applied in its entirety or not at all, and the final state can be reconstructed as a sequence of rule applications. Actions in different rules may attempt to alter the same state element; when those changes conflict (e.g., two rules attempting to enqueue something in a single-port FIFO), the compiler automatically generates control logic (called a schedule) to ensure that only one of the conflicting rules may fire in a given clock cycle. When rule actions do not overlap or can be executed in parallel (e.g., one rule enqueuing into a FIFO and the other dequeuing from it), the compiler will allow the rules to fire in the same clock cycle. Source written in Bluespec is compiled to synthesizable Verilog, which is then combined with any custom Verilog/VHDL modules (such as SRAMs) and synthesized as part of the standard ASIC flow. The quality of results in terms of timing and area has been shown to be on par with hand-optimized Verilog RTL (Arvind et al., 2004).

The atomic rule semantics of Bluespec encourage a coding style where each semantically distinct operation is described separately, instead of a style that focuses on describing each hardware element (as in Verilog). For example, in our stack-ISA CPU, the operations of automatically refilling and spilling the in-CPU stack into backing data memory (`fill_stack` and `spill_stack`) both access the stack registers and the data cache interface,

but are described in two separate rules; the Bluespec compiler automatically creates the necessary data path muxes and control logic. Crucially, rule atomicity means that adding or removing a rule does not require any changes in existing rules: for example, the rule that completes an ALU operation (`alu_op`) also accesses the stack registers, but adding this rule requires no changes to `fill_stack` and `spill_stack`—the compiler will just infer slightly different muxes and control. This independence is handy in the design phase, and encourages a progressive-refinement approach to design "one rule at a time." Far more significantly, however, it means that implementation errors are localized to specific rules; thus, a bug-fix that changes `fill_stack` will not require changing or verifying `alu_op` even if the changes affect which stack registers are accessed and how, and the fix will work regardless of why the stack is being refilled (stack-to-stack computation, outbound migration, etc).

Less obvious but equally important, reasoning about our design in an operation-centric manner and expressing it using atomic rules allowed us to separate *functionality* and *performance*, and verify (and correct) the two aspects independently. By far most of our verification effort focused on functionality (i.e., the module being tested producing the correct output for any given input). Relying on the Bluespec compiler to automatically generate muxes and interlocks for conflicting rules, we did not have to think about the precise cycle-to-cycle operations or worry that concurrent execution of separate operations might combine to cause unexpected bugs. Once functionality was verified, we tuned the cycle-to-cycle operation to meet our performance goals, mostly by guiding the compiler with respect to rule priority and ordering and without changing any of the rules themselves. This separation of functionality and cycle-to-cycle performance also allowed us to optimize performance without fear of breaking functionality. For example, at a late stage, we discovered an unnecessary bubble in our pipeline between some pairs of instructions. In our flow, the fix was simple and localized to a faulty bypass rule; because we knew that the compiler would preserve the operational correctness as described in the rules, we needed to re-verify only the performance aspect. Had we used a design methodology where correctness and performance cannot be easily separated, we might well have judged that the risk of breaking existing functionality was not worth the benefit of improved performance, and taped out without fixing the bug.

7.2. SYNTHESIS AND THE BACKEND

We used Synopsys Design Compiler (DC) to synthesize the RTL Verilog produced by the Bluespec compiler backend, using an ARM sc12 low voltage threshold cell library targeting an IBM 45nm SOI process. Synthesis targeted a clock frequency of 800MHz, and leveraged DC's automatic clock-gating feature.

Synthesized netlists were placed and routed using Cadence's Encounter and Virtuoso

tools, which were also used to lay out the power grid and generate the clock tree (see Figure 7-1). Synopsys PrimeTime was used for static timing analysis to enable post-layout timing closure. Layout-versus-schematic verification and signoff design rule checks were performed using Mentor Calibre.

An ARM 1.8V wirebond I/O library was used for power/ground and chip I/O connections, with a single row of pads surrounding the chip. An IBM PLL was used to generate the clock from an external signal.

Both synthesis and physical layout were performed bottom-up. First, three tile variants were synthesized and laid out: two for the tiles that included external memory controller shims and one variant for the remaining 108 tiles. Netlist simulations and timing analysis were then performed in single-tile, four-tile, and nine-tile configurations. The post-layout tiles were treated as black boxes for full-chip synthesis and layout (see Figure 7-2).

7.3. SYSTEM VERIFICATION

With evolving VLSI technology and increasing design complexity, verification costs have become more critical than ever. Increasing core counts are only making the problem worse because any pairwise interactions among cores result in a combinatorial explosion of the state space as the number of cores grows. Distributed cache coherence protocols in particular are well known to be notoriously complex and difficult to design and verify. The response to a given request is determined by the state of *all* actors in the system (for example, when one cache requests write access to a cache line, any cache containing that line must be sent an invalidate message); moreover, the indirections involved and the nondeterminism inherent in the relative timing of events requires a coherence protocol implementation to introduce many transient states that are not explicit in the higher-level protocol. This causes the number of actual states in even relatively simple protocols (e.g., MSI, MESI) to explode combinatorially (Arvind et al., 2008), and results in complex cooperating state machines driving each cache and directory (Lenoski and Weber, 1995). In fact, one of the main sources of bugs in such protocols is reachable transient states that are missing from the protocol definition, and fixing them often requires non-trivial modifications to the high-level specification. To make things worse, many transient states make it difficult to write well-defined testbench suites: with multiple threads running in parallel on multicores, writing high-level applications that exercise all the reachable low-level transient states—or even enumerating those states—is not an easy task. Indeed, descriptions of more optimized protocols can be so complex that they take experts months to understand, and bugs can result from specification ambiguities as well as implementation errors (Joshi et al., 2003). Significant modeling simplifications must be made to make exploring the state space tractable (Abts et al., 2003), and even formally

verifying a given protocol on a few cores gives no confidence that it will work on 100.

While design and verification complexity is difficult to quantify and compare, both the remote-access-only baseline and the full EM² system we implemented have a significant advantage over directory cache coherence: a given memory address may only be cached in a single place. This means that any request—remote or local—will depend only on the validity of a given line in a single cache, and no indirections or transient states are required. The VALID and DIRTY flags that together determine the state of a given cache line are local to the tile and cannot be affected by state changes in other cores. The thread migration framework does not introduce additional complications, since the data cache does not care whether a local memory request comes from a native thread or a migrated thread: the same local data cache access interface is used. The overall correctness can therefore be cleanly separated into (a) the remote access framework, (b) the thread migration framework, (c) the cache that serves the memory request, and (d) the underlying on-chip interconnect, all of which can be reasoned about separately. This modularity makes the EM² protocols easy to understand and reason about, and enabled us to safely implement and verify modules in isolation and integrate them afterwards without triggering bugs at the module or protocol levels (see verification steps I and II in Figure 7-3).

The homogeneous tiled architecture we chose for EM² allowed us to significantly reduce verification time by first integrating the individual tiles in a 4-tile system. This resulted in far shorter simulation times than would have been possible with the 110 cores, and allowed us to run many more test programs. At the same time, the 4-tile arrangement exercised all of the inter-tile interfaces, and we found no additional bugs when we switched to verifying the full 110-core system. Unlike directory entries in directory-based coherence designs, EM² cores never store information about more than the local core, and all of the logic required for the migration framework—the decision whether to migrate or execute a remote cache access, the calculation of the destination core, serialization and deserialization of network packets from/to the execution context, evicting a running thread if necessary, etc.—is local to the tile. As a result, it was possible to exercise the entire state space in the 4-tile system; perhaps more significantly, however, this also means that the system could be scaled to an arbitrary number of cores without incurring an additional verification burden.

7.4. SYSTEM CONFIGURATION AND BOOTSTRAP

To initialize the EM² chip to a known state at during power-up, we chose to use a scan-chain mechanism. Unlike the commonly employed bootloader strategy, in which one of the cores is hard-coded with a location of a program that configures the rest of the system, successful configuration via the scan-chain approach does not rely on any cores to be operating cor-

rectly: the only points that must be verified are (a) that bits correctly advance through the scan chain, and (b) that the contents of the scan chain are correctly picked up by the relevant core configuration settings. In fact, other than a small state machine to ensure that caches are invalidated at reset, the EM² chip does not have any reset-specific logic that would have to be separately verified.

The main disadvantages here are (a) that the EM² chip is not self-initializing, i.e., that system configuration must be managed external to the chip, and (b) that configuration at the slow rate permitted by the scan chain will take a number of minutes. For an academic chip destined to be used exclusively in a lab environment, however, those disadvantages are relatively minor and worth offloading complexity from the chip itself onto test equipment.

The scan chain itself was designed specifically to avoid hold-time violations in the physical design phase. To this end, the chain uses two sets of registers and is driven by two clocks: the first clock copies the current value of the scan input (i.e., the previous link in the chain) into a "lockup" register, while the second moves the lockup register value to a "config" register, which can be read by the core logic (see Figure 7-4). By suitably interleaving the two scan clocks, we ensure that the source of any signal is the output of a flip-flop that is not being written at the same clock edge, thus avoiding hold-time issues. While this approach sacrificed some area (since the scan registers are duplicated), it removed a significant source of hold-time violations during the full-chip assembly phase of physical layout, likely saving us time and frustration.

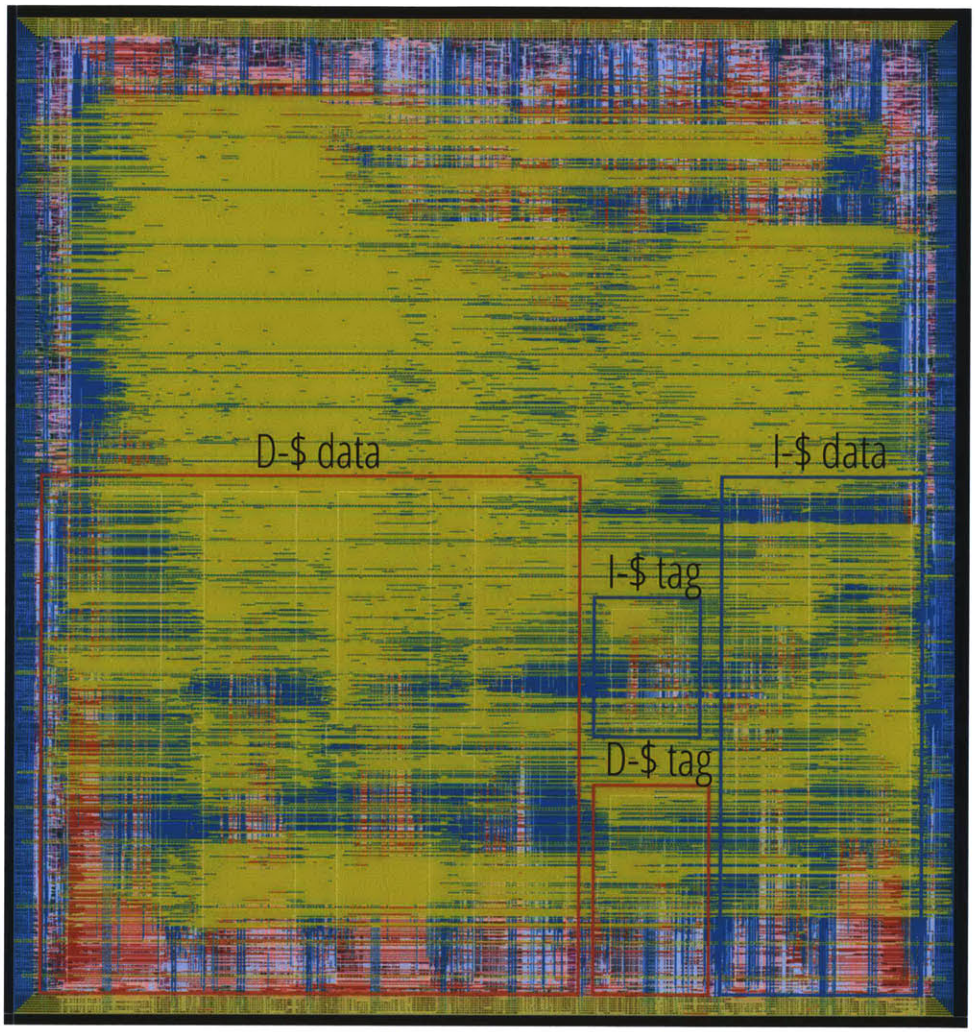


Figure 7-1: The layout of a single tile of the EM² chip. Most of the tile is taken up by the cache SRAMs (outlined), and the rest by the core datapath, cache controllers, and on-chip network routers. This tile was treated as a black box and replicated to obtain the full 110-tile system.

