Marquette University
e-Publications@Marquette

Master's Theses (2009 -) | Dissertations, Theses, and Professional Projects

A Message-Passing, Thread-Migrating Operating System for a Non-Cache-Coherent Many-Core Architecture

Michael W. Ziwisky
Marquette University

Recommended Citation
Ziwisky, Michael W., "A Message-Passing, Thread-Migrating Operating System for a Non-Cache-Coherent Many-Core Architecture" (2012). *Master's Theses (2009 -)*. Paper 156.
http://epublications.marquette.edu/theses_open/156

A MESSAGE-PASSING, THREAD-MIGRATING OPERATING SYSTEM FOR A NON-CACHE-COHERENT MANY-CORE ARCHITECTURE

by

Michael W. Ziwisky

A Thesis Submitted to the Faculty of the
Graduate School, Marquette University,
in Partial Fulfillment of the Requirements for
the Degree of Master of Science

Milwaukee, Wisconsin

August 2012

ABSTRACT

A MESSAGE-PASSING, THREAD-MIGRATING OPERATING SYSTEM
FOR A NON-CACHE-COHERENT MANY-CORE ARCHITECTURE

Michael W. Ziwisky

Marquette University, 2012

The difference between emerging many-core architectures and their multi-core predecessors goes beyond just the number of cores incorporated on a chip. Current technologies for maintaining cache coherency are not scalable beyond a few dozen cores, and a lack of coherency presents a new paradigm for software developers to work with. While shared memory multithreading has been a viable and popular programming technique for multi-cores, the distributed nature of many-cores is more amenable to a model of share-nothing, message-passing threads. This model places different demands on a many-core operating system, and this thesis aims to understand and accommodate those demands.

We introduce Xipx, a port of the lightweight Embedded Xinu operating system to the many-core Intel Single-chip Cloud Computer (SCC). The SCC is a 48-core x86 architecture that lacks cache coherency. It features a fast mesh network-on-chip (NoC) and on-die "message passing buffers" to facilitate message-passing communications between cores. Running as a separate instance per core, Xipx takes advantage of this hardware in its implementation of a message-passing device. The device multiplexes the message passing hardware, thereby allowing multiple concurrent threads to share the hardware without interfering with each other. Xipx also features a limited framework for transparent thread migration. This achievement required fundamental modifications to the kernel, including incorporation of a new type of thread. Additionally, a minimalistic framework for bare-metal development on the SCC has been produced as a pragmatic offshoot of the work on Xipx.

This thesis discusses the design and implementation of the many-core extensions described above. While Xipx serves as a foundation for continued research on many-core operating systems, test results show good performance from both message passing and thread migration suggesting that, as it stands, Xipx is an effective platform for exploration of many-core development at the application level as well.

ACKNOWLEDGMENTS

Michael W. Ziwisky

I simply could not have accomplished this feat without the incredible family, friends, and colleagues that I'm so lucky to have in my life. I am honored to express my sincerest gratitude to:

- My parents, Patricia and Ronald Ziwisky, for their love, encouragement, support, dependability, and money.
- My sister, Carrie Marino, for all of the above except money (though she'd have given that too if I had needed it).
- My extraordinary advisor, Dennis Brylow, for patient guidance, endless inspiration, money, and just being awesome.
- My committee members, George Corliss, Adam Welc, and Mike Johnson, for constructive questions and comments, and for guidance even beyond the scope of this thesis.
- My friend and labmate, Kyle Persohn, for his memory and organizational aptitude, which surely spared me an extra semester or two.
- My friend and lab squatter, Adam Mallen, for excellent conversations, some relevant to the thesis, some enlightening, but most just entertaining.
- My former advisor, Chung-Hoon Lee, for a wealth of guidance and wisdom in my first foray into academic research.
- My favorite hypergeek, Devin Townsend, for unbounded energy, inspiration, and entertainment.
- Evolution, for producing both the coffee bean and the man who discovered the coffee bean.

Thank you all – it's been an amazing time!

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

1.1 Thesis Statement

Xipx, a port of the Embedded Xinu operating system to a distributed x86 architecture, is an effective host for exploring a computational model of concurrent, migratable, promise-based threads on the many-core Intel Single-chip Cloud Computer (SCC).

1.2 Problem Statement

Multi-core processors are ubiquitous, and further scaling in parallelism has led to the emergence of *many-core* architectures. Many-core chips feature dozens of cores communicating with each other via a Network on Chip (NoC) technology. These architectures present a new domain in computation that is ripe for exploration. How does one take advantage of such parallelism? Traditional multi-core machines are cache-coherent, allowing concurrent lightweight threads to access resources in a shared memory space. In contrast, many-core architectures may be non-cache-coherent and are rather suited to executing isolated threads with message-passing communications. Additionally, like any distributed memory system, many-core chips can benefit from computation migration – the ability to

transfer an in-execution unit of computation from one processing core to another. Among other advantages, migration enables active load balancing, which can enhance system throughput and power management.

As a foundational step toward experimentation on a many-core platform, this thesis introduces *Xipx*, a port of the Embedded Xinu operating system [1; 2] to the 48-core Intel Single-chip Cloud Computer (SCC) [3]. Xipx serves as a viable environment for supporting a scalable concurrent computation model centered around migratable message-passing threads – a model well-suited to the emerging class of many-core processors.

As evidence of present interest in such a computational model, Adobe developers currently are building concurrency constructs into the *Tamarin* ActionScript virtual machine [4; 5]. The concurrent computational model is based on share-nothing message-passing threads with "promise-based" inter-thread communications. In general, a *promise* is a placeholder for a value that will exist in the future [6]. In the context of Tamarin, it may also be thought of as a local reference to an object in another thread's address space. Behind the scenes, a promise interacts with an *object proxy* in the other address space, and these two objects communicate via a pair of message-passing channels. While these concurrency constructs may be new to the ActionScript language, JavaScript already supports the model with promise-based "web workers" [7]. With its fast message-passing router mesh, the SCC is a naturally advantageous architecture for

this distributed computational model. Providing a suitable platform for experimental exploration of the model is a primary motivation for our work on Xipx.

The main problems addressed by our initial work on Xipx are how to support efficient message passing between threads and how to support migration of threads. Since message passing is such an elemental component of the many-core paradigm, there has been significant research focused on efficient message-passing implementations for the SCC platform (see Section 3.2.1). However, much of the current research focuses on message passing at the application level, which results in implementations that fully consume the SCC's message-passing hardware to execute just a single parallel application. An important goal for Xipx is to support concurrent execution of multiple parallel programs whose constituent threads may be distributed arbitrarily among the many cores. Our kernel therefore treats the SCC's message-passing hardware as a shared resource that must be managed at the system level.

As an enabling technology with numerous benefits for distributed systems (see Section 3.2.2), computation migration is another important topic in the realm of many-core architectures. An interesting feature of the SCC is that each core has the ability to remap dynamically the logical memory space of any of the cores to a large system-wide memory space. This behavior is exploited in Xipx to reduce dramatically the latency of inter-core data movement. As data transfers are typically the most expensive part of a migration procedure, realizing an efficient

data transfer procedure helps to achieve very low "freeze times" for migrating threads.

Finally, of the numerous modern operating systems targeting many-core architectures (see Section 3.1), the current offerings with ports to the SCC are bloated and complex. The SCC port of Barrelfish is composed of over 450,000 lines of code, while SCC Linux version 1.4.1.2 tops 20,000,000 lines. Such overwhelming codebases make these kernels difficult to understand and modify. In contrast, the agile Xipx kernel comprises under 15,000 lines of code, while providing many of the same features as mainstream kernels such as multitasking with a preemptive priority scheduler, thread isolation and kernel protection, and an API for inter-thread communication. The flexibility offered by Xipx allows for relatively easy development of system-level constructs to exploit SCC-specific hardware features for ground-up support of our target computational model.

1.3 Contributions

Although this work has been focused primarily on developing an environment to provide support for a particular model of computation, the fruits of our labor have resulted in a foundation that opens avenues for a wide range of research opportunities. Concretely, the work comprising this thesis makes the following contributions:

- **Many-Core Port of Embedded Xinu.** Xipx is the first port of the

lightweight and agile Embedded Xinu operating system to a platform in the emerging class of many-core architectures.

- **Message-Passing Device Implementation.** We present a scalable, efficient message-passing device for the Intel SCC that is able to service multiple concurrent threads per core.
- **Messaging Performance Analysis.** Experimental bandwidth measurements of our kernel-level message-passing device reveal its competitiveness with a less flexible user-level library alternative. Iterative measurements during the device's development quantify the performance benefits of SCC-specific caching mechanisms.
- **Thread Migration Implementation.** We present a simple thread migration protocol and two implementations. One is portable to any distributed system, while the other takes advantage of SCC-specific hardware for extremely low-latency inter-core data transfers.
- **SCC "Bare-Metal" Development Framework.** A bare-metal development framework has been created as a launching pad for operating-system-free development of applications targeting the Intel SCC.

1.4 Thesis Organization

The remaining chapters of this thesis are organized as follows.

- Chapter 2 presents an overview of some history and background of many-core computing including cache coherency, sharing data, asynchronous semantics, and parallel architectures.
- Chapter 3 summarizes recent related work in many-core operating systems, message passing implementations, and computation migration.
- Chapter 4 describes the design and implementation of Xipx, a many-core port of the Embedded Xinu operating system, and *BareMichael*, a bare-metal programming framework for the Intel SCC architecture.
- Chapter 5 discloses the measured performance of message passing and thread migration implementations in Xipx.
- Chapter 6 summarizes the thesis and offers suggestions for future work on Xipx.

CHAPTER 2

Overview of Many-Core Computing

This chapter outlines the history and concepts surrounding many-core computing that are relevant to this thesis.

2.1 Origins of Many-Core Computers and Cache Coherency

In the beginning, computers were simple. Early computers only had one central processing unit (CPU), no more than one single instruction could be executed in a single clock cycle, and accesses to random access memory (RAM) occurred at the same speed as executions of CPU instructions [8]. However, the advancement of CPU execution speeds far outpaced that of RAM speeds, and memory accesses soon became a major bottleneck for computers. To address this issue, manufacturers introduced a faster storage medium called *cache* memory [9]. Because it typically is located on the same die as the CPU, and therefore is an expensive commodity, cache memory generally comes in very limited quantity [10]. In contrast, RAM is comparatively plentiful. As such, cache is not a replacement for RAM, but a staging area between RAM and the CPU. It does not provide additional memory space, but redundant space, as it is used to hold copies of data from RAM for low-latency access from the CPU.

2.1.1 Cache Operation

Cache memory has well-established operational principles and terminology [9; 10]. It is organized as a number of fixed-size segments that are called *lines* (one line may be 32 bytes in a typical system [11]), and data copies between RAM and cache occur one entire line at a time. The first time a processor wants to read some data from memory, it simultaneously loads the data into the CPU and loads the line-sized chunk of RAM containing that data into a cache line. If the caching policy is *write-back*, subsequent writes to that cached memory address only update the cache, and RAM will contain outdated data. This condition is recorded by setting a control bit which marks the cache line as *dirty*. Since the size of RAM is so much greater than that of cache, each cache line must be shared between many line-sized sections of RAM, and eventually, an occupied line will need to hold a different section of RAM. When this happens and the current line is dirty, the line gets written back to RAM before it is overwritten by the new section. If the line was not dirty, no write-back occurs. Alternatively, executing a write instruction on a CPU with a *write-through* cache causes RAM to be updated immediately to keep it synchronized with the cache.

2.1.2 Chip Parallelism and Coherency

Advancing technologies in silicon fabrication throughout the late twentieth century enabled the creation of smaller and smaller transistors. With smaller size

came faster switching speeds, and CPU manufacturers were able to realize performance gains by, among other tricks [12], continually increasing the clock frequencies of their chips [13]. However, a chip's power consumption scales linearly with its operating frequency, and heat dissipation becomes a major challenge with clocks operating beyond a few GHz. In response to this situation, a new paradigm has emerged for CPU design. Transistors continue to shrink, but rather than scaling up chip clock frequency, manufacturers now take advantage of the smaller devices by scaling up chip parallelism [14]. Clock speeds remain stagnant, but the number of computational cores on a single die is increasing. An N-core processor can execute N simultaneous instructions each clock cycle, leading to an N-fold performance increase under ideal circumstances.

This paradigm shift has some undesirable consequences. In particular, the combination of cache memory and multiple computing cores introduces some additional complexity into a system [15]. Each core in a multi-core system has its own independent cache and therefore has its own "idea" of what is in RAM. If two or more cores have each cached a copy of the same resource and one core modifies its copy, then all other cores with a copy must be informed of the change so they know not to operate on their stale idea of what that resource is. In a *cache coherent* multi-core system, hardware exists to ensure that when a resource in one cache is modified, the other caches with a copy of that resource know about it [10]. If one of those other cores needs to use the resource again, it will first update its copy to

maintain consistency with the core that modified it. The particular method and timing of this update is a policy decision made by the cache designer [9].

Cache coherency hardware frees the system programmer from much of the burden of dealing with cache, making its existence nearly transparent. Unfortunately, current hardware techniques for maintaining coherency have difficulty scaling beyond a handful of cores [16]. The accompanying hardware overhead, bus use, and power consumption of coherency hardware is significant. Therefore, as the number of cores on a chip increases and we move from the multi-core class to the *many-core* class, the difference between the two is not simply a matter of scale. Current many-core architectures either make no attempt to maintain coherency in hardware, or they use a network-on-chip (NoC) technology to do the job. In addition, inter-core communication and RAM accesses on a many-core typically are carried out over a NoC rather than via a traditional bus [3; 17; 18].

2.2 Terminology: Processes and Threads

Having explored some important aspects of multicore hardware, we will now discuss relevant software concepts involved in parallel programming. In this section, we cover some terminology that typically is used in the literature [19].

A *process* is an instance of a computer program that is executing in a multiprocessing computer. This includes the code and static data of the written program along with stack and heap memory that are used during execution. With a

technique called *virtual memory management*, modern CPUs have the ability to map system addresses to physical addresses arbitrarily (with a certain granularity). For example, if the CPU executes an instruction to read a value from some memory address, the virtual memory management hardware translates that address before going to RAM to request the read. This ability to remap memory addresses is used to isolate each process both from other processes in the system and from the operating system kernel itself. Therefore, each process exists in its own unique memory space. A schematic illustration of virtual memory mapping is seen in Figure 2.1.

Every process consists of one or more flows of execution, or *threads*. A thread is the thing that "does the work" in a process. It is a logical division of a process' work. A process may be composed of multiple threads, each of which can execute concurrently. ("Concurrently" means "at the same time," but it can refer both to threads executing truly in parallel on multiple cores, or to threads sequentially sharing time on a single core [20].) All threads in a particular process exist in the same memory space and therefore share program code and global data. However, each thread has its own state, which includes a stack and register values (including a stack pointer and an instruction pointer). A schematic diagram depicting the relationship of threads and processes is given in Figure 2.2. Shared memory space may be used to share data between concurrent threads, but this practice can lead to certain pitfalls as discussed in Section 2.3.1. Threads are considered to be

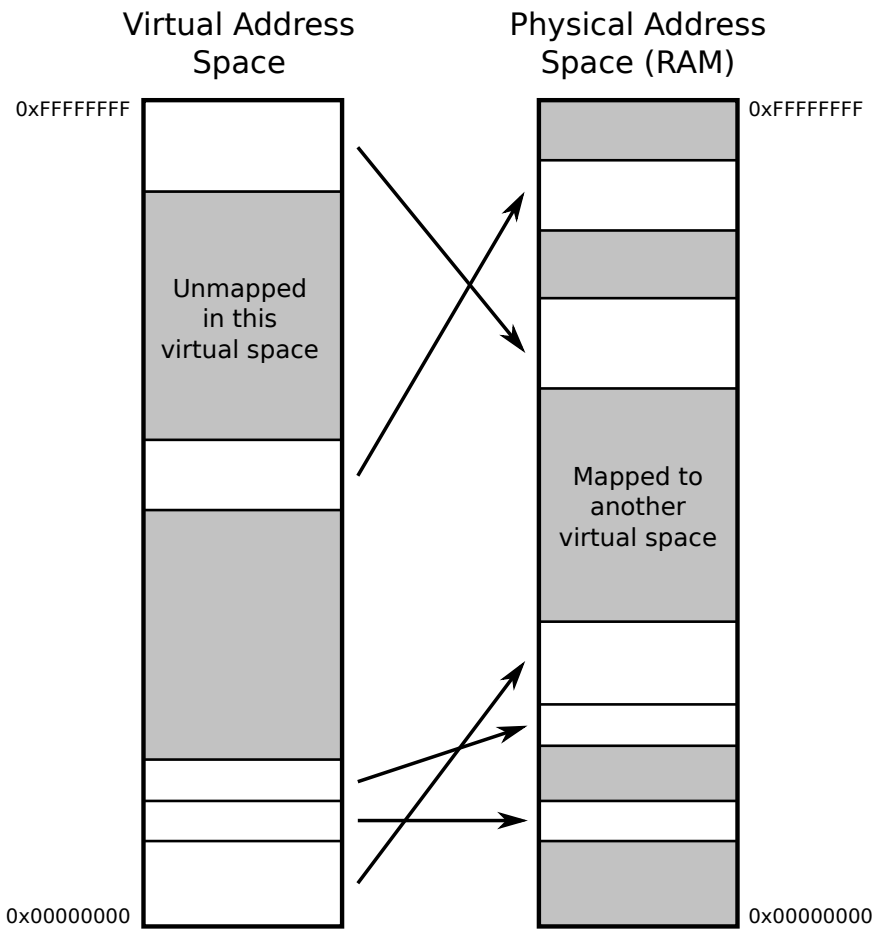

Figure 2.1: Virtual memory management maps addresses in a virtual address space to physical addresses in main memory (after [19]).

"lightweight" in comparison to "heavy" processes mainly because the cost of context switching between two threads is much lower than that of switching between two processes. The increased cost of a process switch comes from the need to change the virtual memory space [21]. While this alone makes the process switch more expensive than a thread switch, another costly side effect is that an entirely new memory space cannot use any cached data, so most memory accesses performed soon after the process switch have to go to RAM. On top of that, the translation

lookaside buffer (TLB), which is a cache for the tables that are used to translate virtual addresses to physical addresses, contains useless data for the incoming process and has to be refilled just to be able to translate the new memory addresses. This results in even more accesses to RAM rather than cache memory.

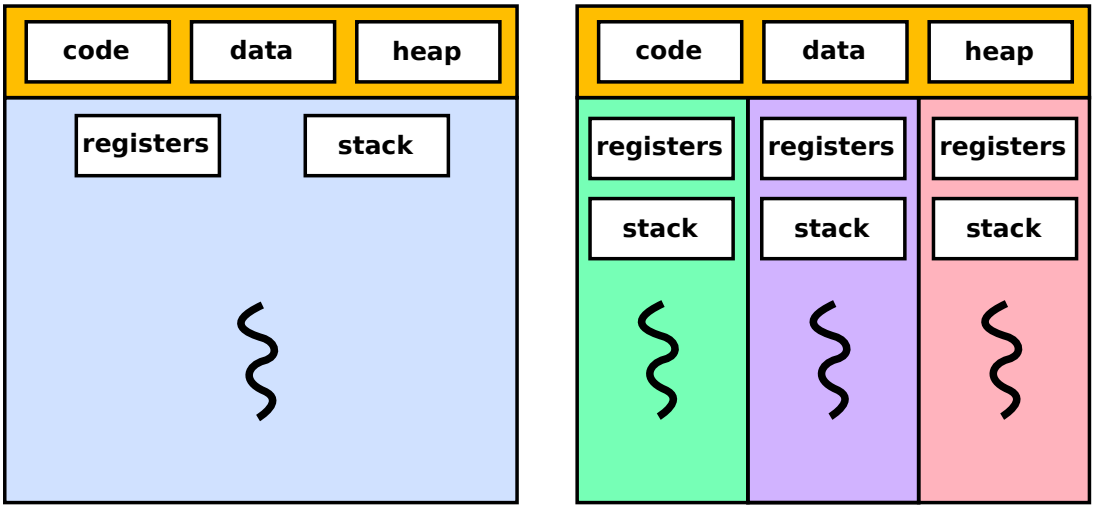

Figure 2.2: A single-threaded process and a multi-threaded process (after [19]). All threads in a multi-threaded process share the same code, static data, and heap space, but each one has its own stack and registers.