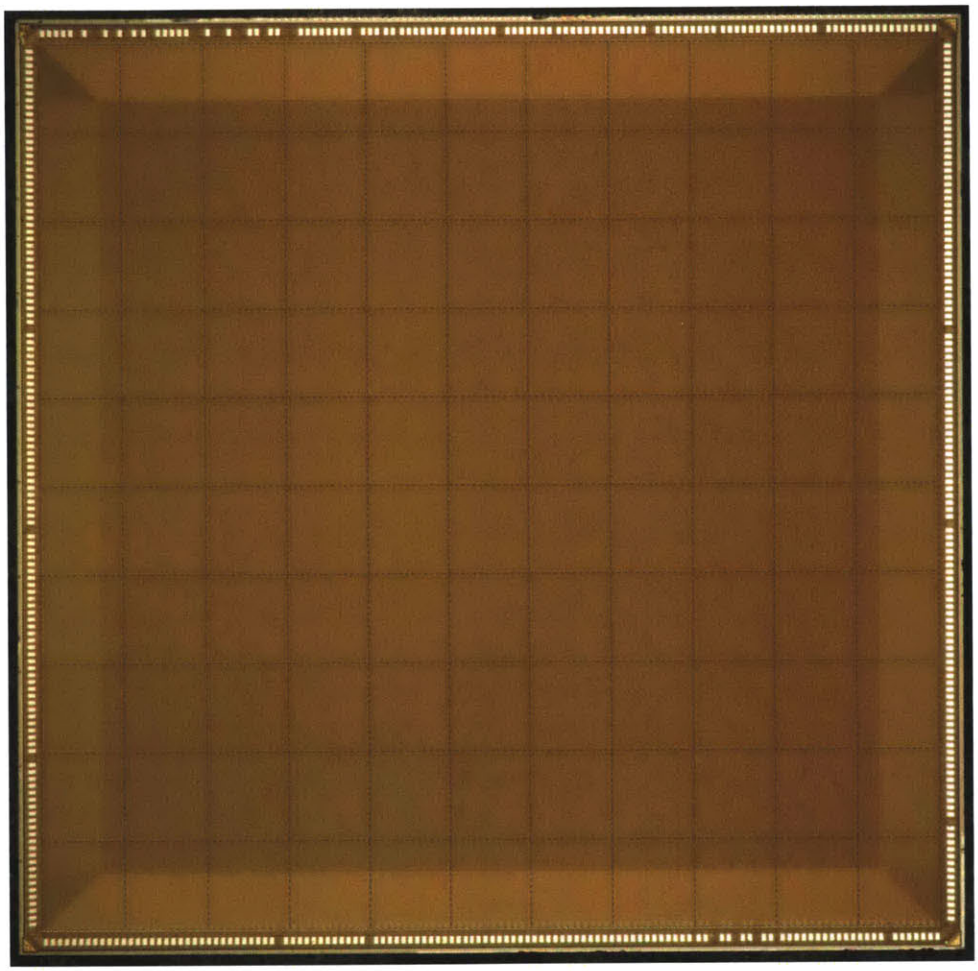


Figure 7-2: A microphotograph of the fabricated EM² chip, with the outlines of the 110-tiles superimposed.

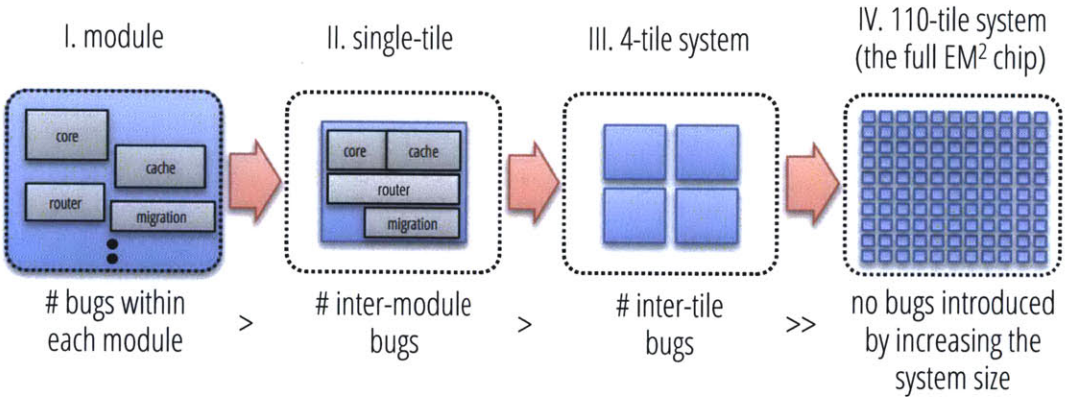


Figure 7-3: Bottom-up verification methodology of EM². The high modularity and design simplicity of EM² enabled verification to scale as we integrated modules and increase the system size: the number of bugs found decreased at each step, and no new bugs were discovered by moving from a 4-tile model to the full 110-tile system.

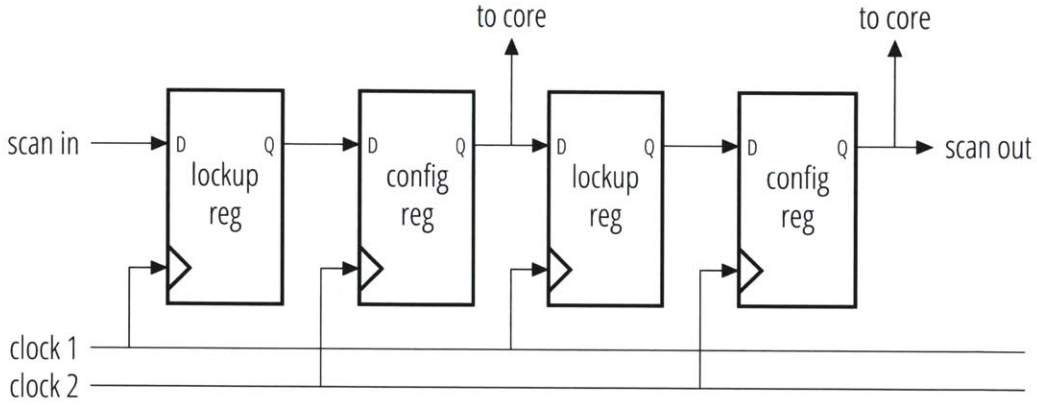


Figure 7-4: The two-stage scan chain used to configure the EM² chip. The two separate scan clocks and two sets of registers prevent hold time violations due to short paths between the scan chain registers.

CHAPTER 8

PERFORMANCE EVALUATION

In this chapter, we evaluate the fine-grained hardware-level thread migration techniques we have implemented in silicon, and examine how they compare to a remote-access-only distributed cache architecture and to directory-based cache coherence.

Because the packaged chips were in the bringup phase at the time of this writing and it was not yet possible to obtain performance and area measurements directly from the silicon, our evaluation here is based on RTL simulation and synthesis estimates.

8.1. METHODS

To evaluate the EM^2 implementation, we chose an idealized cache-coherent baseline architecture with a two-level cache hierarchy (a private L1 data cache and a shared L2 cache). In this scheme, the L2 is distributed evenly among the 110 tiles and the size of each L2 slice is 512KB. An L1 miss results in a cache line being fetched from the L2 slice that corresponds to the requested address (which may be on the same tile as the L1 cache or on a different tile). While this cache fetch request must still traverse the network to the correct L2 slice and bring the cache line back, our cache-coherent baseline is idealized in the sense that rather than focusing on the details of a specific coherence protocol implementation, it does not include a directory and never generates any coherence traffic (such as invalidates and acknowledgements); coherence among caches is ensured "magically" by the simulation infrastructure. While such an idealized implementation is impossible to implement in hardware, it represents an upper bound on the performance of any implementable directory coherence protocol, and serves as the ultimate baseline for performance comparisons.

RTL simulation

To obtain the on-chip traffic levels and completion times for our architecture, we began with the post-tapeout RTL of the EM² chip, removed such ASIC-specific features as scan chains and modules used to collect various statistics at runtime, and added the same shared-L2 cache hierarchy as the cache-coherent baseline. Since our focus is on comparing on-chip performance, the working set for our benchmarks is sized to fit in the entire shared-L2 aggregate capacity. All of the simulations used the entire 110-core chip RTL; for each benchmark, we report the completion times as well as the total amount of on-chip network traffic (i.e., the number of times any flit traveled across any router crossbar).

The ideal CC simulations only run one thread in each core, and therefore only use the native context. Although the EM² simulations can use the *storage space* of both contexts in a given core, this does not increase the parallelism available to EM²: because the two contexts share the same I$ port, only one context can be executing an instruction at any given time.

Both simulations use the same 8KB L1 instruction cache as the EM² chip. Unlike the PC, instruction cache entries are not migrated as part of the thread context; while this might at first appear to be a disadvantage when a thread first migrates to a new core, we have observed that in practice at steady state the I$ has usually already been filled (either by other threads or by previous iterations that execute the same instruction sequence), and the I$ hit rate remains high.

Area and power estimates

Area and power estimates were obtained by synthesizing RTL using Synopsys Design Compiler (DC). For the EM² version, we used the post-tapeout RTL with the scan-chains and statistics modules deleted; we reused the same IBM 45nm SOI process with the ARM sc12 low-power ASIC cell library and SRAM blocks generated by IBM Memory Compiler. Synthesis targeted a clock frequency of 800MHz, and leveraged DC's automatic clock-gating feature.

To give an idea of how these costs compare against that of a well-understood, realistic architecture, we also estimated the area and leakage power of an equivalent design where the data caches are kept coherent via a directory-based MESI protocol (CC). We chose an exact sharer representation (one bit for each of the 110 sharers) and either the same number of entries as in the data cache (CC 100%) or half the entries (CC 50%);[1] in both versions the directory was 4-way set-associative. To estimate the area and leakage power of the directory,

[1]Note that because multiple data caches can "gang up" on the same directory slice, the 100% version does not guarantee freedom from directory-capacity invalidations.

we synthesized a 4-way version of the data cache controller from EM² chip with SRAMs sized for each directory configuration, using the same synthesis constraints (since a directory controller is somewhat more complex than a cache controller, this approach likely results in a slight underestimate).

For area and leakage power, we report the synthesis estimates computed by DC, i.e., the total cell area in μm^2 and the total leakage power. While all of these quantities typically change somewhat post-layout (because of factors like routing congestion or buffers inserted to avoid hold-time violations), we believe that synthesis results are sufficient to make architectural comparisons.

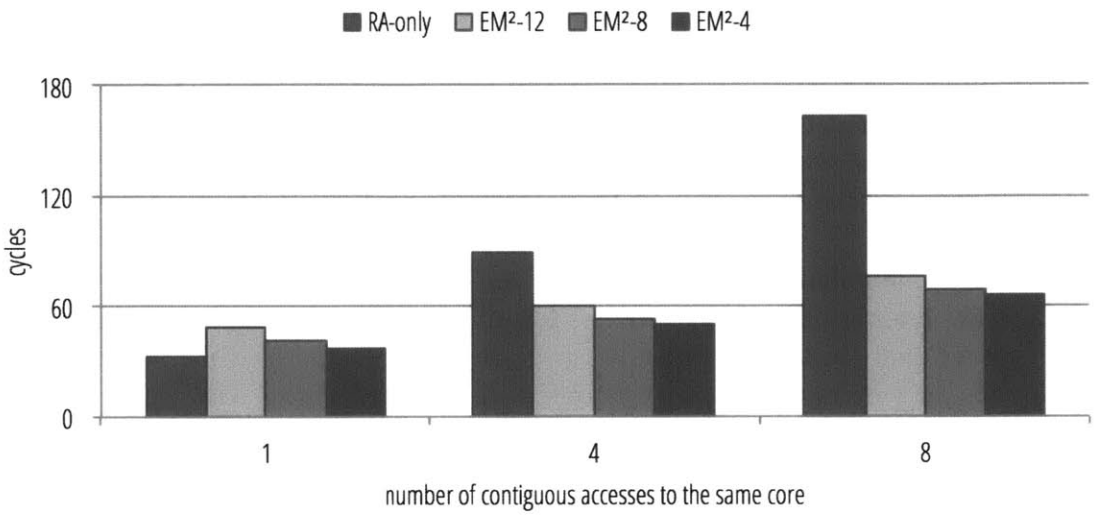
Dynamic power dominates the power signature, but is highly dependent on the specific benchmark, and obtaining accurate estimates for all of our benchmark is not practical. Instead, we observe that for the purposes of comparing EM² to the baseline architecture, it suffices to focus on the differences, which consist of (a) the additional core context, (b) the migration predictor, and (c) differences in cache and network accesses. The first two are insignificant: our implementation allowed only one of the EM² core contexts to be active in any given cycle, so even though the extra contexts adds leakage, dynamic power remains constant. The migration predictor is a small part of the tile and does not add much dynamic power (for reference, DC estimated that the predictor accounts for < 4.5% of the tile's dynamic power). Since we ran the same programs and pre-initialized caches, the cache accesses were the same, meaning equal contribution to dynamic power. The only significant difference is in the dynamic network power, which is directly proportional to the on-chip network traffic (i.e., the number of network flits sent times the distance traveled by each flit); we therefore report this for all benchmarks as a proxy for dynamic power.

8.2. EVALUATION

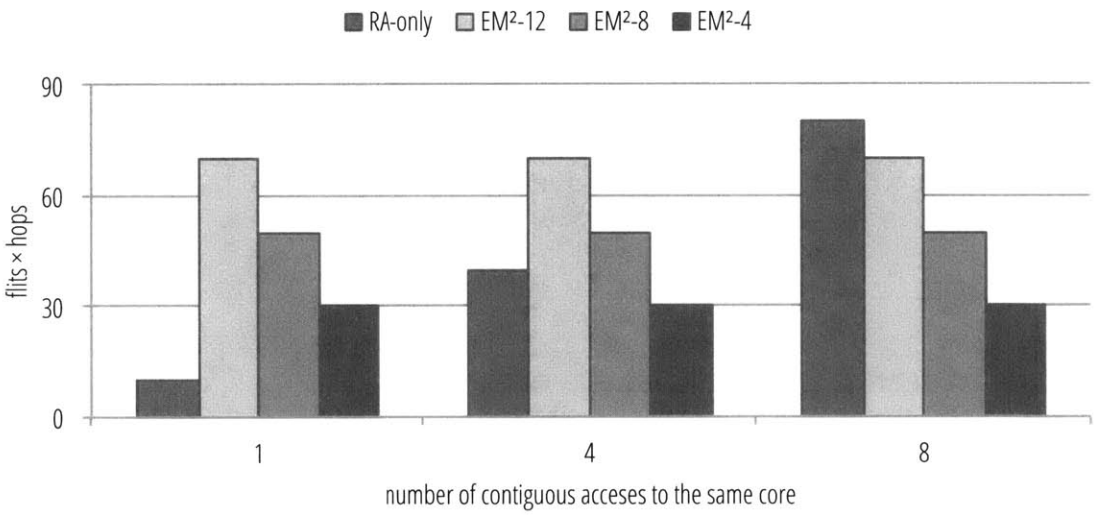
Performance tradeoff factors

To precisely understand the conditions under which fast thread migration results in improved performance, we created a simple parameterized benchmark that executes a sequence of loads to memory assigned to a remote L2 slice. There are two parameters: the *run length* is the number of contiguous accesses made to the given address range, and *cache misses* is the number of L1 misses these accesses induce (in other words, this determines the stride of the access sequence); we also varied the on-chip distance between the tile where the thread originates and the tile whose L2 caches the requested addresses.