Our description of threads so far is fairly general, but we must make a distinction between *kernel threads* and *user threads* [19]. A kernel thread is an operating system construct that is managed and scheduled by the kernel. A user thread is managed in user space without kernel support. Ultimately, a user thread must be associated with a kernel thread in order to be dispatched to a processor. There are three common ways in which user threads get mapped to kernel threads, as illustrated in Figure 2.3. The one-to-one model maps a each user thread to a

kernel thread. The many-to-one model incorporates a user-space scheduler that maps many user threads to a single kernel thread. The many-to-many model also involves a user-space scheduler, but the scheduler multiplexes many user threads to a smaller or equal number of kernel threads. Further discussion, including tradeoffs for each of the models, is given in [19] and [20].

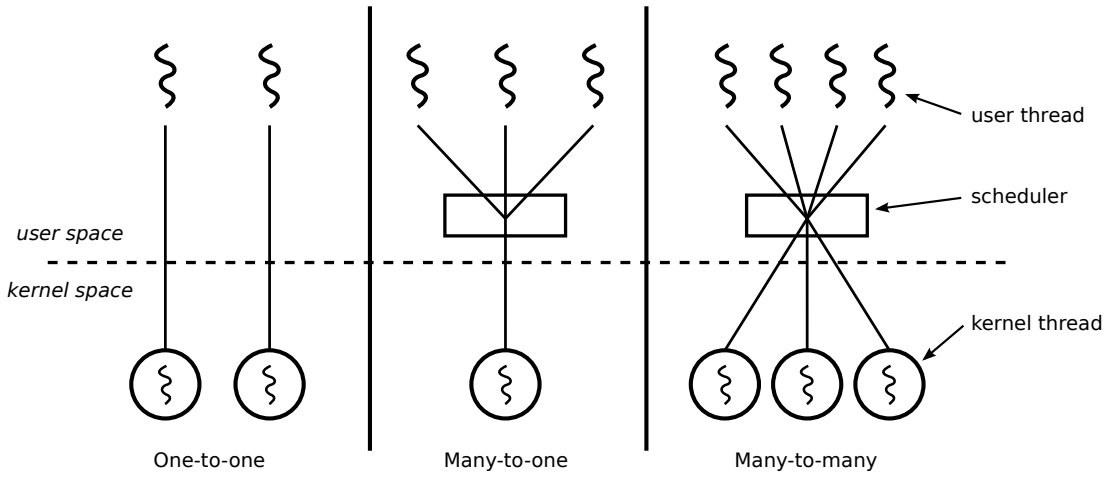

Figure 2.3: Three ways to map user threads to kernel threads (after [19]).

2.3 Sharing Data

This section describes some typical methods for sharing data between concurrent threads or processes.

2.3.1 Shared Memory

Perhaps the most straightforward way for threads to share a resource is to have that resource located in an area of memory that is accessible by all of the threads that want to share it [22]. Since threads of a single process exist in the same memory space, any thread needing access to the shared resource would simply need to know its address. A read-only shared resource may be accessed by any thread at any time without consequence. However, if the resource is able to be modified by executing threads, some synchronization constructs are necessary to ensure that concurrent threads do not see an inconsistent view of the resource.

To see what can go wrong when modifying shared resources, consider spawning two threads of the simple function `increment_var()` shown in Figure 2.4. One would expect the global variable `var` to equal 2 after the threads finished executing. However, looking in Figure 2.5 at the x86 assembly that could implement this C code, we see that the increment is actually split into three instructions: read the data from memory, add one, and write back to memory. Consider two threads executing this code concurrently with the following timing. Thread 1 executes line 5 and 6, then gets preempted. Upon preemption, the value of register `eax` is 1, and this value gets saved during the context switch. Thread 2 then executes the entire `increment_var` function, reading `var` from memory, incrementing it to 1, then

storing it as 1. Some time later, Thread 1 resumes execution at line 7, where it writes the value of register `eax`, which is 1, to `var` in memory.

```
1  int var = 0;
2  void increment_var(void);
3
4  void main()
5  {
6      spawn_thread(increment_var);
7      spawn_thread(increment_var);
8  }
9
10 void increment_var()
11 {
12     var++;
13 }
```

Figure 2.4: A C program that is vulnerable to a race condition.

```
1  var:
2      .word 0
3
4  increment_var:
5      mov     var, %eax  # load var to register eax
6      add     $1, %eax   # increment the value of eax
7      mov     %eax, var  # store eax to var
8      ret
```

Figure 2.5: The vulnerable C function implemented in x86 assembly.

We see that the particular timing of the execution of these two threads has led to an unexpected result. Although we intended for `var` to equal 2 in the end, it

turned out to be 1. When the result of the execution of concurrent threads is unexpectedly dependent on the timing of the execution, the situation is referred to as a *race condition* [23]. Because the timing of concurrent thread execution is generally variable, race conditions can be difficult both to detect and to debug. However, once identified, a race condition can be remedied by ensuring execution of the multiple threads does not occur in a problematic sequence. We define a *critical section* of code as a section that accesses a shared resource that must not be concurrently accessed by multiple threads. In our simple example, the critical section is actually the whole body of `increment_var` – line 12 in Figure 2.4, and lines 5–7 in Figure 2.5.

In a single-core environment, a critical section can be protected simply by disabling interrupts during its execution. This ensures that the thread will not be preempted during the critical section, and a race will not occur. However, this is insufficient in a multi-core environment, where multiple threads are executing simultaneously on separate cores. We need some additional means to enforce *mutual exclusion* – to ensure that no more than one core accesses the critical resource at a time.

Most modern architectures provide mechanisms that may be used to protect critical sections in a multi-core system. (These mechanisms may also be used in a single-core system when disabling interrupts is undesirable.) At the heart of these operations is the ability to update atomically a memory location. An *atomic* update

is one that appears to happen instantaneously. I.e., the procedure of reading, modifying, and writing a location happens all together, uninterrupted, during which all other processors in the system are disallowed access to the location. A simple example of an atomic hardware primitive is a *test-and-set* instruction, the definition of which is given in Figure 2.6 [19]. Passing it a memory location, `target`, `test_and_set()` sets the value at that location to `FALSE` and returns its previous value. Note that this operation is written out in C code for illustration purposes only – the actual execution of the operation must be atomic in order for it to serve its purpose.

```
bool test_and_set(bool *target)
{
    bool ret = *target;
    *target = FALSE;
    return ret;
}
```

Figure 2.6: The definition of the test-and-set operation [19].

If the value at `target` is initialized to `TRUE`, we see that the first time `test_and_set()` is called it returns `TRUE` and sets `target` to `FALSE`. Any subsequent calls will return `FALSE` until `target` is reset to `TRUE` (which can be done by a normal, non-atomic write). A test-and-set therefore may be used to protect a critical section as seen in Figure 2.7. The critical section is not entered until a call

to `test_and_set(&lock)` returns `TRUE`, at which time we say we've acquired the lock. Upon exiting the critical section, we release the lock by resetting `lock` to `TRUE`. Because of its atomic operation, proper use of `test_and_set()` guarantees that only one caller acquires the lock at any time.

```
bool lock = TRUE;

void my_function()
{
    while (!test_and_set(&lock))
        ; /* loop until lock is acquired */

    ... /* critical section */

    lock = TRUE; /* release the lock */
}
```

Figure 2.7: Protecting a critical section with a test-and-set lock.

This issue of ensuring mutual exclusion is a separate problem from that of cache coherency. Coherency generally is taken care of automatically by the hardware, but it is necessarily the programmer's responsibility to avoid race conditions in the software.

While using shared memory for inter-thread communication may seem straightforward, it is prone to pitfalls, such as race conditions, which may be difficult to detect. In the following section, we explore an alternative.

2.3.2 Message Passing

Message passing is fundamentally different from shared memory [22]. Conceptually, each actor in the system owns a mailbox. (By "actor," we mean any entity that can affect memory, including computational cores and memory-mapped I/O hardware, but also including the abstractions of processes, threads, and worker threads.) These mailboxes can be thought of as real-world mailboxes – those that can hold many letters from many different senders. When one actor wants to communicate with another, it sends a message to the other's mailbox. While any actor can send a message to any mailbox, a mailbox can only be opened and read from by its owner. In this way, memory locations are not "shared" because none are ever accessible to more than one actor. Rather, when one actor passes a message to another, the message data is copied directly into the private memory space of the recipient.

Because it does not require a shared memory space, message passing is well suited both for inter-process communication and for communication between worker threads. Abandoning a shared memory model also eliminates the possibility of race conditions since any object may only be accessed by a single thread. Furthermore, when using message passing for communication in a multi-core environment, cache coherency hardware is unnecessary because all memory locations are private to a

single core. In distributed systems, in which separate computational cores have no means to share memory, message passing is the only option for data sharing.

Some message passing APIs offer both synchronous and asynchronous send and receive functions. Beyond just data sharing, synchronous message passing can be used for synchronization of actors. This is particularly useful in an environment where both message-passing and shared memory are in use. For example, before accessing some shared resource, one actor may wait to receive a message from another which indicates that it is safe to do so.

When implementing a message passing system, the properties of the actual message channels must be taken into account. For example, it is generally required that messages sent from one point to another must arrive at their destination in the order in which they were sent. If delivery order is not guaranteed by the hardware, then additional software constructs are needed to resolve the issue. Other channel properties to consider are reliability (can one count on a sent message to always arrive at its destination) and data integrity (is there a chance that a sent message becomes corrupted before it is received).

Further discussion of message passing implementation considerations may be found in Section 4.3.

2.4 Promises and Asynchronous Calls

It is generally agreed upon [19; 24; 25] that concurrent programs are more difficult to write and to understand than sequential programs. However, concurrent programming is necessary to take advantage of the increasing parallelism of computer architectures. Thus, there is a demand for language constructs that make concurrent programs easy to understand and reason about. *Asynchronous calls* are an example of a high-level abstraction that help make a concurrent program look more like a sequential one from the programmer's perspective. When a task makes an asynchronous function call, it immediately receives a returned object, called a *promise*, and continues executing. At the same time, the called function may execute in parallel as an independent task. In general, a promise is a placeholder for a value that will exist in the future [6]. The promise in this case is a placeholder for the return value that is being calculated by the called function. Sometime later in its execution, when the caller needs the return value itself, it uses the promise to retrieve it.

2.4.1 ActionScript Promises

Adobe researchers recently began incorporating concurrency constructs into the ActionScript language and building support for these new constructs into the open source Tamarin virtual machine [5]. These extensions follow a model of

computation that is based upon concurrent worker threads interacting with each other via promises. An initial worker thread – which we will refer to as the local worker – can create another worker thread – a remote worker – from a file containing a worker description. The local worker invokes the `start()` function on the remote worker, which returns a promise representing the remote worker's global scope. The global scope promise gives the local worker access to all globally exposed methods and members of the remote worker.

When a local worker invokes a method call in a remote worker, a promise is always returned, and the local worker continues working immediately. The remote worker exists outside of the local worker's memory space, and therefore an ActionScript promise is a handle to an object in an external memory space. Hidden from the programmer is the fact that the promise interacts, via a pair of one-way message passing channels, with an *object proxy* in the external memory space. The object proxy interacts with the actual object and services requests from the promise to act on or retrieve results from the object. In case the remote method call results in an uncaught exception, the exception is propagated to the caller where it is re-thrown when the caller attempts to retrieve the result of the original call. A graphical illustration of an ActionScript asynchronous remote method invocation is shown in Figure 2.8.

Promise results may be retrieved by the local worker either synchronously, in which case the local worker sends a request to the remote worker and blocks until

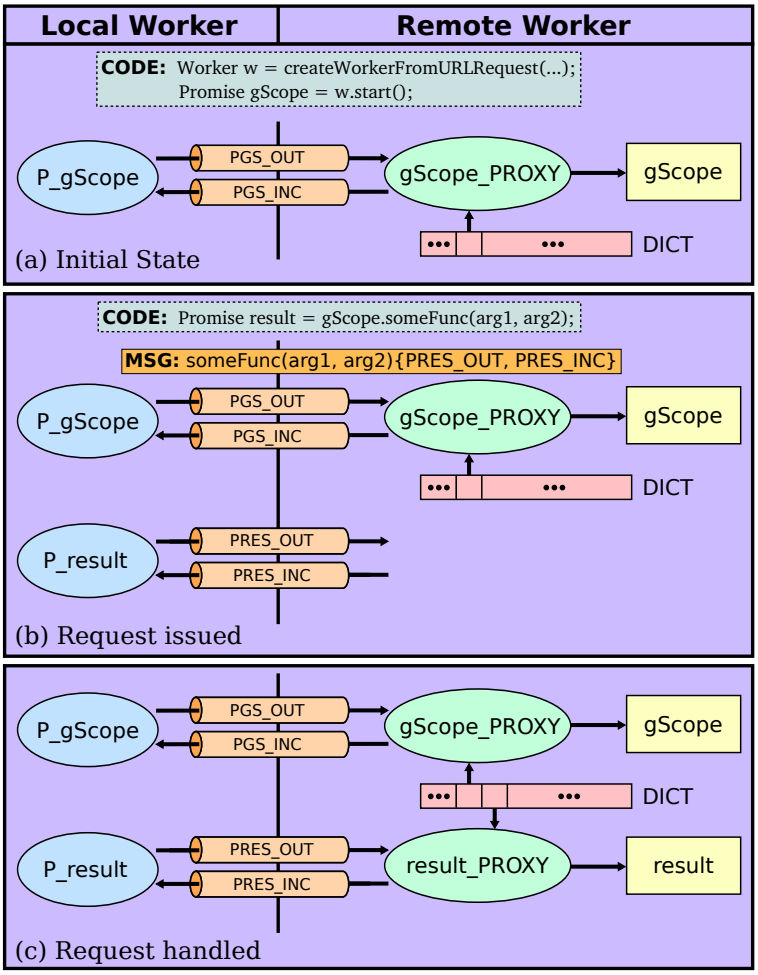

Figure 2.8: Asynchronous remote method invocation (from [5]). (a) Initially, the local worker creates and starts a remote worker; as a result, the local worker holds a promise representing the remote worker's global scope. (b) The local worker creates a new promise which implicitly initializes a pair of message-passing channels ('PRES_OUT and 'PRES_INC'). Then the local worker issues a request to invoke the someFunc() function of the remote worker. Arguments ('arg1' and 'arg2') are passed explicitly while channel information is passed implicitly. (c) Upon completing the requested computation, the remote worker creates an object proxy for the result and links the proxy to the established communication channels.

the result is received, or asynchronously in a slightly more complex fashion. Before issuing an asynchronous result request, a sender creates a local callback function – a *promise resolution handler* – and an object proxy for the callback. The sender then

issues the asynchronous result request on the local promise, which implicitly passes to the remote object proxy the channel information for its callback object proxy. When the remote worker has the result ready, it creates and links a promise with the local callback object proxy and uses it to invoke the callback function. The callback resolves the result of the original promise, and result retrieval is complete.

The semantics of promises allows ActionScript worker threads to be location transparent; i.e., the source code looks the same no matter where workers are physically executing, whether on a single local processor, on a hardware accelerator, or on distributed resources accessed over a network. Thus there is the potential for a single source code to take advantage dynamically of whatever computational resources happen to be available at runtime.

2.5 Intel Single-Chip Cloud Computer

The Single-Chip Cloud Computer (SCC) experimental processor is a "concept vehicle" created by Intel Labs as a platform for many-core software research [3; 26]. It features 48 *GaussLake* processing cores based on the Pentium P54C and a 256 Gb/s bisection bandwidth network-on-chip (NoC) routing mesh. The chip is organized into 24 tiles, each of which contains two cores, a router, and 16 kB of shared memory that is accessible to all tiles via the NoC. This fast, on-chip memory is referred to as the "message-passing buffer" (MPB). A schematic representation of the SCC is seen in Figure 2.9.

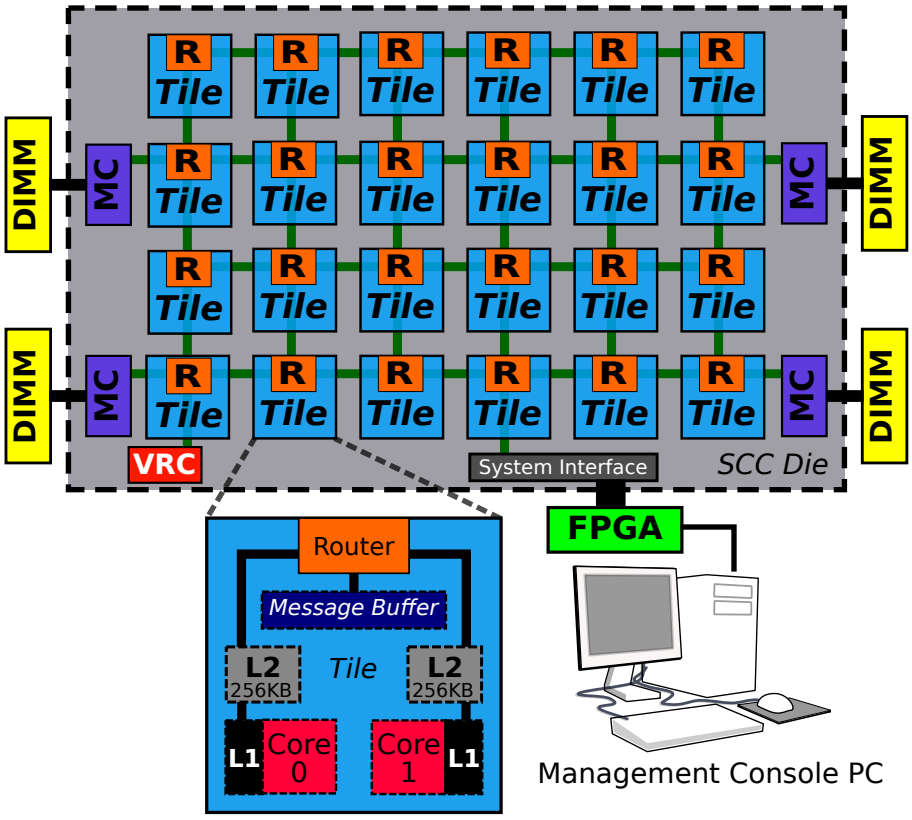

Figure 2.9: Schematic diagram of the 48-core Intel SCC [26].

To reduce hardware overhead and power consumption and to achieve high scalability, cache coherency hardware for shared memory is nonexistent in the SCC. Without coherency, shared RAM is not a viable method for high-performance data sharing between cores. However, access latencies for MPB memory are much lower than they are for RAM, and the MPB therefore offers relatively high performance without caching. In lieu of coherency hardware, the SCC provides the NoC and MPBs as a means to implement an efficient message-passing system for inter-core communications.

The SCC die includes four memory controllers to interface with RAM; a

voltage regulator controller (VRC) capable of scaling chip voltage levels at a granularity of four tiles per domain; and a system interface that interacts, through a field programmable gate array, with a management console PC (MCPC), which is a computer that serves as the user's interface to the SCC. Each of these on-die components is connected to the mesh network and therefore is potentially addressable by any of the 48 cores.