Figure 8-1 shows how a program that only makes remote cache accesses (RA-only) compares with a program that migrates to the destination core 4 hops away, makes the memory



(a) completion time



(b) network traffic

Figure 8-1: Performance and network traffic comparison of EM² vs. RA: EM² performs better as the run length of contiguous accesses to the same core grows (synthetic benchmark).

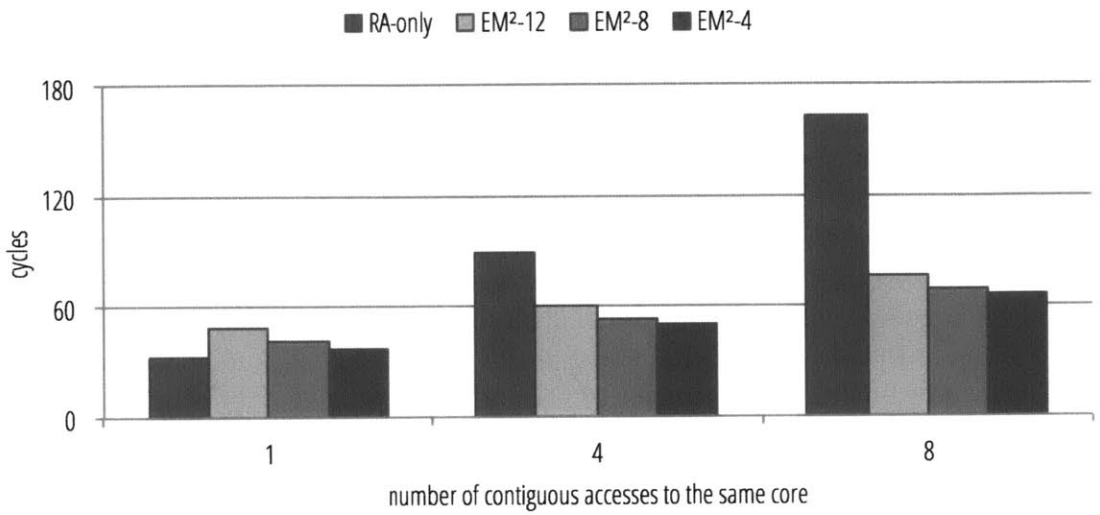
accesses, and returns to the core where it originated (EM²), where the migrated context size is 4, 8, and 12 stack entries (EM²-4, EM²-8, and EM²-12). Since the same L1 cache is always accessed—locally or remotely—both versions result in exactly the same L1 cache misses, and the only relevant parameter is the run length. For a singleton access (*run length* = 1), RA is slightly faster than any of the migration variants because the two migration packets involved are longer than the RA request/response pair, and, for the same reason, induce much more network traffic. For multiple accesses, however, the single migration round-trip followed by local cache accesses performs better than the multiple remote cache access round trips, and the advantage of the migration-based solution grows as the run length increases.

The tradeoff against our "ideal cache coherence" private-cache baseline (CC) is less straightforward than against RA: while CC will still make a separate request to load every cache *line*, subsequent accesses to the same cache line will result in L1 cache hits and no network traffic. Figure 8-2 illustrates how the performance of CC and EM² depends on how many times the same cache line is reused in 8 accesses. When all 8 accesses are to the same cache line (*cache misses* = 1), CC requires one round-trip to fetch the entire cache line, and is slightly faster than EM², which needs to unload the thread context, transfer it, and load it in the destination core. Once the number of misses grows, however, the multiple round-trips required in CC become more costly than the context load/unload penalty of the one round-trip migration, and EM² performs better. And in all cases, EM² can induce less on-chip network traffic: even in the one-miss case where CC is faster, the thread context that EM² has to migrate is often smaller than the CC request and the cache line that is fetched.

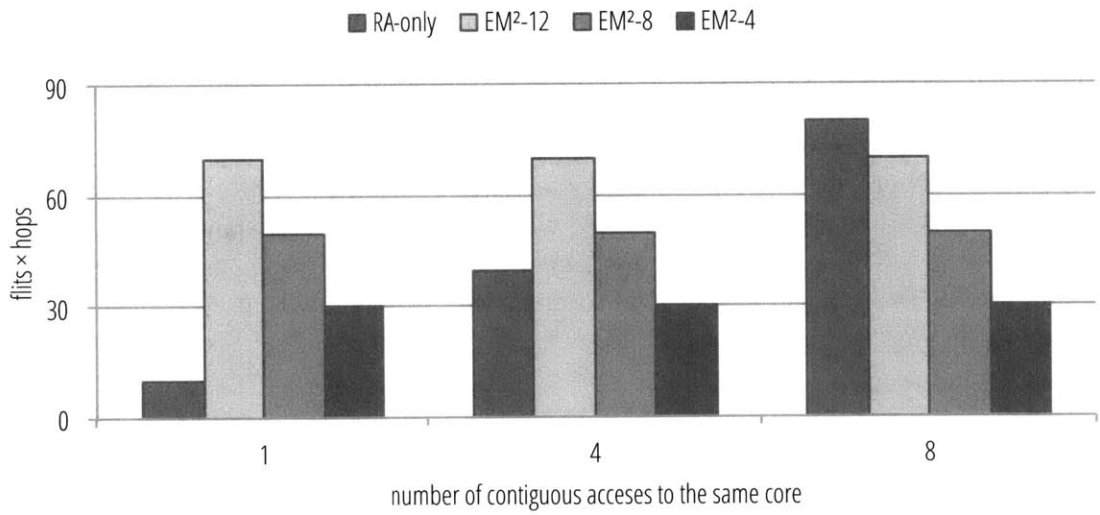
Finally, Figure 8-3 examines how the three schemes are affected by the on-chip distance between the core where the thread originates and the core that caches the requested data (with *run length* = 8 and *cache misses* = 2). RA, which requires a round-trip access for every word, grows the fastest (i.e., eight round-trips), while CC, which only needs a round-trip cache line fetch for every L1 miss (i.e., two round-trips), grows much more slowly. Because EM² only requires one round-trip for all accesses, the distance traveled is not a significant factor in performance.

Benchmark performance

Figures 8-4 and 8-5 show how the performance of EM² compares to the ideal CC baseline for several benchmarks. These include: (1) single-threaded *memcpy* in next-neighbor (*near*) and cross-chip (*far*) variants, (2) parallel *k*-fold cross-validation (*par-cv*), a machine learning technique that uses stochastic gradient learning to improve model accuracy, (3) 2D Jacobi iteration (*jacobi*), a widely used algorithm to solve partial differential equations, and (4) partial table scan (*tbscan*), which executes queries that scan through a part of a globally shared data



(a) completion time



(b) network traffic

Figure 8-2: Performance and network traffic comparison of EM² vs. CC-ideal: EM² performs better as the rate of cache misses increases (synthetic benchmark).

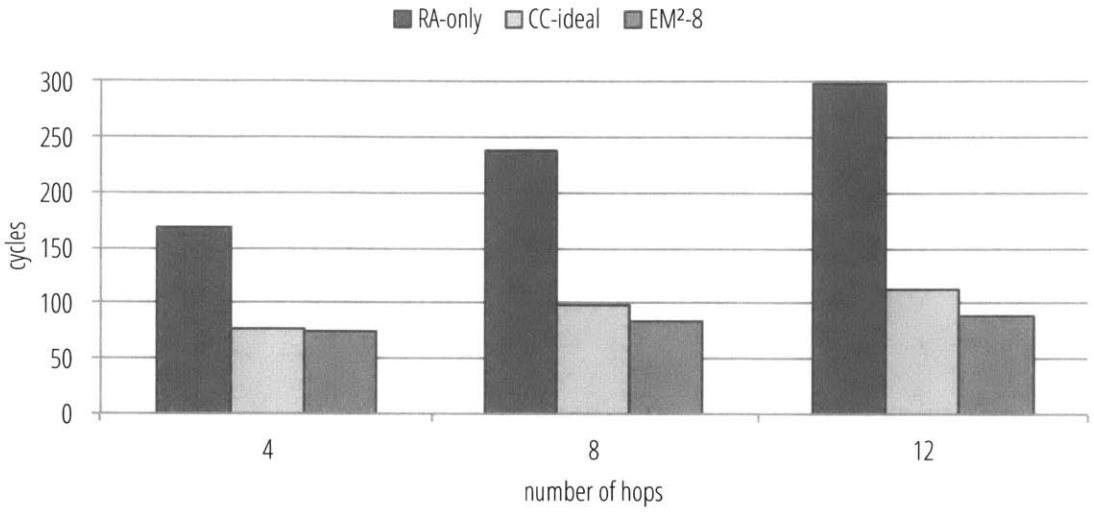


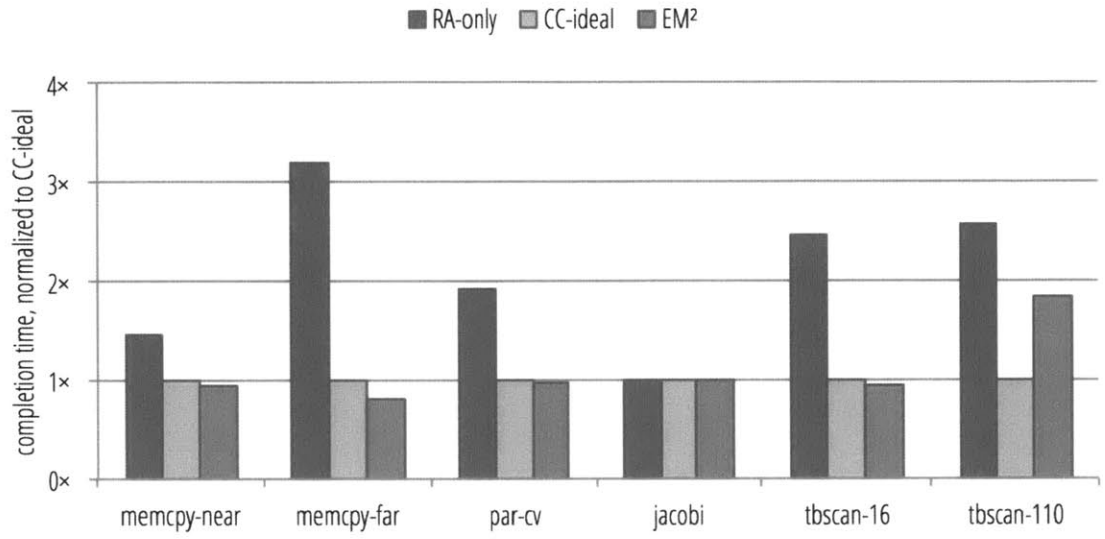
Figure 8-3: The effect of intercore distance on completion time. EM² outperforms RA and CC-ideal as the number of cores grows and the average on-chip distance increases.

table distributed among the cache shards. We first note some overall trends and then discuss each benchmark in detail below.

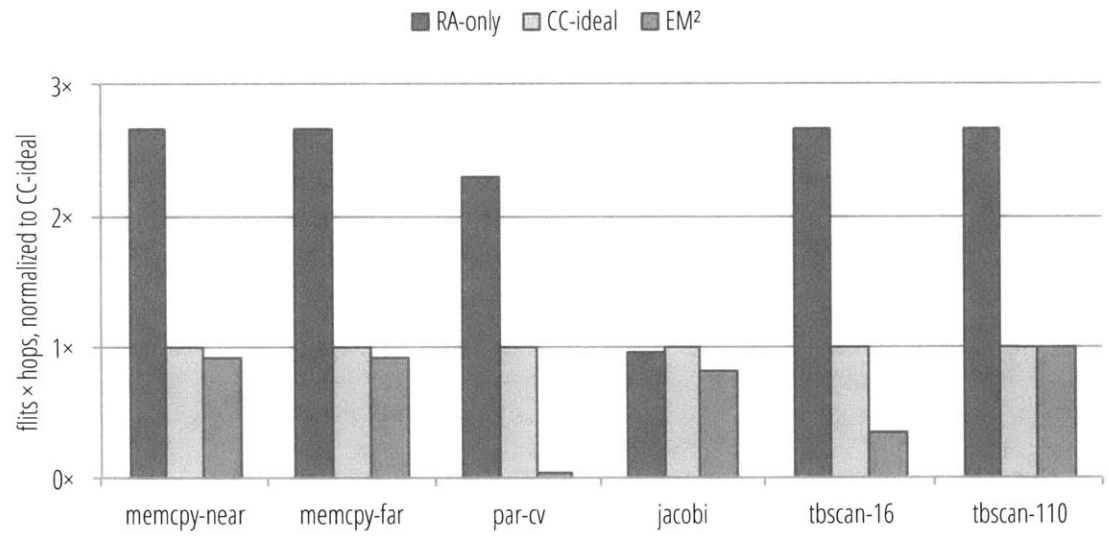
Overall remarks. First, Figure 8-4a illustrates the overall performance (i.e., completion time) and on-chip network traffic of the ideal directory-based baseline (CC), the remote-access-only variant (RA), and the EM² architecture. Overall, EM² always outperforms RA, offering up to 3.9× reduction in run time, and as well or better than CC in all cases except one. Throughout, EM² also offers significant reductions in on-chip network traffic, up to 42× less traffic than CC for *par-cv*.

Migration rates, shown in Figure 8-5a, range from 0.2 to 20.9 migrations per 1,000 instructions depending on the benchmark. These quantities justify our focus on *efficient* thread movement: if migrations occur at the rate of nearly one in every hundred to thousand instructions, taking 1000+ cycles to move a thread to a different core would indeed incur a prohibitive performance impact. Most migrations are caused by data accesses, with stack under/overflow migrations at a negligible level, and evictions significant only in the *tbscan* benchmarks.

Even with many threads, effective migration latencies are low (Figure 8-5b, bars), with the effect of distance clearly seen for the *near* and *far* variants of *memcpy*; the only exception here is *par-cv*, in which the migration latency is a direct consequence of delays due to inter-thread synchronization (as we explain below). At the same time, migration sizes (Figure 8-5b,

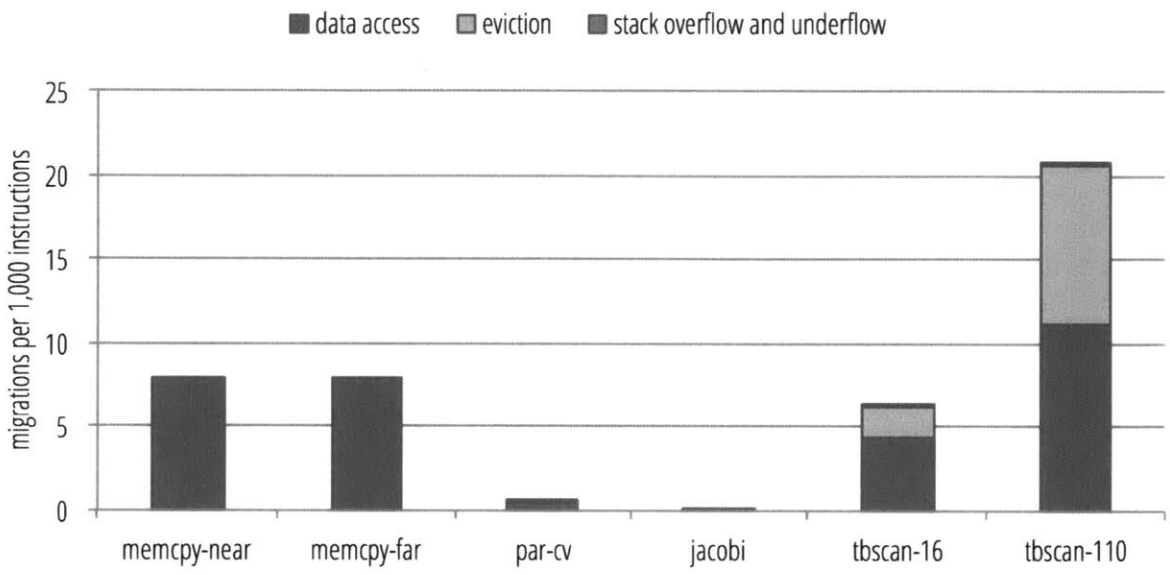


(a) performance normalized to CC-ideal

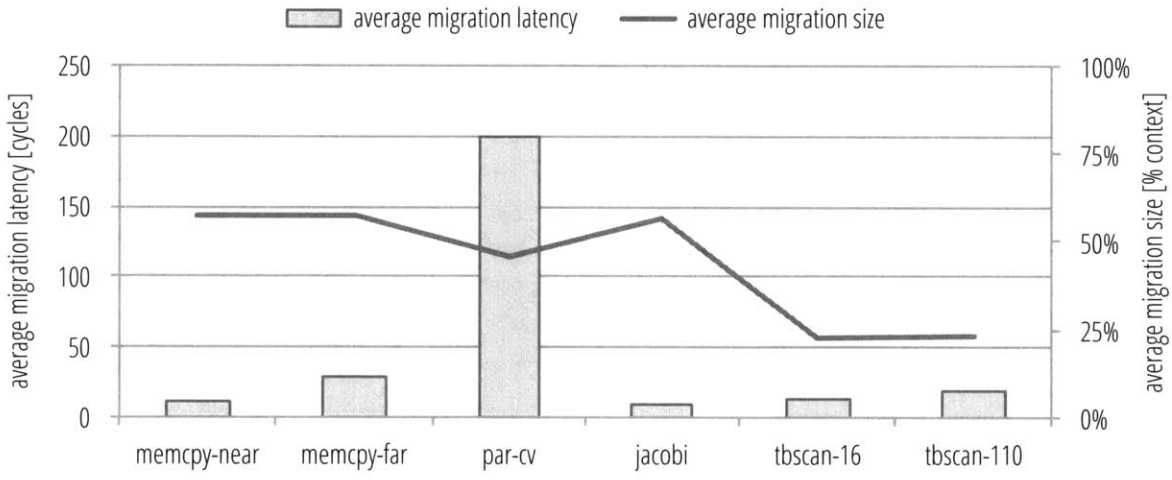


(b) network traffic normalized to CC-ideal

Figure 8-4: Performance and network traffic under EM², RA, and CC-ideal on various benchmarks, normalized to CC-ideal (RTL simulation).



(a) migrations per 1000 instructions



(b) thread migration performance

Figure 8-5: EM² migration rates and migration performance on various benchmarks (RTL simulation).

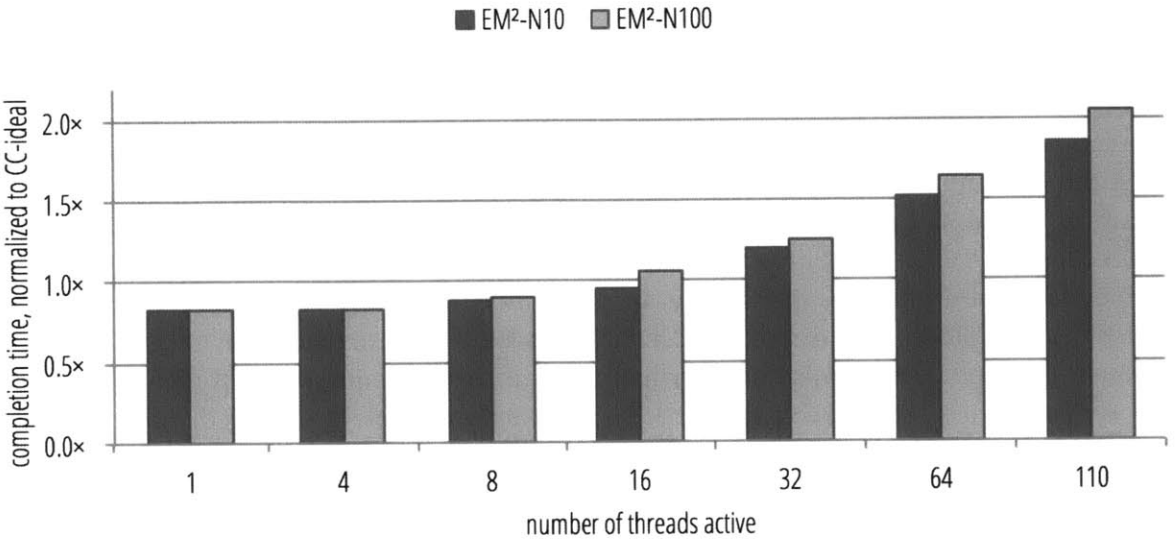
line) vary significantly, and stay well below the 60% mark (44% on average): since most of the on-chip traffic in the EM² case is due to migrations, forgoing partial-context migration support would have significantly increased the on-chip traffic (cf. Figure 8-4b).

Memory copy. The *memcpy-near* and *memcpy-far* benchmarks copy 32KB (the size of an L1 data cache) from a memory address range allocated to a next-neighbor tile (*memcpy-near*) or a tile at the maximum distance across the 110-core chip (*mempcy-far*). In both cases, EM² is able to repeatedly migrate to the source tile, load up a full thread context's worth of data, and migrate back to store the data at the destination addresses; because the maximum context size exceeds the cache line size that ideal CC fetches, EM² has to make fewer trips and performs better both in terms of completion time and network traffic. Distance is a significant factor in performance—the fewer round-trips of EM² make a bigger difference when the source and destination cores are far apart—but does not change the % improvement in network traffic, since that is determined by the the total amount of data transferred in EM² and CC.

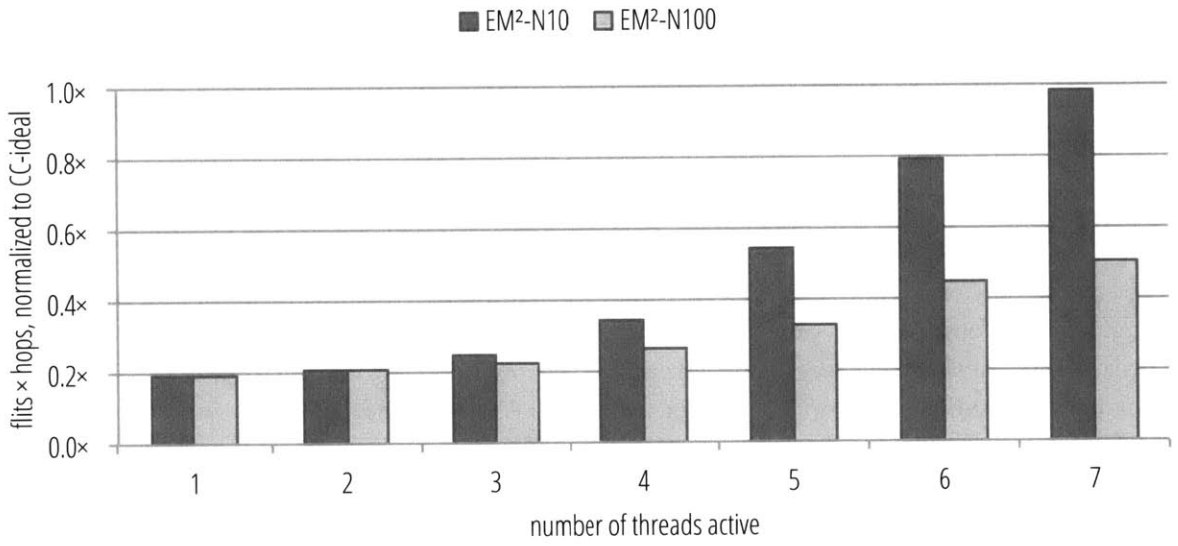
Partial table scan. In this benchmark, random SQL-like queries are assigned to separate threads, and the table that is searched is distributed in equal chunks among the per-tile L2 caches. We show two variants: a light-load version where only 16 threads are active at a time (*tbscan-16*) and a full-load version where all of the 110 available threads execute concurrently (*tbscan-110*); under light load, EM² finishes slightly faster than CC-ideal and significantly reduces network traffic (2.9×), while under full load EM² is 1.8× slower than CC-ideal and has the same level of network traffic.

Why such a large difference? Under light load, EM² takes full advantage of data locality, which allows it to significantly reduce on-chip network traffic, but performs only slightly better than CC-ideal because queries that access the same data chunks compete for access to the same core and effectively serialize some of the computation. Because the queries are random, this effect grows as the total number of threads increases (Figure 8-6), resulting in very high thread eviction rates under full load (Figure 8-5a); this introduces additional delays and network traffic as threads ping-pong between their home core and the core that caches the data they need.

This ping-pong effect, and the associated on-chip traffic, can be reduced by guaranteeing that each thread can perform N (configurable in hardware) memory accesses before being evicted from a guest context. Figure 8-6 illustrates how *tbscan* performs when N = 10 and N = 100: a longer guaranteed guest-context occupation time results in up to 2× reductions in network traffic at the cost of a small penalty in completion time due to the increased level of serialization. This highlights an effective tradeoff between performance and power: with more serialization, EM² can use far less dynamic power due to on-chip network traf-



(a) completion time



(b) network traffic

Figure 8-6: Performance and network traffic with different number of threads for *tbscan* under EM². Overheads grow significantly worse when the number of threads approaches the number of available cores.

fic (and because fewer cores are actively computing) if the application can tolerate lower performance.

Parallel *k*-fold cross validation. In the *k-fold cross-validation* technique common in machine learning, data samples are split into k disjoint chunks and used to run k independent leave-one-out experiments. For each experiment, k – 1 chunks constitute the training set and the remaining chunk is used for testing; the results are then averaged to estimate the final prediction accuracy of the algorithm being trained. Since the experiments are computationally independent, naturally map to multiple threads (indeed, for sequential machine learning algorithms, such as stochastic gradient descent, this is the *only* practical form of parallelization because the model used in each experiment is necessarily sequential). The chunks are typically spread across the shared cache shards, and each experiment repeatedly accesses a given chunk before moving on to the next one.

With overall completion time slightly better under EM² than under CC-ideal and much better than under RA-only, *par-cv* stands out for its 42× reduction in on-chip network traffic vs. CC-ideal (96× vs. RA). This is because the cost of every migration is amortized by a large amount of local cache accesses on the destination core (as the algorithm learns from the given data chunk), while CC-ideal continuously fetches more data to feed the computation.

Completion time for *par-cv*, however, is only slightly better because of the nearly 200-cycle average migration times at full 110-thread utilization (Figure 8-5b). This is because of a serialization effect similar to that in *tbscan*: a thread that has finished learning on a given chunk and migrates to proceed onto the next chunk must sometimes wait en route while the previous thread finishes processing that chunk. Unlike *tbscan*, however, where the contention results from random queries, the threads in *par-cv* process the chunks in order, and avoid the penalties of eviction. As a result, at the same full utilization rate of 110 threads, *par-cv* has a better completion time under EM² but *tbscan* performs better under CC. (At a lower utilization, the average migration latency of *par-cv* falls: e.g., at 50 threads it becomes 9 cycles, making the EM² version 11% faster than CC.)