In addition to the cores and NoC hardware, each SCC tile contains a set of configuration registers. These registers include:

- LOCK0 & LOCK1 – a pair of test-and-set locks,
- GLCFG0 & GLCFG1 – a pair of core configuration registers,
- GCBCFG – a global clock configuration register,
- MYTILEID – a read-only unique identifier, and
- LUT0 & LUT1 – a pair of system address lookup tables.

The LOCK0/1 registers are atomic test-and-set locks that can be used to enforce mutual exclusion. These are particularly useful as a tool to prevent race conditions when accessing MPBs for a message-passing implementation.

The GLCFG0/1 registers each have two bits connected to the LINT0 and LINT1 pins of the local advanced programmable interrupt controller (APIC) for their corresponding cores. Each core has access to these bits for all of the cores in the system and therefore can trigger a hardware interrupt on any core.

The GCBCFG register allows a programmer to set the clock frequency of the local tile and of the local router. While all routers in the system must have the same clock to prevent data loss and corruption in the NoC, tiles are able to be clocked heterogeneously. Together with the voltage scaling capabilities provided by the VRC, this frequency scaling allows for fine-grained power management across the chip. The GCBCFG register also has a bit linked to each of the cores' reset pins. Offering the capability to reset discrete cores allows for even greater control over power consumption and facilitates recovery from faults in individual cores.

The MYTILEID register contains a unique identifier for the local tile in the form of its (x,y) location in the 6×4 grid of the chip. The value read from this register differs by a single bit depending on which of the two cores on the tile reads it, allowing each core to identify itself. The most common way to program the SCC is to load the same single image into the private memory of all of the cores. For such homogeneous image configurations, the MYTILEID register can be used to differentiate code paths for different cores via conditional jumps.

On top of the standard segmentation and paging features of a typical P54C core, the SCC has an additional layer of address translation that maps a core's 32-bit physical address into a 46-bit system address. A system address includes the identification of a router and one of the router's ports, and it can refer to RAM, MPBs, configuration registers, the VRC, or the system interface. Each core's LUT register contains 256 entries, one for each of the 16 MiB segments that make up a

core's 4 GiB physical address space. Each entry is able to point to any system address including the LUT itself, which enables dynamic system memory mapping. Exploitation of this ability, along with further details of the LUT and system addresses, is discussed in Section 4.4.4.

2.5.1 Many-Core Alternatives

The SCC is the architecture upon which this thesis work was developed. However, it is worth mentioning a few of its contemporaries to compare the features of the platforms and illustrate some viable alternatives in design.

The Tilera TILE64 is a commercially available processor with 64 cores and multiple independent mesh NoCs [28]. Whereas the SCC has a single mesh for handling traffic generated from events such as memory accesses, I/O accesses, data streaming, and servicing cache misses, the TILE64 distinguishes between these different events by using a different mesh for each one. Five on-chip meshes – called the user dynamic network, I/O dynamic network, static network, memory dynamic network, and tile dynamic network – each serve a particular purpose [27]. A schematic representation of these networks is shown in Figure 2.10.

Tilera has released the TILEPro64 as the successor to the TILE64. Unlike the TILE64 and the SCC, the TILEPro64 maintains cache coherency between cores. The messaging traffic generated by the coherency protocol is significant enough to

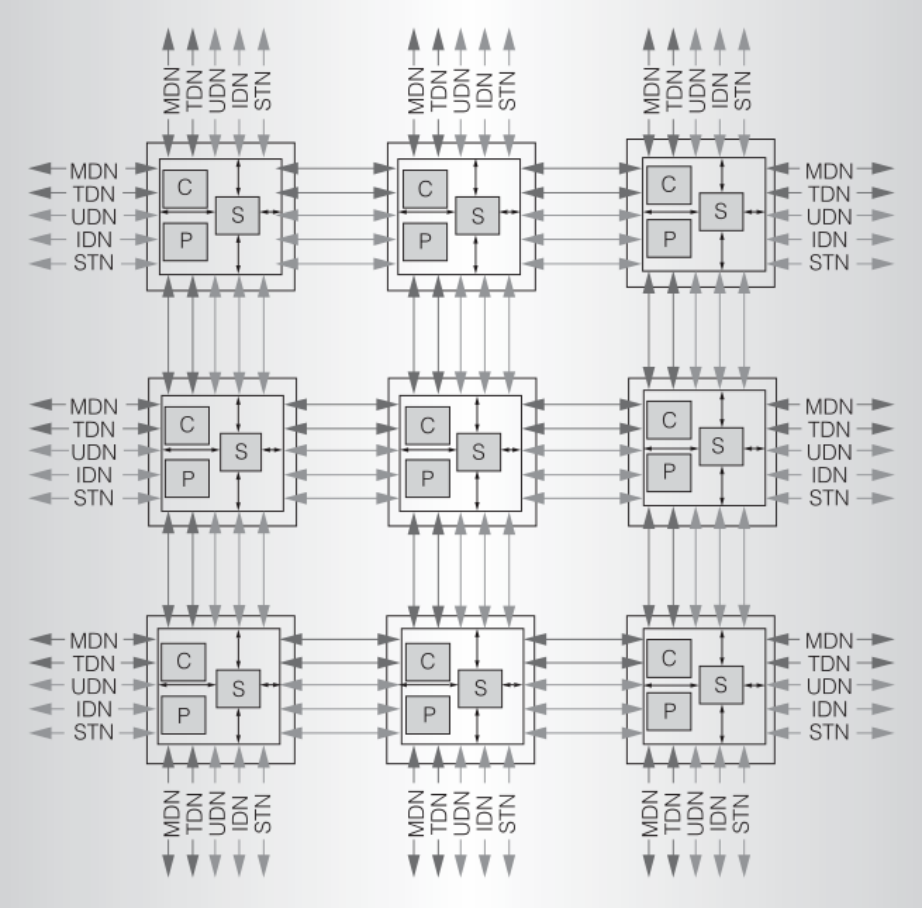

Figure 2.10: Schematic 3 x 3 array of tiles connected by five separate networks on the TILE64 architecture (from [27]).

cause Tilera to dedicate an entire additional mesh network solely to this purpose [17].

Another cache-coherent design, the MIT ATAC is an unrealized architecture focused in part on how to provide inter-core communication when the processor scales beyond 1000 cores [29]. As illustrated in Figure 2.11, it is a tiled architecture featuring a hierarchical interconnection network. The chip is divided into clusters of cores. Each cluster consists of a hub connecting several cores in a star configuration,

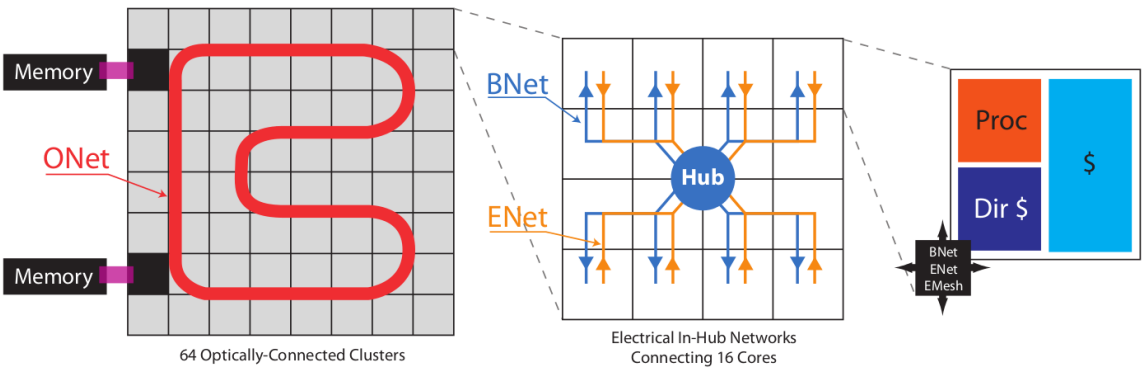

Figure 2.11: Block diagram of the hierarchical on-chip network of the ATAC (from [29]).

and the clusters are linked together by connecting all of the hubs in a ring topology via an optical waveguide medium. While the ATAC architecture provides cache coherency, it does so with a reliance upon this chip-scale optical communication network, an unproven technology that has yet to be successfully mass produced.

2.6 Summary of Many-Core Computing

The evolution of semiconductor fabrication processes has driven computer architectures from single-core processors to cache-coherent multi-core systems, and it is now leading to non-cache-coherent many-core platforms. While some modern architectures with dozens of cores are able to maintain coherency, the technologies used to do so are not scalable. Therefore as chips scale further in parallelism, it is important to figure out how non-coherent architectures can be leveraged by system software for performance gains. Since the absence of coherency precludes efficient shared memory processing, distributed memory paradigms, such as message-passing

communications, offer promising models for the many-core era. The following chapter discusses recent work focused on distributed architectures, with a particular emphasis on work related to the Intel SCC.

CHAPTER 3

Related Work in Distributed Architectures

3.1 Operating Systems for Many-Core Architectures

There are three general approaches to designing an operating system for non-cache-coherent many-core architectures. The first is to manage coherency in software so that traditional symmetric multi-processing (SMP) paradigms may be used, such as employing a shared-memory kernel with data structures protected by locks. The second approach is to abandon shared memory, make inter-core communications strictly explicit, and design the system around this new paradigm. While Boyd-Wickizer et al. argue that the latter is not necessary with currently available hardware [30], they acknowledge that the status quo design would lack performance in a future consisting of processors without high-performance cache coherency. The third design approach is to view a many-core chip as a distributed system and run a separate OS instance on each core. Insights from the fields of networking and traditional distributed systems may benefit such on-chip clusters. While Xipx fits into this last category, this chapter summarizes work that has been done in each of the three domains.