2D Jacobi iteration. In its essence, the *jacobi* benchmark propagates a computation through a matrix, and so the communication it incurs is between the boundary of the 2D matrix region stored in the current core and its immediate neighbors stored in the adjacent cores. Since the data accesses are largely to a thread's own private region, intercore data transfers are a negligible factor in the overall completion time, and the runtime for all three architectures is approximately the same. Still, EM² is able to reduce the overall network traffic because it can amortize the costs of migrating by consecutively accessing many matrix elements in the boundary region, while CC-ideal has to access this data with several L2 fetches.

Area and power costs

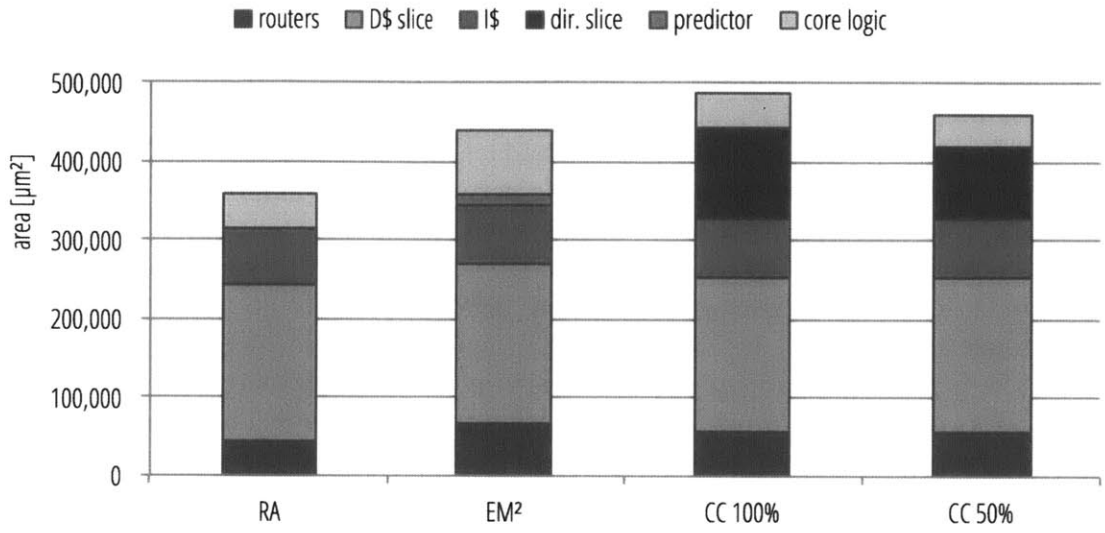
Since the CC-ideal baseline we use for the performance evaluation above has idealized directories, it does not make a good baseline for area and power comparison. Instead, we estimated the area required for MESI implementations with the directory sized to 100% and 50% of the total L1 data cache entries, and compared the area and leakage power to that of EM². The L2 cache hierarchy, which was added for more realistic performance evaluation and not a part of the actual chip, is not included here for both EM² and CC.

Table 8.1 summarizes the architectural components that differ. EM² requires an extra architectural context (for the guest thread) and on-chip networks for migrations and evictions as well as RA requests and responses. Our EM² implementation also includes a learning migration predictor; while this is not strictly necessary in a purely instruction-based migration design, it offers runtime performance advantages similar to those of a hardware branch predictor. In comparison, a deadlock-free implementation of MESI would replace the four migration and remote-access on-chip networks with three (for coherence requests, replies, and invalidations), implement D$ controller logic required to support the coherence protocol, and add the directory controller and associated SRAM storage.

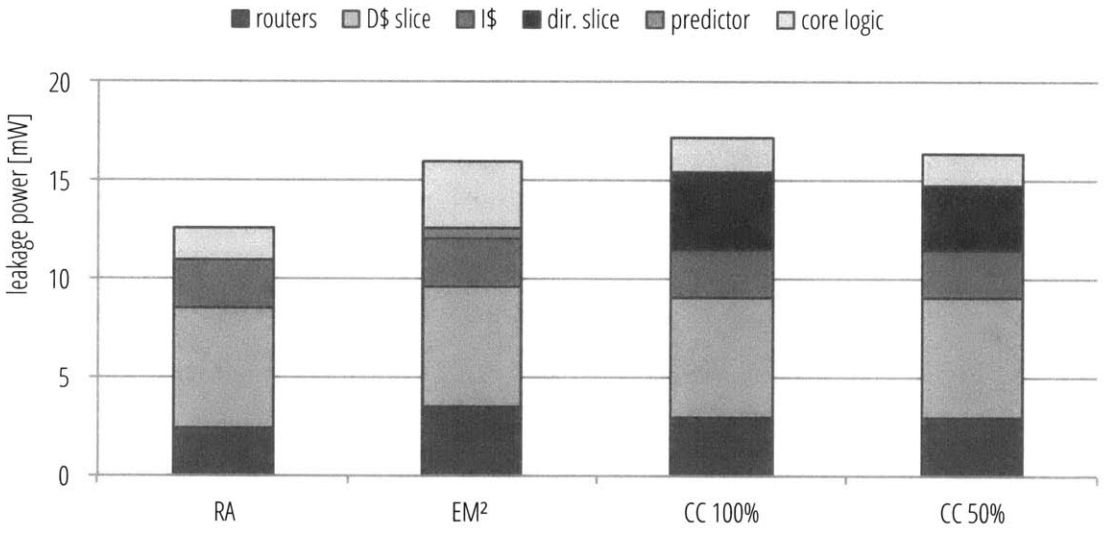
Figure 8-7 shows how the silicon area and leakage power compare. Not surprisingly, blocks with significant SRAM storage (the instruction and data caches, as well as the directory in the CC version) were responsible for most of the area in all variants. Overall, the extra thread context and extra router present in EM² were outweighed by the area required for the directory in both the 50% and 100% versions of MESI, which suggests that EM² may be an interesting option for area-limited CMPs.

	EM²	CC
extra execution context in the core	yes	no
migration predictor logic & storage	yes	no
remote cache access support in the D$	yes	no
coherence protocol logic in the D$	no	yes
coherence directory logic & storage	no	yes
number of independent on-chip networks	6	5

Table 8.1: A summary of architectural costs that differ in the EM² and CC implementations.



(a) silicon area



(b) leakage power

Figure 8-7: Relative area and leakage power costs of EM² vs. estimates for exact-sharer CC with the directory sized to 100% and 50% of the D$ entries (DC Ultra, IBM 45nm SOI hvt library, 800MHz).

CHAPTER 9

CONCLUSIONS

This dissertation has explored two main novel ideas: (a) fast, fine-grained thread migration implemented entirely in hardware, and (b) a sequentially consistent shared memory scheme based on thread migration. In this chapter, we draw conclusions from the design, implementation, and evaluation of the project, and outline some directions for future research.

9.1. THREAD MIGRATION

In this work, we have successfully demonstrated the feasibility of a hardware-only implementation of thread migration. With a deadlock-free migration protocol and partial-context migration, migrations take only a few cycles between adjacent cores and tens of cycles across a 110-core multiprocessor.

Results from a detailed cycle-accurate evaluation generally support the hypothesis that an efficient hardware-level implementation of thread migration is possible, and that it achieves migrations that are sufficiently fast and fine-grained to support even such extremely demanding applications as a shared-memory implementation.

The main weakness identified by our experiments was in our choice to make the migrations purely autonomous. As demonstrated by our oversubscribed tablescan benchmark, this opens up the opportunity for a birthday-paradox-like core congestion effect where multiple threads wish to execute on the same core. In this situation, each thread enters the core, runs for a few instructions, and is evicted because another thread is waiting; the original thread, however, still needs to run on the same core and simply gets back in the queue of threads waiting for it. This effect is mitigated somewhat by our ability to configure the number of memory accesses that must be completed before a thread may be evicted, but this requires careful orchestration on the part of the programmer. While it can be very effective (as demonstrated by the results on the parallel cross-validation benchmark), it is difficult to do for relatively

unstructured parallel workloads, and for workloads whose access pattern is tightly tied to the input data (such as the tablescan benchmark).

Future research can address this shortcoming in one of two ways. One is to implement a centralized scheduler that collects migration requests from various cores, schedules them to minimize congestion, and sends back grants to the threads that are allowed to proceed (in our shared-memory implementation, this would mean that threads whose requests were not granted must use remote cache access to execute non-local loads and stores). This would, however, significantly add to the latency experienced by average migrations, and would therefore render thread migration unsuitable for such sensitive applications as shared memory. A better option could be for cores to collect and transmit information about core and on-chip network congestion: while information from a far-away core would have to travel a number of cycles and as a result would be somewhat stale, core congestion is a phenomenon that becomes a problem only if it lasts for many cycles, and stale information would likely suffice to avoid or ameliorate this effect.

Another direction for future research is the evaluation of thread migration on a more complex (and realistic) core design that includes virtual memory and a full operating system. The limitations inherent in designing and implementing a large ASIC in academic conditions meant that this work was focused on a proof-of-concept pilot implementation in a bare-metal accelerator model. The present work, however, validates the feasibility and applicability of fine-grained thread migration, and paves the way for more complex cores and OS involvement.

9.2. SHARED MEMORY IMPLEMENTATION

As the other significant contribution of this work we have designed and implemented sequentially consistent shared memory based on thread migration, and demonstrated that it preforms competitively with an idealized state-of the art equivalent.

Although the migration-based shared memory scheme is simpler to implement than directory-based cache coherence, occupies comparable chip area, and offers equivalent performance, the address-based assignment of memory regions to per-core caches makes optimization of many classes of applications difficult. Potentially, a more complex processor with virtual memory support could provide an extra level of indirection and allow for automatic balancing of memory-to-cache mapping; nevertheless, a design working on page granularity would still be more difficult to optimize than a cache-coherence system working on cache-line granularity.

A more effective application of thread migration to shared memory might therefore be as a supplement to an already-existing cache coherence system, with the aim of, for example,

reducing cache pollution and L1-to-L2 distance. Although some studies have addressed using migration to accelerate performance in such a system, they have not considered the kind of fast, instruction-granularity migration mechanism we have presented here.

9.3. AN ENABLING TECHNOLOGY

Perhaps the most promising aspect of the fast, fine-grained thread migration mechanism we have presented here is that it is a general technique with many possible applications. Previous work has already considered several, ranging from performance to salvaging of partially functional cores and thermal load balancing, and we believe that our contribution of a proven migration mechanism with quantified tradeoffs opens the door to many more.

APPENDIX A

STACK ISA

A.1. ARCHITECTURAL STATE

The processor operates on 32-bit values, with signed arithmetic in two's complement. Memory is word-addressable, for a total of 16GB accessible address space. Internal storage is provided by two stacks, the *main stack* and the *auxiliary stack*; most instructions operate exclusively on the main stack, and several ferry data between the two stacks. Both stacks are transparently spilled to and refilled from data memory by the processor hardware, and so appear infinite to the programmer.

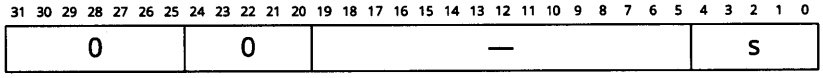
Memory instructions—loads, stores, and a load-reserve/store-conditional pair—access the data caches at word granularity. If the relevant address is not mapped to the local cache, the memory instruction involved may either execute via remote cache access or cause a migration to the core where the address can be cached. Stores come in two flavors: *acknowledged* and *unacknowledged*. The former increment an unacknowledged-stores counter upon issue and decrement it upon completion (which may take some time, especially if the instruction is executed via remote cache access), while the latter neither increment the counter nor generate an acknowledgement. For memory consistency reasons, any outstanding unacknowledged stores must be acknowledged before a thread may migrate, and a migrating instruction will stall until the unacknowledged-stores counter falls to 0. In addition, all memory instructions have *fenced* variants, which stall until all previous stores have been acknowledged before executing.

The instruction set, listed below, is sparsely encoded in 32-bit words, and detailed below.