Sobania et al. analyze the aspects of the SCC that fall short of the *Intel MultiProcessor specification* [31], an accepted standard for SMP hardware support

in x86 systems [32]. Two such deficiencies are the architecture's lack of cache coherency and inability to transfer an interrupt vector number as a part of an inter-processor interrupt (IPI) signal. The authors introduce *RockyVisor*, a distributed hypervisor that emulates the necessary support in a virtual layer. RockyVisor could be implemented as either a Type 1 (running directly on the physical hardware) or a Type 2 (running as a process within a host operating system) hypervisor [33]. A Type 2 prototype implementation is still in early stages of development, so it remains to be seen how effective this solution is for distributed architectures.

Lankes et al. are developing *MetalSVM*, a hypervisor that is based on a shared virtual memory (SVM) management system [34]. The authors are implementing MetalSVM as a Type 1 hypervisor and therefore have the freedom to interact directly with the SCC hardware. As a result, much of their published work discusses the low-level mechanisms, such as synchronization [35] and inter-core communications [36], on which the hypervisor will depend. Again, the hypervisor itself is not yet developed to a level that reveals the efficacy of the solution. Our Xipx OS disallows memory sharing between cores and, therefore, has no need for a coherency-enforcing hypervisor layer.

Rather than trying to cast many-core architectures into the cache-coherent mold of their multi-core ancestors, some choose to rethink the mold. Allowing many individual cores to operate within isolated memory spaces begets the likeness of a

distributed system, not a unified system with multiple computing elements. While the latter is traditionally managed by a "single system image" (SSI) kernel, the former demands a new model. Baumann et al. introduce the term "multikernel" to describe their kernel model that is tailored to distributed systems [37]. The three guiding design principles of the model are: making all inter-core communications explicit (via message passing), making OS structure hardware-neutral, and replicating rather than sharing system state. The authors realize a multikernel with their implementation of the *Barrelfish* OS. Originally written for x86-64 multi-core systems, Barrelfish also has been ported to the SCC [38]. While the OS assumes a "shared nothing" paradigm for kernel functionality, it does not enforce the same for applications. Because many modern parallel programs are based on a model of many concurrent threads operating in a shared memory space, Barrelfish implements a shared virtual address space over the multikernel.

Also taking a non-traditional approach, the Factored Operating System, or *fos* [39], is designed specifically for multi-core, many-core, and cloud computing systems, but it does not fit the model of a multikernel. The distinguishing design philosophy of fos is to employ space sharing rather than time sharing. Instead of having user applications compete with kernel services for CPU time, the two are executed on separate, dedicated cores. The kernel itself is similarly factored with different cores hosting different services such as page allocation, process management, and file serving. Space sharing offers a reduction or elimination of

context switching, which in turn enhances cache locality and results in a more efficient use of implicit resources such as caches and TLBs. To ensure scalability and manage protection, fos uses message passing for interactions between cores. The fos design depends upon the existence of systems of sufficiently large scale to justify complete dedication of cores to individual services, and it depends upon a sufficiently low-latency messaging system to ensure that the benefits of cache locality outweigh the overhead of inter-core communications [40]. In contrast, Xipx still runs all OS services along with applications on each core and therefore employs time sharing as much as a traditional OS.

Finally, Intel provides a modern port of the Linux kernel (version 3.1) which, running on the SCC as a separate instance per core, is able to load and launch user applications [41]. SCC Linux is neither a traditional SSI, because it runs as a separate instance per core, nor a multikernel, because the kernel instances are fully separated without global state. Rather, SCC Linux treats the cores of the chip like nodes in a cluster. Parallelism is recognized at the application level, not at the kernel level. RockyVisor uses SCC Linux as its host OS. Xipx is similar to SCC Linux as it is also a distributed OS with a separate kernel instance running per core. Lacking in SCC Linux is built-in support for process migration, an important feature of our Xipx OS.

As the SCC is strictly a prototype platform, it lacks certain hardware that is typically found in a multi-core computer and expected by the Linux kernel, such as

a Basic Input Output System (BIOS). Therefore, SCC Linux entails some non-standard kernel modifications to work with the unique hardware. Intel's original approach to adapt the kernel involved, for example, compiling it with hard-coded values for certain parameters that otherwise would be filled through queries to the BIOS. Non-Intel researchers have since proposed some more portable solutions, including BIOS emulation [42], which are incorporated into later versions of the OS.

Another consideration for many-core OS design is overall system efficiency. While many researchers in the field focus on keeping many cores actively performing useful work in parallel, few focus on the efficiency of each individual core. Vasudevan et al. advise us not to forget about exploitable single instruction, multiple data (SIMD) [43] parallelism, such as SSE vector instructions and GPU resources, in the MIMD world of many-cores [44]. While their Vector Operating System (VOS) design is certainly able to increase performance by eliminating redundancy in parallel operations, it leaves some significant open questions. Should OS designers break from long-standing system call interfaces and force application developers to specify work in terms of vectors of resources? If not, how can the OS recognize and exploit vectorizable operations? As the SCC lacks vector processing hardware, the questions of whether and how to increase SIMD parallelism in Xipx are moot points.

3.2 Inter-Core Communications

The most common form of inter-core communication in a distributed system is message passing. However, computation migration is essentially another form of communication in which the data transferred between cores represent a unit of computation. This section surveys work done in each of these realms.

3.2.1 Message Passing

Message passing is a form of inter-process communication that has roots in the early days of distributed computing [45; 46]. It is useful in systems that consist of communicating processes that reside in disjointed memory spaces. The Intel SCC features a distributed memory layout with hardware to support low-latency inter-core message passing, and there are several researchers investigating how to best take advantage of that hardware. While section 2.3.2 discusses the paradigm in detail, this section discusses some of the research that has been focused on message passing both in general and in the context of the Intel SCC.

The Message Passing Interface (MPI) standard is a library specification for message passing [47]. Completed in 1994, MPI Version 1.0 was proposed by an international committee of vendors, researchers, implementors, and users. The standard aims to describe portable primitives for communication in distributed memory systems of various scales. These primitives include functions for

point-to-point communications, collective communications, synchronization, and management of channels. With efficient implementations for a wide range of hardware, MPI has become the *de facto* standard for message-passing-based parallel programming [22; 48].

There have been a few independent efforts to develop an MPI implementation for the SCC. The RCKMPI library [49] features three SCC-specific MPICH2 [50] channels: SCCMPB, which uses the message passing buffer exclusively; SCCSHM, which uses off-chip RAM exclusively; and SCCMULTI, which uses a combination of the two. In general, MPI programs can benefit from user-supplied communication topology information, but this feature was not supported in the original RCKMPI. Christgau et al. enabled the feature by rearranging and resizing data structures in the MPB [51]. Separately, the SCCMPB channel later was improved with the addition of dynamic process support (a part of the MPI Version 2 standard) and a more efficient communication protocol [52].

Based on MP-MPICH [53], the SCC-MPICH library [54] stands as another MPI implementation for the SCC. The developers have demonstrated acceptable performance from this library, but they refuse to share it with the community for fear that they may lack the human resources for user support [55].

The proven MPI standard has been influential in the design of other SCC message passing protocols. RCCE [41] (pronounced "rocky") is a message passing library with a semantics based on a subset of the MPI standard. Developed by

Intel, RCCE was co-designed with the SCC hardware. In addition to the communications primitives, RCCE provides an API for SCC power management operations such as frequency and voltage scaling. Although the library offers high performance in terms of message passing bandwidth, it is subject to certain restrictions including that only one parallel RCCE program may be executing on the chip at a time [56]. Additionally, RCCE send and receive calls are synchronous and blocking. In contrast, our device-layer management of the message-passing hardware, discussed in Section 4.3.2, is asynchronous and allows for an arbitrary number of parallel applications to share the hardware concurrently.

The RCCE library can be compiled both for SCC Linux and for "bare metal." Because bare-metal RCCE is merely a library and not an entire execution environment, it alone does not provide the framework needed to run bare-metal applications. Rather, it depends upon a bare-metal environment that will configure an SCC core for 32-bit protected mode execution and provide a few particular POSIX functions, file operations, and C library functions. In Section 4.5, we introduce *BareMichael*, a minimalistic framework for bare-metal program execution on the SCC. BareMichael serves as the foundation for Xipx, and it (optionally) provides sufficient support for the bare-metal RCCE library.

Clauss et al. developed some useful extensions to the RCCE library [57]. Dubbed iRCCE (for "improved RCCE"), their extensions not only add new non-blocking send and receive functions, but also improve the performance of the

standard blocking versions of the functions by optimizing memory copies between MPBs and private RAM. For message lengths above a certain threshold, iRCCE uses pipelining to further improve bandwidth. In addition, iRCCE introduces two wildcard arguments, one of which may be used to receive messages of arbitrary size, the other to receive from arbitrary senders. Standard RCCE requires that both of these parameters are specified explicitly for calls to the receive function. While the non-blocking primitives of iRCCE offer performance benefits, the library requires the user to call a pushing function to make progress. This makes for more complicated code compared to our Xipx device, which implicitly makes progress in the background. Performance comparisons between RCCE, iRCCE, and the Xipx MPB device are given in Section 5.1.

As another improvement on RCCE, Chandramowlishwaran and Vuduc developed efficient collective communication algorithms, namely `broadcast` and `reduce`, that outperform those of the RCCE library by a factor of 22 and 6.4 respectively [58]. The improved algorithms were guided by a performance model the authors built from micro-benchmarks of the SCC NoC.

Also using micro-benchmarks, Rotta measured various latencies in the SCC and used the results to analyze the tradeoffs involved in a number of design choices for a message passing implementation [59]. In addition, he produced a quantitative analysis of several existing protocols. However, the Xipx implementation does not

fit well into the design space proposed by Rotta, and therefore it is not easily analyzed by his methods.

The Barrelfish OS implements a unique message passing layer that uses MPB space only for inter-core notifications [38]. Payloads are transferred via off-chip RAM. This design decision was motivated in part by the limited size of MPB memory, the need to support multiple message channels simultaneously, and the lack of fine-grained cache control. Like our Xipx MPB driver, the Barrelfish driver is interrupt-driven.

3.2.2 Computation Migration

Computation migration is the act of moving an executing unit of computation (e.g., a process or a thread) between two processors in a distributed memory system. Some applications of migration include:

- dynamic load balancing, which is useful for enhanced throughput [60], reduced power consumption [61], and control of thermal balance [62];
- reduction of communication latency, which is gained by moving communicating entities closer together [63];
- and fault tolerance, which is enhanced by migrating work away from failing cores [64].

With the increasing parallelism of new architectures, efficient migration is becoming more important than ever.

Different challenges are faced depending on the migration granularity that is pursued. The state of an executing process consists of many resources including code, static data, dynamically allocated data, local stack data, and register state. It also incorporates communication state, which includes open communication channels and pending outgoing and incoming network messages, and kernel state, which includes open devices and files [19]. In general, all of this state must be transferred to migrate a process. On the other hand, a given thread may not need all of the resources of its associated process, and therefore thread migration has the potential to be a more lightweight procedure. However, a significant challenge in thread migration is determining just how much state needs to go with the thread for it to operate in its new home, and how to deal with the resources that are not migratable [65]. In Xipx, migration is only supported for user threads (Section 4.4.1), which are similar to single-threaded processes. Process migration is therefore the more relevant field of study for the purposes of this thesis, however we will also touch on a couple of recent developments at the coarser- and finer-grained ends of the spectrum.

In a straightforward migration implementation, a process is frozen, then its complete state and address space is transferred to the destination machine, and finally it is resumed at its new location. However, as data transfer is typically the

most expensive operation in a migration [66], this protocol incurs significant latency. As an alternative strategy, the V kernel pre-copies memory pages while allowing a process to continue executing on the source machine [67]. After a complete pre-copy, the process is frozen and its state is migrated, then any pages that were modified during the pre-copy period must be updated on the destination before resuming the process. Copying some pages twice increases the communication overhead, but this strategy greatly reduces the freeze time of the migrated process. Xipx user thread migration is done in the straightforward manner, but it uses unique SCC hardware features to perform rapid bulk data transfer between cores.

An alternative strategy for data transfer is a lazy protocol, in which a minimal amount of resources and address space is transferred initially, and further memory pages get shipped on-demand when they are needed. This approach, used in Accent [68], offers reduced freeze time at the time of migration but incurs many short delays later when additional pages are migrated. A significant advantage is that a process often may not reference a substantial portion of its address space after migration, so many pages may not need to be transferred at all. However, a disadvantage is that the source processor needs to retain the yet-unreferenced resources of any processes that have migrated away. This is one example of a *residual dependency*, which is, in general, a resource on the source machine upon which a migrated process continues to rely. Besides adding complexity and overhead to a system, residual dependencies decrease reliability. A failing processor may take

down not only its local processes, but also any processes for which it holds residual dependencies. A process that makes multiple hops in its lifetime may leave residual dependencies scattered throughout the system and therefore become increasingly fragile as it becomes sensitive to the potential failure of more and more processors [66].

Sprite [69] uses a variation of the lazy data transfer approach which eliminates the residual dependencies discussed above. It relies upon a networked file server to which a process's dirty pages get flushed before the process is migrated. The destination core then retrieves pages from the server rather than from the source core when page faults occur. Dirty pages get transferred twice, both from the source to the server and from the server to the destination, which incurs some increased overhead.

Sprite, however, is not devoid of residual dependencies. The kernel is largely focused on *transparency* in the sense that a process should always appear, both to the user and to all processes in the system, as though it is executing on its "home machine," the machine on which it was created. In other words, from outside of the kernel, it should look like migration never happens. Sprite achieves this goal at the expense of some residual dependencies. For example, a user or process may send the `kill` signal to a process at any time, and, in a transparent system, this signal would be directed to the home machine. If the process has been migrated, then the home machine is responsible for forwarding that signal to its new location. Similarly, some

system calls, such as `gettimeofday`, get forwarded to and serviced by the home node so that migration is transparent to the migrated process itself. To mitigate the problem, Sprite ensures that subsequent migrations after an initial one do not leave residual dependencies on the intermediate nodes, thereby limiting them to the home machine only.

The type of residual dependency just described, that which is due to communication redirection, commonly arises from the effort to enforce migration transparency [70]. The Amoeba OS [71] avoids communication-based dependencies by use of the Fast Local Internet Protocol (FLIP) [72], which associates a network address with a particular process rather than with a host, thus making the address itself migratable. Both LOCUS [73] and V [67] maintain local caches of last-known locations of recently-accessed processes and are therefore able to bypass the home node. While V corrects for stale process location information by broadcasting a request for it, LOCUS retrieves updated information from the home node and therefore retains a dependency. Our initial implementation of migration in Xipx leaves no residual dependencies but sacrifices a degree of transparency. Inter-process message passing remains transparent, but some system calls do not. For example, the `kill` system call can only kill local threads, so a thread may not realize that a once-local thread has migrated away and therefore that it cannot be killed.

In addition to being pursued at the granularity of individual processes, migration has also attracted interest from the virtual machine (VM) community.

With many of the same motivations that drive process migration – including load balancing, fault tolerance, communication latency reduction, and power management – investigators have developed ways to migrate entire VMs across hosts [74; 75; 76]. A VM conveniently encapsulates all of the state of hardware and software running within, thereby simplifying migration by moving not only a set of processes, but also the entire context in which they are executing. Furthermore, this approach enables migration even when the source and destination host machines are running different operating systems. VM migration has been used in practice to move, for example, active web servers [77]. "Live migration" is achieved using the pre-copying introduced in V, and local-area network migrations have been demonstrated with server downtime of only a few seconds. Wide-area network migrations achieve downtimes of tens of seconds.

On the other end of the spectrum from VM migration, task migration involves movement of very fine-grained units of computation. The task parallel programming model [78] prescribes the use of a number of concurrent worker threads to execute a great number of individual tasks. The feasibility of this model is investigated specifically in the context of the SCC by both [79] and [80]. Two common scheduling strategies for task parallelism are *work sharing* and *work stealing*. The work sharing approach uses a single centralized queue from which all worker threads retrieve tasks to execute. The work stealing strategy employs a separate work queue for each worker thread, and threads that exhaust their queue

may attempt to steal tasks from others to stay busy. The authors in [80] assert that work stealing is the superior strategy on the SCC, but the experiments in [79] show that work sharing exhibits better performance when there is a high variation in the amount of work done per task. While these papers do not focus on the technical challenges of process migration, fine-grained task migration is nonetheless a common practice in which a workload is shifted between processing cores.

The distributed Barrelfish Inter-core Adaptive Scheduling (BIAS) scheduler [81], a two-layer extension of the RBED scheduler [82], performs task migration for load balancing. Two different methods of pre-migration task suspension are performed depending on the implementation details of the worker thread framework that is used. In spite of being an unoptimized implementation, the adaptive load-balancing scheduler shows improved throughput for some of the tested benchmarks.

3.3 Summary of Related Work

This chapter has discussed recent work in many-core operating systems of various designs, message passing implementations, and solutions for computation migration in distributed systems. Most of the work covered has been directly related to the Intel SCC. Throughout the chapter, we have mentioned ways in which our work on Xipx compares to the work being discussed. The following chapter

describes Xipx in detail, first introducing the Embedded Xinu kernel from which it is derived, then explaining the many-core extensions that adapt it for use on the SCC.

CHAPTER 4

Xipx: A Many-Core Operating System

In this chapter, we introduce *Xipx*, a lightweight operating system for a modern many-core architecture. The target platform for Xipx is the Intel SCC, a non-cache-coherent, cluster-on-a-chip architecture with hardware to support low-latency inter-core message passing. Due to the lack of cache coherency on the SCC, the many cores of the chip are typically configured to have access to disjoint private memory spaces, thereby resembling a distributed environment rather than a traditional multi-core platform. As a result, system services such as task scheduling are best handled at the core level, not the chip level. Therefore, Xipx is designed to be booted as a separate instance on each core. In the following sections, we discuss the design and implementation of some crucial aspects for our distributed operating system. The chapter concludes with a description of *BareMichael*, a minimalistic bare-metal programming framework for the Intel SCC that was extracted from the Xipx kernel and released as open-source software for public use.

4.1 Original System

Xipx was built on the Embedded Xinu kernel [1; 2]. The original Xinu operating system [83] has been ported to many platforms in the past

quarter-century, and it has a proven track record of both classroom and commercial use. The modernized Embedded Xinu port is a multitasking kernel featuring a preemptive scheduler, dynamic memory management, synchronization and inter-thread communication primitives, and a robust device driver API. Despite its wide set of capabilities, Embedded Xinu remains a lean, agile kernel comprising under 20,000 lines of code. Being composed of such a modest codebase lends to the understandability, accessibility, and adaptability of the kernel, all of which make it an excellent research vehicle. Recently, Embedded Xinu has been the underlying platform for research in such fields as IP-based telecommunications [84; 85] and lock-free software concurrency mechanisms suitable for arbitrary data structures [86].

4.2 Extensions for Many-Core Support

Although Xinu is a very capable operating system for a single-core environment, significant extensions are needed to exploit adequately the unique hardware features of a distributed many-core architecture. We have chosen the Intel SCC as the target platform for the inaugural development of our many-core operating system. The SCC is a highly scalable architecture developed by a major leader in the CPU industry, and there is a sizeable, active community of researchers using it to pursue diverse avenues of research on the next generation of parallel computing [87]. Two of the most crucial areas in which an OS can offer support for

development on a many-core platform are in providing frameworks for message passing and for computation migration.

Computation migration – the act of moving an executing thread or process from one processor to another – is an ability from which any distributed system can benefit. Migration enables active load balancing, which can enhance system throughput [60] and power management [61]. It also allows a single core in a distributed system to launch a parallel task that can spread out to take advantage of the distributed resources.

With its on-tile message passing buffer (MPB) memory and on-chip mesh network, the SCC was designed for fast, scalable inter-core message passing. As a fundamental system resource upon which multiple processes may rely simultaneously, the message passing hardware is something that is best managed by the kernel. Therefore, a natural extension for Xipx is to provide a lightweight framework and adequate API to give users access to the fast message-passing hardware of the SCC.