A.2. INSTRUCTION ENCODING AND SEMANTICS

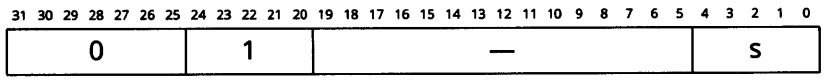
A complete list of instructions implemented in the EM² stack ISA follows. The descriptions use S[0] to mean the element at the top of the main stack, S[1] to mean the element just below the top of the main stack, and, in general, S[d] to mean the entry at depth d from the top of the main stack; similarly, A[0] indicates the element at the top of the auxiliary stack. Arithmetic, including effective address calculation, is signed unless noted otherwise.

sll



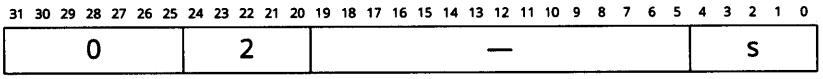
Shift left. Removes S[0] and pushes its value shifted left by s bits on top of the main stack.

srl



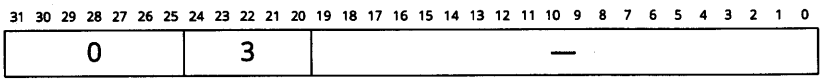
Shift right, logical. Removes S[0] and pushes its value shifted right by s bits on top of the main stack, zero-extended.

sra



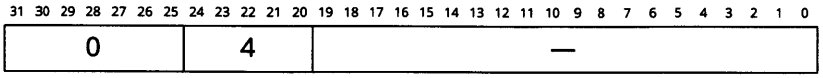
Shift right, arithmetic. Removes S[0] and pushes its value shifted right by s bits on top of the main stack, sign-extended.

lnot



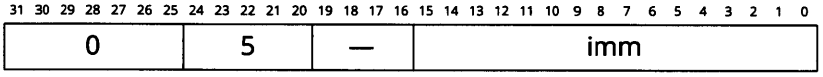
Logical negation. Removes S[0] and pushes its value negated on top of the main stack (i.e., 0 becomes 1 and any other value becomes 0).

bnot



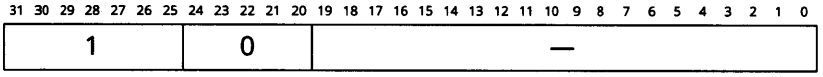
Bitwise negation. Removes S[0] and pushes its bitwise complement on top of the main stack.

sethi



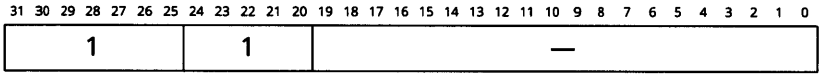
Set high half-word. Sets bits 31:16 of S[0] to the value of imm.

land



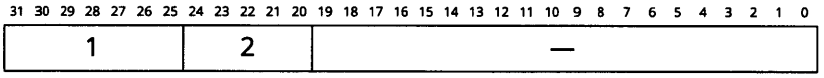
Removes S[0] and S[1], and pushes their logical conjunction on top of the main stack.

lor



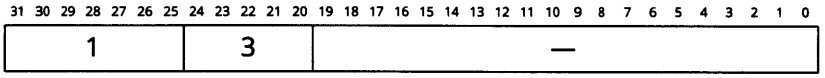
Removes S[0] and S[1], and pushes their logical disjunction on top of the main stack.

band



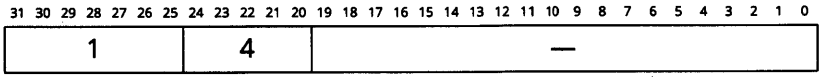
Removes S[0] and S[1], and pushes their bitwise conjunction on top of the main stack.

bor



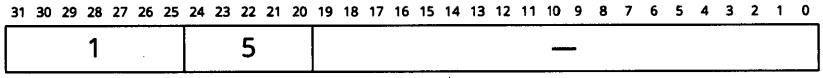
Removes S[0] and S[1], and pushes their bitwise disjunction on top of the main stack.

bxor



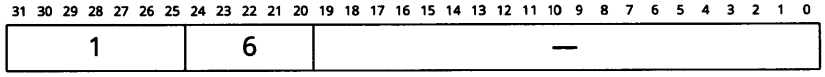
Removes S[0] and S[1], and pushes their bitwise exclusive disjunction on top of the main stack.

add



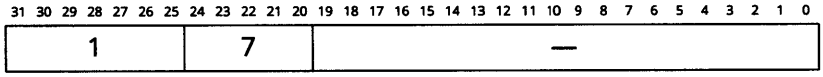
Removes S[0] and S[1], and pushes their arithmetic sum on top of the main stack.

sub



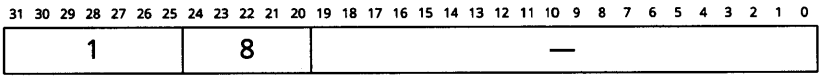
Removes S[0] and S[1], subtracts the second from the first, and pushes the result on top of the main stack.

cmp_eq



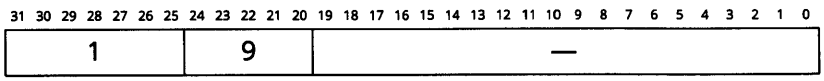
Removes S[0] and S[1] and performs a comparison; if their values are equal, pushes 1 on top of the main stack, otherwise pushes 0 on top of the main stack.

cmp_ne



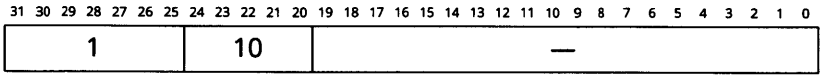
Removes S[0] and S[1] and performs a comparison; if their values are not equal, pushes 1 on top of the main stack, otherwise pushes 0 on top of the main stack.

cmp_ugt



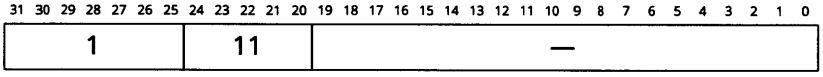
Removes S[0] and S[1] and performs an unsigned comparison; if the first is greater than the second, pushes 1 on top of the main stack, otherwise pushes 0 on top of the main stack.

cmp_uge



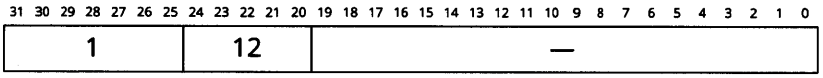
Removes S[0] and S[1] and performs an unsigned comparison; if the first is greater than or equal to the second, pushes 1 on top of the main stack, otherwise pushes 0 on top of the main stack.

cmp_ult



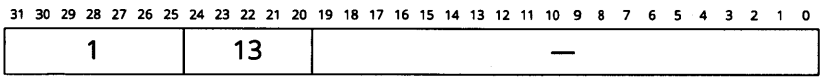
Removes S[0] and S[1] and performs an unsigned comparison; if the first is less than the second, pushes 1 on top of the main stack, otherwise pushes 0 on top of the main stack.

cmp_ule



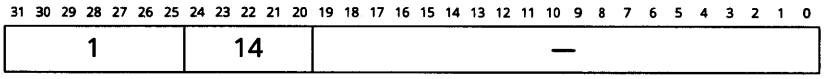
Removes S[0] and S[1] and performs an unsigned comparison; if the first is less than or equal to the second, pushes 1 on top of the main stack, otherwise pushes 0 on top of the main stack.

cmp_sgt



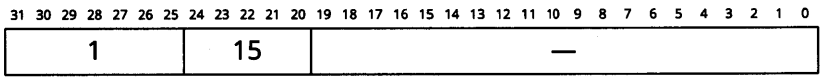
Removes S[0] and S[1] and performs a signed comparison; if the first is greater than the second, pushes 1 on top of the main stack, otherwise pushes 0 on top of the main stack.

cmp_sge



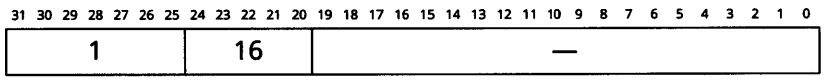
Removes S[0] and S[1] and performs a signed comparison; if the first is greater than or equal to the second, pushes 1 on top of the main stack, otherwise pushes 0 on top of the main stack.

cmp_slt



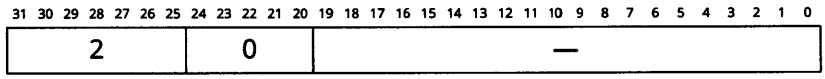
Removes S[0] and S[1] and performs a signed comparison; if the first is less than the second, pushes 1 on top of the main stack, otherwise pushes 0 on top of the main stack.

cmp_sle



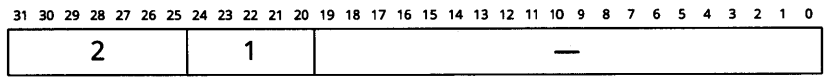
Removes S[0] and S[1] and performs a signed comparison; if the first is less than or equal to the second, pushes 1 on top of the main stack, otherwise pushes 0 on top of the main stack.

multu



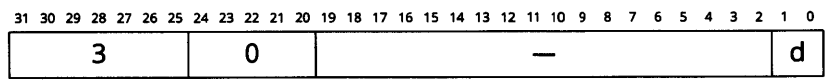
Removes S[0] and S[1] and pushes their unsigned product on top of the main stack.

mult



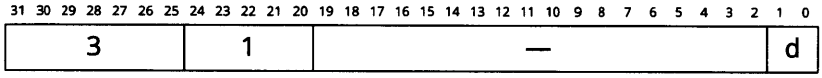
Removes S[0] and S[1] and pushes their signed product on top of the main stack.

pull



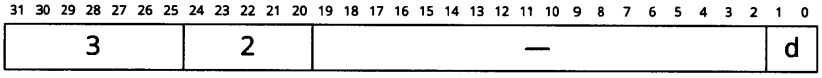
Removes S[d] and pushes it on top of the main stack.

pull_cp



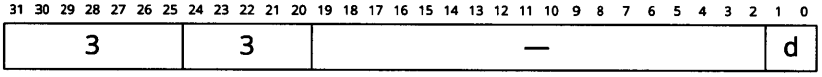
Pushes a copy of S[d] on top of the main stack.

tuck



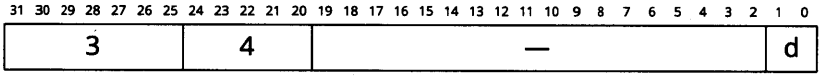
Removes S[0] and inserts it in position S[d].

tuck_cp



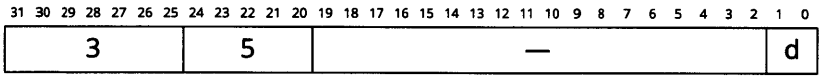
Inserts a copy of S[0] in position S[d].

swap



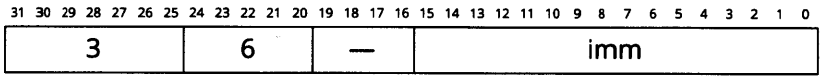
Swaps S[0] and S[d].

drop



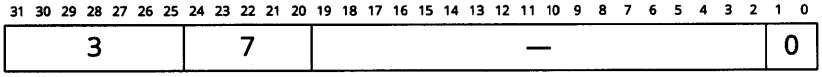
Removes S[d].

push



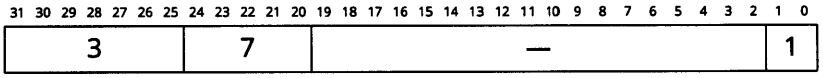
Pushes imm on top of the main stack.

push_pc



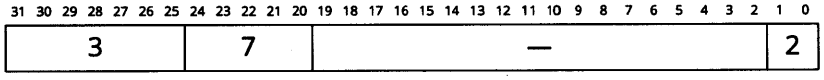
Pushes the address of the next instruction on top of the main stack.

push_core



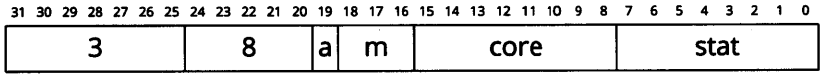
Pushes the ID of the core where the instruction is being executed on top of the main stack.

push_thread



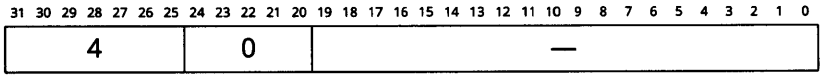
Pushes the ID of the thread in which the instruction is being executed on top of the main stack.

push_stat



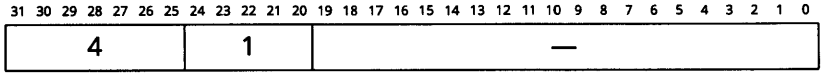
Reads statistics register stat in core with ID core and pushes it on top of the main stack. If this instruction causes the thread to migrate, the migrated context includes the top 2×m main stack entries and 2×a auxiliary stack entries.

main2aux_cp



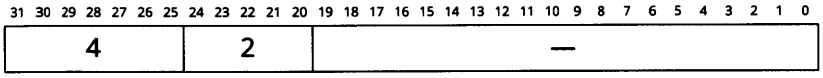
Pushes a copy of S[0] on top of the auxiliary stack.

main2aux



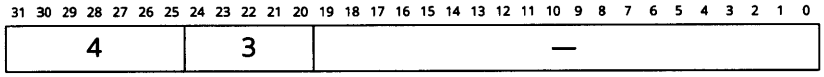
Removes S[0] and pushes it on top of the auxiliary stack.

aux2main_cp



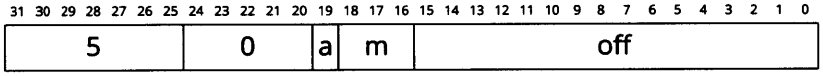
Pushes a copy of A[0] on top of the main stack.

aux2main



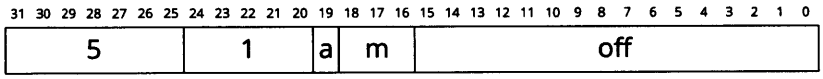
Removes A[0] and pushes it on top of the main stack.

ld



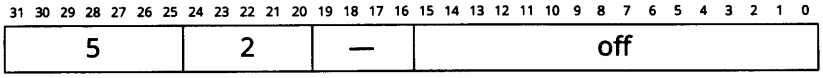
Removes S[0] and adds it to off; retrieves the word at the resulting address and pushes it on top of the main stack. By default uses the migration predictor for both the migration decision and the partial context sizes; if the prediction of the latter is disabled and this instruction causes the thread to migrate, the migrated context includes the top 2×m main stack entries and 2×a auxiliary stack entries.

ld_em



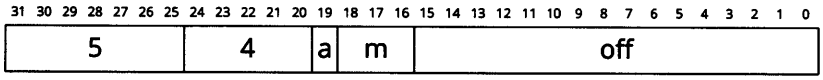
Removes S[0] and adds it to off; retrieves the word at the resulting address and pushes it on top of the main stack. Always migrates if the address is not cacheable in the local core; in this case, the migrated context includes the top 2×m main stack entries and 2×a auxiliary stack entries.

ld_ra



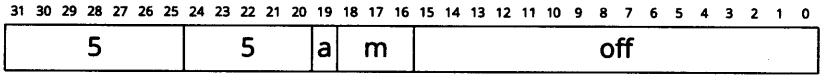
Removes S[0] and adds it to off; retrieves the word at the resulting address and pushes it on top of the main stack. Always executed via remote cache access if the address is not cacheable in the local core.

ld_rsv



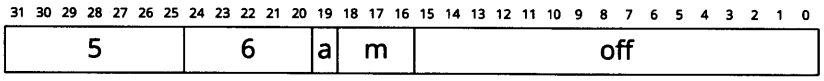
Atomic load-reserve. Removes S[0] and adds it to off; retrieves the word at the resulting address and pushes it on top of the main stack, simultaneously writing the address into the reservation register at the home core of the address. By default uses the migration predictor for both the migration decision and the partial context sizes; if the prediction of the latter is disabled and this instruction causes the thread to migrate, the migrated context includes the top 2×m main stack entries and 2×a auxiliary stack entries.

ld_rsv_em



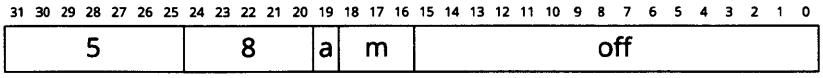
Atomic load-reserve. Removes S[0] and adds it to off; retrieves the word at the resulting address and pushes it on top of the main stack, simultaneously writing the address into the reservation register at the home core of the address. Always migrates if the address is not cacheable in the local core; in this case, the migrated context includes the top 2×m main stack entries and 2×a auxiliary stack entries.

ld_rsv_ra



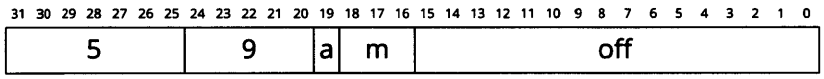
Atomic load-reserve. Removes S[0] and adds it to off; retrieves the word at the resulting address and pushes it on top of the main stack, simultaneously writing the address into the reservation register at the home core of the address. Always executed via remote cache access if the address is not cacheable in the local core.

fnc_ld



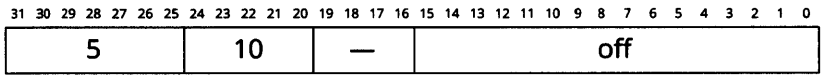
First, stalls execution until acknowledgements for all outstanding stores have been received. Next, removes S[0] and adds it to off; retrieves the word at the resulting address and pushes it on top of the main stack. By default uses the migration predictor for both the migration decision and the partial context sizes; if the prediction of the latter is disabled and this instruction causes the thread to migrate, the migrated context includes the top 2×m main stack entries and 2×a auxiliary stack entries.

fnc_ld_em



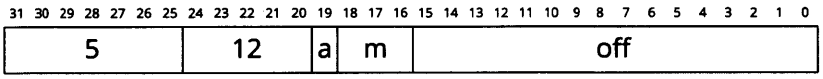
First, stalls execution until acknowledgements for all outstanding stores have been received. Next, removes S[0] and adds it to off; retrieves the word at the resulting address and pushes it on top of the main stack. Always migrates if the address is not cacheable in the local core; in this case, the migrated context includes the top 2×m main stack entries and 2×a auxiliary stack entries.

fnc_ld_ra



First, stalls execution until acknowledgements for all outstanding stores have been received. Next, removes S[0] and adds it to off; retrieves the word at the resulting address and pushes it on top of the main stack. Always executed via remote cache access if the address is not cacheable in the local core.

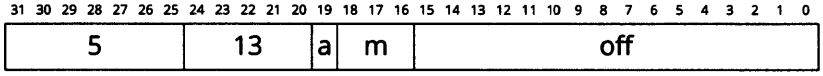
fnc_ld_rsv



Fenced atomic load-reserve. First, stalls execution until acknowledgements for all outstanding stores have been received. Next, removes S[0] and adds it to off; retrieves the word at the resulting address and pushes it on top of the main stack, simultaneously writing the address

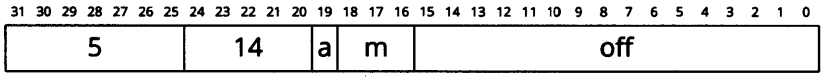
into the reservation register at the home core of the address. By default uses the migration predictor for both the migration decision and the partial context sizes; if the prediction of the latter is disabled and this instruction causes the thread to migrate, the migrated context includes the top 2×m main stack entries and 2×a auxiliary stack entries.

fnc_ld_rsv_em



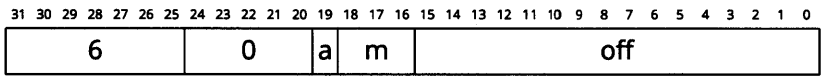
Fenced atomic load-reserve. First, stalls execution until acknowledgements for all outstanding stores have been received. Next, removes S[0] and adds it to off; retrieves the word at the resulting address and pushes it on top of the main stack, simultaneously writing the address into the reservation register at the home core of the address. Always migrates if the address is not cacheable in the local core; in this case, the migrated context includes the top 2×m main stack entries and 2×a auxiliary stack entries.

fnc_ld_rsv_ra



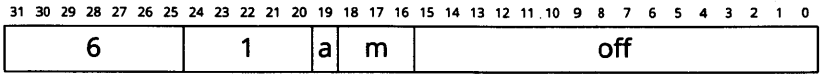
Fenced atomic load-reserve. First, stalls execution until acknowledgements for all outstanding stores have been received. Next, removes S[0] and adds it to off; retrieves the word at the resulting address and pushes it on top of the main stack, simultaneously writing the address into the reservation register at the home core of the address. Always executed via remote cache access if the address is not cacheable in the local core.

st_noack



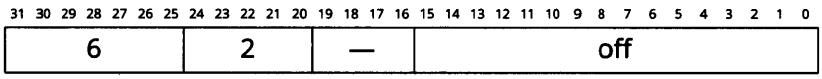
Adds the value of S[0] to off and stores the value of S[1] at the resulting address; removes both S[0] and S[1]. Does not increment the outstanding store counter and does not generate a completion acknowledgement. By default uses the migration predictor for both the migration decision and the partial context sizes; if the prediction of the latter is disabled and this instruction causes the thread to migrate, the migrated context includes the top 2×m main stack entries and 2×a auxiliary stack entries.

st_em_noack



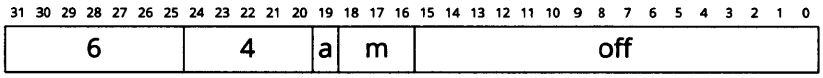
Adds the value of S[0] to off and stores the value of S[1] at the resulting address; removes both S[0] and S[1]. Does not increment the outstanding store counter and does not generate a completion acknowledgement. Always migrates if the address is not cacheable in the local core; in this case, the migrated context includes the top 2×m main stack entries and 2×a auxiliary stack entries.

st_ra_noack



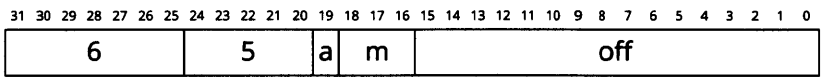
Adds the value of S[0] to off and stores the value of S[1] at the resulting address; removes both S[0] and S[1]. Does not increment the outstanding store counter and does not generate a completion acknowledgement. Always executed via remote cache access if the address is not cacheable in the local core.

st_cond



Atomic store-conditional. Adds the value of S[0] to off and stores the value of S[1] at the resulting address if and only if the reservation register at the home core contains that address; removes both S[0] and S[1], and writes 1 on top of the stack if the store was completed or 0 if the reservation had been broken. By default uses the migration predictor for both the migration decision and the partial context sizes; if the prediction of the latter is disabled and this instruction causes the thread to migrate, the migrated context includes the top 2×m main stack entries and 2×a auxiliary stack entries.

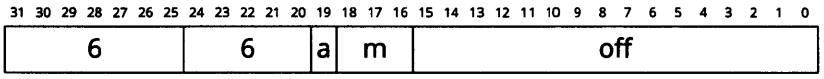
st_cond_em



Atomic store-conditional. Adds the value of S[0] to off and stores the value of S[1] at the resulting address if and only if the reservation register at the home core contains that address;

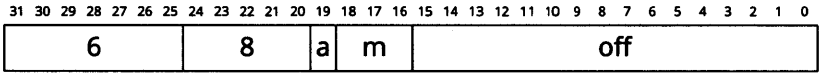
removes both S[0] and S[1], and writes 1 on top of the stack if the store was completed or 0 if the reservation had been broken. Always migrates if the address is not cacheable in the local core; in this case, the migrated context includes the top 2×m main stack entries and 2×a auxiliary stack entries.

st_cond_ra



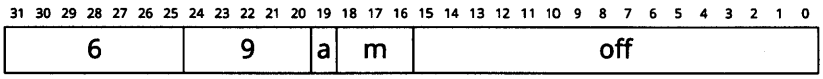
Atomic store-conditional. Adds the value of S[0] to off and stores the value of S[1] at the resulting address if and only if the reservation register at the home core contains that address; removes both S[0] and S[1], and writes 1 on top of the stack if the store was completed or 0 if the reservation had been broken. Always executed via remote cache access if the address is not cacheable in the local core.

fnc_st_noack



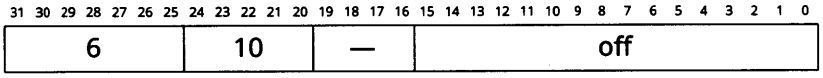
First, stalls execution until acknowledgements for all outstanding stores have been received. Next, adds the value of S[0] to off and stores the value of S[1] at the resulting address; removes both S[0] and S[1]. Does not increment the outstanding store counter and does not generate a completion acknowledgement. By default uses the migration predictor for both the migration decision and the partial context sizes; if the prediction of the latter is disabled and this instruction causes the thread to migrate, the migrated context includes the top 2×m main stack entries and 2×a auxiliary stack entries.

fnc_st_em_noack



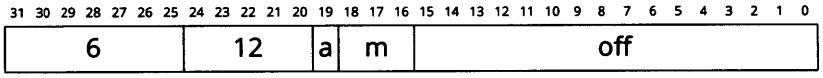
First, stalls execution until acknowledgements for all outstanding stores have been received. Next, adds the value of S[0] to off and stores the value of S[1] at the resulting address; removes both S[0] and S[1]. Does not increment the outstanding store counter and does not generate a completion acknowledgement. Always migrates if the address is not cacheable in the local core; in this case, the migrated context includes the top 2×m main stack entries and 2×a auxiliary stack entries.

fnc_st_ra_noack



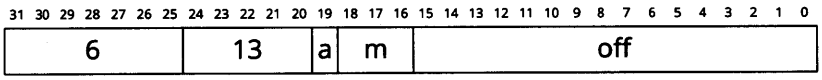
First, stalls execution until acknowledgements for all outstanding stores have been received. Next, adds the value of S[0] to off and stores the value of S[1] at the resulting address; removes both S[0] and S[1]. Does not increment the outstanding store counter and does not generate a completion acknowledgement. Always executed via remote cache access if the address is not cacheable in the local core.

fnc_st_cond



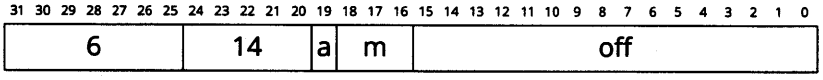
Fenced atomic store-conditional. First, stalls execution until acknowledgements for all outstanding stores have been received. Next, adds the value of S[0] to off and stores the value of S[1] at the resulting address if and only if the reservation register at the home core contains that address; removes both S[0] and S[1], and writes 1 on top of the stack if the store was completed or 0 if the reservation had been broken. By default uses the migration predictor for both the migration decision and the partial context sizes; if the prediction of the latter is disabled and this instruction causes the thread to migrate, the migrated context includes the top 2×m main stack entries and 2×a auxiliary stack entries.

fnc_st_cond_em



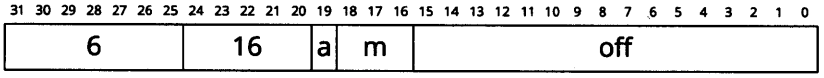
Fenced atomic store-conditional. First, stalls execution until acknowledgements for all outstanding stores have been received. Next, adds the value of S[0] to off and stores the value of S[1] at the resulting address if and only if the reservation register at the home core contains that address; removes both S[0] and S[1], and writes 1 on top of the stack if the store was completed or 0 if the reservation had been broken. Always migrates if the address is not cacheable in the local core; in this case, the migrated context includes the top 2×m main stack entries and 2×a auxiliary stack entries.

fnc_st_cond_ra



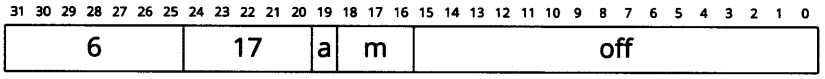
Fenced atomic store-conditional. First, stalls execution until acknowledgements for all outstanding stores have been received. Next, adds the value of S[0] to off and stores the value of S[1] at the resulting address if and only if the reservation register at the home core contains that address; removes both S[0] and S[1], and writes 1 on top of the stack if the store was completed or 0 if the reservation had been broken. Always executed via remote cache access if the address is not cacheable in the local core.

st



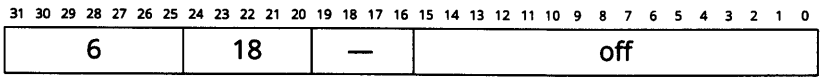
Adds the value of S[0] to off and stores the value of S[1] at the resulting address; removes both S[0] and S[1]. Increments the outstanding store counter and generates an acknowledgement when the store has been committed to the relevant cache. By default uses the migration predictor for both the migration decision and the partial context sizes; if the prediction of the latter is disabled and this instruction causes the thread to migrate, the migrated context includes the top 2×m main stack entries and 2×a auxiliary stack entries.

st_em



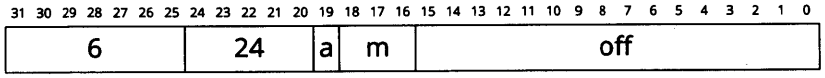
Adds the value of S[0] to off and stores the value of S[1] at the resulting address; removes both S[0] and S[1]. Increments the outstanding store counter and generates an acknowledgement when the store has been committed to the relevant cache. Always migrates if the address is not cacheable in the local core; in this case, the migrated context includes the top 2×m main stack entries and 2×a auxiliary stack entries.

st_ra



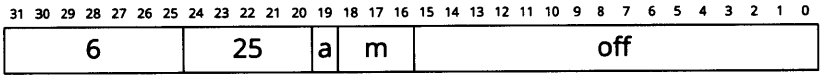
Adds the value of S[0] to off and stores the value of S[1] at the resulting address; removes both S[0] and S[1]. Increments the outstanding store counter and generates an acknowledgement when the store has been committed to the relevant cache. Always executed via remote cache access if the address is not cacheable in the local core.

fnc_st



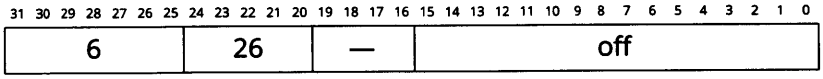
First, stalls execution until acknowledgements for all outstanding stores have been received. Next, adds the value of S[0] to off and stores the value of S[1] at the resulting address; removes both S[0] and S[1]. Increments the outstanding store counter and generates an acknowledgement when the store has been committed to the relevant cache. By default uses the migration predictor for both the migration decision and the partial context sizes; if the prediction of the latter is disabled and this instruction causes the thread to migrate, the migrated context includes the top 2×m main stack entries and 2×a auxiliary stack entries.

fnc_st_em



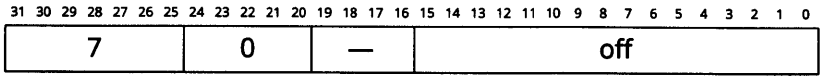
First, stalls execution until acknowledgements for all outstanding stores have been received. Next, adds the value of S[0] to off and stores the value of S[1] at the resulting address; removes both S[0] and S[1]. Increments the outstanding store counter and generates an acknowledgement when the store has been committed to the relevant cache. Always migrates if the address is not cacheable in the local core; in this case, the migrated context includes the top 2×m main stack entries and 2×a auxiliary stack entries.

fnc_st_ra



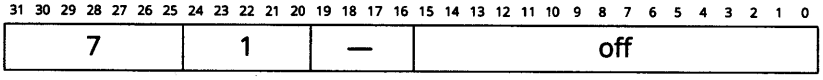
First, stalls execution until acknowledgements for all outstanding stores have been received. Next, adds the value of S[0] to off and stores the value of S[1] at the resulting address; removes both S[0] and S[1]. Increments the outstanding store counter and generates an acknowledgement when the store has been committed to the relevant cache. Always executed via remote cache access if the address is not cacheable in the local core.

j



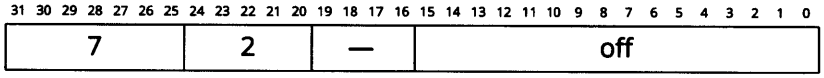
Absolute jump. Removes S[0], adds it to off, and resumes execution at the resulting address.

j_pc



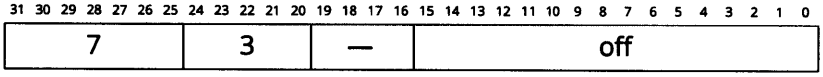
PC-relative jump. Adds the address of the next instruction to off, and resumes execution at the resulting address.

call



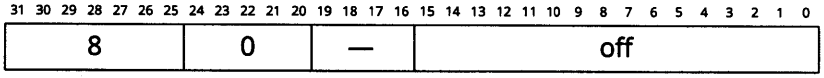
Absolute subroutine call. Removes S[0], adds it to off, and resumes execution at the resulting address; pushes the return address on top of the main stack.

call_pc



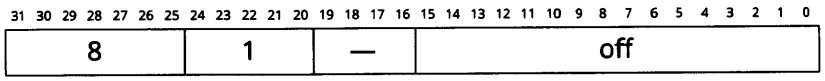
PC-relative subroutine call. Adds the address of the next instruction to off, and resumes execution at the resulting address; pushes the return address on top of the main stack.

bz



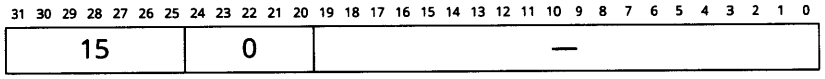
PC-relative branch on zero. Removes S[0]; if the value is 0, adds the address of the next instruction to off, and resumes execution at the resulting address. Otherwise equivalent to a no-op.

bnz



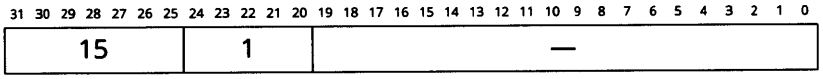
PC-relative branch on non-zero. Removes S[0]; if the value is not 0, adds the address of the next instruction to off, and resumes execution at the resulting address. Otherwise equivalent to a no-op.

halt



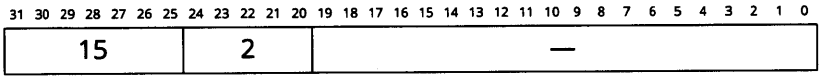
First, if the current thread is running in a guest context, it migrates to its native core. Next, the thread stops execution.

dflush



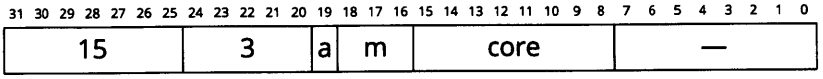
Flushes the entire data cache in the current core.

iflush



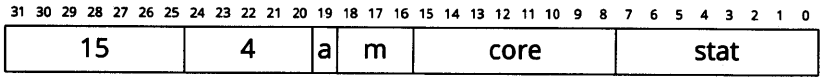
Flushes the entire instruction cache in the current core.

newthread



If the core ID is not core, migrates to core identified by core; the migrated context includes the top 2×m main stack entries and 2×a auxiliary stack entries. Next, removes S[0] and writes its value to the native context program counter. The effects of this instruction are not defined if the value of core is equal to the current thread ID.

reset_stat



Sets statistics register stat in core with ID core to 0. If this instruction causes the thread to migrate, the migrated context includes the top 2×m main stack entries and 2×a auxiliary stack entries.

BIBLIOGRAPHY

D. Abts, S. Scott, and D. J. Lilja. So Many States, So Little Time: Verifying Memory Coherence in the Cray X1. In *PDP*, 2003.

A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The mit alewife machine: Architecture and performance. In *ISCA*, 1995.

Arvind, N. Dave, R. S. Nikhil, and D. Rosenband. High-level synthesis: An Essential Ingredient for Designing Complex ASICs. In *ICCAD*, 2004.

Arvind, N. Dave, and M. Katelman. Getting formal verification into design flow. In *FM2008*, 2008.

M. Awasthi, K. Sudan, R. Balasubramonian, and J. Carter. Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches. In *HPCA*, 2009.

M. M. Beckmann and D. A. Wood. Managing wire delay in large chip-multiprocessor caches. In *MICRO*, 2004.

S. Boyd-Wickizer, R. Morris, and F. Kaashoek. Reinventing scheduling for multicore systems. In *HotOS*, 2009.

K. Chakraborty, P. M. Wells, and G. S. Sohi. Computation spreading: employing hardware migration to specialize CMP cores on-the-fly. In *ASPLOS*, 2006.

M. Chaudhuri. PageNUCA: selected policies for page-grain locality management in large shared chip-multiprocessor caches. In *HPCA*, 2009.

Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Distance associativity for high-performance energy-efficient non-uniform cache architectures. In *ISCA*, 2003.

M. H. Cho. *On-chip networks for manycore architecture.* PhD thesis, Massachusetts Institute of Technology, 2013.

S. Cho and L. Jin. Managing distributed, shared L2 caches through OS-Level page allocation. In *MICRO*, 2006.

B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou. DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism. In *PACT*, 2011.

W. J. Dally and B. Towles. *Principles and practices of interconnection networks.* Morgan Kaufmann, San Francisco, CA, 2003. ISBN 0-12-200751-4.

R. H. Dennard, F. Gaennslen, H. Yu, L. Rideout, E. Bassous, and A. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE J. Solid-State Circuits*, 9: 256–268, 1974.

A. DeOrio, A. Bauserman, and V. Bertacco. Post-silicon verification for cache coherence. In *ICCD*, 2008.

M. Dorojevets and D. Strukov. Memory latency reduction with fine-grain migrating threads in numa shared-memory multiprocessors. In *PDCS*, 2002.

M. Feldman, P. Lotfi-Kamran, K. Balet, and B. Falsafi. Cuckoo directory: a scalable directory for many-core systems. In *HPCA*, 2011.

C. Fensch and M. Cintra. An OS-Based Alternative to Full Hardware Coherence on Tiled CMPs. In *HPCA*, 2008.

N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive NUCA: near-optimal block placement and replication in distributed caches. In *ISCA*, 2009.

J. C. Hoe and Arvind. Scheduling and Synthesis of Operation-Centric Hardware Descriptions. In *ICCAD*, 2000.

W. C. Hsieh, P. Wang, and W. E. Weihl. Computation migration: enhancing locality for distributed-memory parallel systems. In *PPOPP*, 1993.

W. Hu, X. Tang, B. Xie, T. Chen, and D. Wang. An efficient power-aware optimization for task scheduling on noc-based many-core system. In *CIT*, pages 172–179, 2010.

J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler. A NUCA substrate for flexible CMP cache sharing. In *ICS*, 2005.

A. Itzkovitz, A. Schuster, and L. Shalev. Thread migration and its applications in distributed shared memory systems. *Journal of Systems and Software*, 42, 1998.

J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt. Bottleneck identification and scheduling in multithreaded applications. In *ASPLOS*, 2012.

R. Joshi, L. Lamport, J. Matthews, S. Tasiran, M. Tuttle, and Y. Yu. Checking Cache-Coherence Protocols with TLA+. *Formal Methods in System Design*, 22:125–131, 2003.

C. Kim, D. Burger, and S. W. Keckler. An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches. In *ASPLOS*, 2002.

L. Kontothanassis, G. Hunt, R. Stets, N. Hardavellas, M. Cierniak, S. Parthasarathy, Jr. W. Meira, S. Dwarkadas, and M. Scott. VM-based shared memory on low-latency, remote-memory-access networks. In *ISCA*, 1997.

L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comp.*, 1979.

I. Lebedev. Execution model and optimizing compilation for execution migration. Master's thesis, Massachusetts Institute of Technology, 2013.

D. E. Lenoski and W.-D. Weber. *Scalable Shared-memory Multiprocessing*. Morgan Kaufmann, 1995.

M. Lis, P. Ren, M. H. Cho, K. S. Shim, C. W. Fletcher, O. Khan, and S. Devadas. Scalable, accurate multicore simulation in the 1000-core era. In *ISPASS*, 2011.

E. Mascarenhas and V. Rego. Ariadne: Architecture of a portable threads system supporting mobile processes. *Software: Practice and Experience*, 26, 1996.

T. G. Mattson, R. F. Van der Wijngaart, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe. The 48-core SCC Processor: the Programmer's View. In *SC*, 2010.

S. Melvin, M. Nemirovsky, E. Musoll, and J. Huynh. A massively multithreaded packet processor. In *NP2: Workshop on Network Processors*, 2003.

P. Michaud. Exploiting the cache capacity of a single-chip multi-core processor with execution migration. In *HPCA*, 2004.

J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *HPCA*, 2010.

M. Misler and N. Enright Jerger. Moths: Mobile threads for on-chip networks. In *PACT*, pages 541–542, 2010.

M. D. Powell, A. Biswas, S. Gupta, and S. S. Mukherjee. Architectural core salvaging in a multi-core processor for hard-error tolerance. In *ISCA*, 2009.

K. K. Rangan, G.-Y. Wei, and D. Brooks. Thread motion: fine-grained power management for multi-core systems. In *ISCA*, 2009.

D. Sanchez and C. Kozyrakis. SCD: A scalable coherence directory with flexible sharer encoding. In *HPCA*, 2012.

K. A. Shaw and W. J. Dally. Migration in single chip multiprocessor. In *Computer Architecture Letters*, pages 12–12, 2002.

K. S. Shim. *Directoryless Shared Memory Architecture using Thread Migration and Remote Access.* PhD thesis, Massachusetts Institute of Technology, 2014.

A. C. Sodan. Message-passing and shared-data programming models — wish vs. reality. In *HPCS*, 2005.

K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis. Micro-pages: increasing DRAM efficiency with locality-aware data placement. *SIGARCH Comput. Archit. News*, 38:219–230, 2010.

M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *ASPLOS*, 2009.

K. Thitikamol and P. J. Keleher. Thread migration and communication minimization in dsm systems. *Proceedings of the IEEE*, 87:487–497, 1999.

B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating system support for improving data locality on cc-numa compute servers. *SIGPLAN Not.*, 31:279–289, 1996.

B. Weissman, B. Gomes, J. W. Quittek, and M. Holtkamp. Efficient fine-grain thread migration with active threads. In *IPPS/SPDP*, pages 410–414, 1998.

S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *ISCA*, 1995.

H. Zeffer, Z Radović., M. Karlsson, and E. Hagersten. TMA: A Trap-Based Memory Architecture. In *ICS*, 2006.

M. Zhang and K. Asanović. Victim replication: maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *ISCA*, 2005.

M. Zhang, A. R. Lebeck, and D. J. Sorin. Fractal coherence: Scalably verifiable cache coherence. In *MICRO*, 2010.