Although message passing and migration are very general capabilities for a many-core OS, our work on Xipx is motivated particularly by a goal to support the concurrency constructs that currently are being built into Tamarin, the open source ActionScript virtual machine. These extensions follow a model of computation based on memory-isolated, location-transparent worker threads interacting via

message-passing communications [5]. The distributed nature of the SCC makes it an ideal host for such a model.

The following sections present specific details regarding the implementation of the discussed many-core extensions to the Embedded Xinu kernel.

4.3 Message Passing

To improve scalability, the SCC omits cache coherency in favor of a fast on-chip network and low-latency, tile-local memory buffers. Such hardware encourages the use of message passing rather than shared memory for inter-core communications [16]. This section discusses the design and implementation of a message passing framework that takes advantage of this hardware and is consistent with Embedded Xinu's philosophy of lightweight, minimalistic kernel design.

4.3.1 Message Passing with the MPB

Functionally, the SCC message passing hardware is nothing more than on-chip shared memory that is evenly distributed among the 24 tiles [26]. The on-chip routers provide no special support for broadcasting data; they only facilitate writes from a core to a single location. To the programmer, the routers are transparent, and every MPB read and write looks like a simple memory access. It is upon this hardware that a message-passing framework as described in Section 2.3.2 must be implemented.

In [59], a thorough exploration of a many-dimensional design space for a message-passing framework on the SCC reveals the tradeoffs that exist for different design decisions. The dimensions of the space include message placement (sender's MPB vs. receiver's MPB) and notification methods (including polling and inter-core interrupts), and different communication patterns, such as scattering (one core writes one message to many cores) and gathering (many cores each write a message to one core), will benefit from different design choices. For example, a scattering pattern using sender-side placement requires a single write of the data, while scattering with receiver-side placement requires a write per receiver. In contrast, a gathering pattern will benefit from receiver-side placement because local reads are slightly faster than remote reads, and senders can do their remote writing in parallel.

One of our design parameters is the decision of how to allocate MPB memory for messages. In consideration of this dimension, let us assume messages are placed on the MPB of the receiving core. One option is to divide evenly the MPB space into fixed size *message slots*. Each slot may be statically allocated for use only by a particular sender, or the system may allow any sender to use any available slot. While the former may offer a simpler implementation, the latter has the advantage of scalability as the number of slots required does not increase with additional cores. The former also has the drawback of underuse of MPB space when a core has few communicating partners. However, even the implementation with

dynamically allocated slots will underuse MPB space when messages are smaller than a slot. Such a condition is a form of internal fragmentation [19].

In the design chosen for Xipx, messages have receiver-side placement, and they are variable in size. By "variable in size," we mean that there are no predefined message slots; regardless of length, messages are tightly packed on the MPB. (In fact, our implementation constrains messages to begin at cache line boundaries for performance reasons, so there may be as many as $L - 1$ bytes of unused space between messages, where L is the length of a cache line.) To avoid the use of slots and placement restrictions, each message is prepended with a small header that includes source and destination information and the length of the message.

A "first in, first out" (FIFO) queue is created in the MPB space by treating it as a circular buffer with the first few bytes reserved to hold the head and tail indices. Viewing message passing as a producer-consumer scenario, receiver-side placement means a queue has multiple producers but only one consumer. The benefits of this implementation are two-fold. First, while the many producers require mutually exclusive accesses to write to the buffer, the single consumer can pull messages off of the queue at any time. This is possible because the consumer only updates the tail, and the tail is not updated by producers. In other words, there are no shared resources that get updated by both the consumer and some other code. Unfettered consumption is important because message retrieval occurs in Xipx within an interrupt handler, and without the need to acquire a lock for

synchronization, the handler will run with low *jitter*, or variability in execution time. Second, using a FIFO allows the handler to check one memory location – the tail index of the local MPB – to locate the message(s) to retrieve. Using sender-side placement and/or abandoning the FIFO in favor of a structure that allows out-of-order accesses could allow multiple senders to write messages in parallel, but it necessarily would create a more complicated look-up scheme for message retrieval which could increase handler execution time.

Regarding this last trade-off, certainly when there are many cores trying to write to a single receiver at the same time, the bottleneck of forcing sequential writes becomes a much more significant system-level effect than the increased overhead of a look-up scheme. A slotted implementation would alleviate this bottleneck by allowing senders to write messages simultaneously. However, if any slot can be written by any core, then slot allocation still must be done sequentially, although presumably at a lower cost than sequential message writing. Using slots that are preallocated to specific cores eliminates the need for any sequential operations between multiple senders, but this is not very scalable as the fraction of MPB memory any one core may write reduces as $1/(n-1)$ for an n-core system. Considering sender-side message placement rather than receiver-side opens the possibility of parallel writes with dynamically allocated slots, but remote reads on the receiver's end come with a greater latency than local reads. Clearly, message placement decisions can have significant effects on performance for various

communication patterns, and further theoretical and empirical analysis is a worthy avenue for future research.

Another consequence of using variable sized messages delimited by headers is that the MPB becomes a linked list rather than an array. As a result, if a single message header's `length` field is corrupted, the system is unable to determine the location of the next message, and the entire list is lost. This is a common problem for any linked list, but the risk is reduced in our implementation by ensuring the `length` field is hidden from the user and only written by system code.

While this section is concerned with implementation details for a message passing framework, the following section mainly discusses the interface of the Xipx message passing device.

4.3.2 MPB Driver

The message passing hardware of the SCC is exposed to the Xipx user through the standard Xinu device API [83]. Several instances of the MPB device are created in the OS, the number of which is defined at compile time, and each device acts as a two-way message passing channel. To support this abstraction, message headers specify a sender-side and receiver-side socket identifier. The complete set of fields that make up a message header are tabulated in Table 4.1. Threads may allocate and open an MPB device at runtime, and they must specify whether to open each device in ACTIVE mode or PASSIVE mode. In general, an

ACTIVE mode device may send messages only to a single socket on a single core and receive messages only from that same core at a single local socket. A PASSIVE device may send to any socket on any core and may receive from any core, but only from a single specific local socket.

Table 4.1: Message header format.

Field	Length (bits)	Description
`src`	16	core ID of the sender
`dst`	16	core ID of the receiver
`sndrsock`	16	socket of the sender
`rcvrsock`	16	socket of the receiver
`len`	32	length of message in bytes

When an MPB device is opened in ACTIVE mode, the user specifies the `dst`, `txsock`, and `rxsock` to use. While `dst` and `txsock` *must* be specified, the user may pass a 0 as the `rxsock` to tell the device to allocate an unused socket of its choice. When the user calls `write()` on an ACTIVE MPB device, he or she must pass three arguments: the ID of the device, a pointer to a buffer containing the payload to send, and the length of the buffer in bytes. The driver then prepends the payload with a header whose fields are filled according to the values with which the device was opened, and then the header and payload are written to the MPB of the `dst` core. Before returning, the driver issues an interrupt on the receiving core. The handler for this interrupt is described below.

When an MPB device is opened in PASSIVE mode, the only parameter that

matters is the `rxsock`. Calling `write()` on a PASSIVE MPB device requires the same three parameters as it does on an ACTIVE device – a device ID, a buffer pointer, and the buffer length – but the buffer must contain a header in addition to the payload. Since the user explicitly writes the message header, he or she is able to route the message to any destination by setting the `dst` and `rcvrsock`, but he or she is also able to make it look like the message came from any source by setting the `src` and `sndrsock`. The buffer length argument passed to `write()` should be the length of only the payload. The driver copies this length into the `len` field of the message header before delivering the message. PASSIVE mode MPB devices are useful to allow daemons to retrieve and service requests that can come from any core.

Before a `write()` call returns, it triggers an interrupt on the receiving core. The handler for this interrupt looks at the local MPB to find the least recently written message, reads the message header, and searches for an open local device that either (a) is ACTIVE and has a `dst` and `rxsock` that match the `src` and `rcvrsock` of the header, respectively; or (b) is PASSIVE and has an `rxsock` that matches the `rcvrsock` of the header. When such a device is found, the message is copied to a buffer in private memory, and the state of that device is changed to indicate that a message is waiting.

The `read()` function in Xipx is the dual of the `write()` function. It takes three arguments: the ID of the device to read from; a pointer to a buffer in which to copy the retrieved message; and the buffer length, which is the maximum number of

bytes that can be copied into the buffer. The MPB device is implemented as a synchronous device. When a user calls `read()` on an MPB device, the driver checks to see if a message is waiting. If so, the message gets copied into the passed buffer, and the function returns immediately. If there is no message available, the thread blocks and does not resume until one is received. If the device is in ACTIVE mode, only the payload is copied back to the user; for a PASSIVE mode device, the header is also copied. The value returned by `read()` reflects the number of bytes copied into the buffer (including the size of the header for a PASSIVE device).

There are four ways in which a message delivery can fail in Xipx. The simplest way is if a user tries to send a message that is larger than a defined maximum size, `MPB_MAX_PKT_SIZE`. The driver refuses to send messages that are longer than this constant. A delivery also fails if the driver finds that there is not enough room for the message on the receiving core's MPB. Both of these errors are reported to the sender via the return value of the `write()` function. If a message does make it from the sending core to the receiver's MPB, delivery fails if there is no open device that is configured to receive it (based on the device mode and the message header's `src` and `rcvrsock` fields). Finally, failure also occurs if an appropriate device is found to receive the message, but the device has no available buffers in which to store it. Because we implement an asynchronous MPB device, the sender returns immediately after a write to the receiver's MPB succeeds, and therefore it does not get notified in the event that either of these last two failures

occur. This design was chosen for increased performance. However, there are simple ways of implementing a synchronous device in case the user needs the reliability of a blocking send. For example, a sender could clear a single dedicated byte on its MPB prior to a send, then repeatedly poll it while the receiver is handling the message that was just sent. The receiver would change the byte to indicate the result of the reception, and the sender would pass this up to the user.

Optimizations for SCC Hardware

The GaussLake cores of the SCC offer some unique features not found on the standard P54C, the core on which its design is based. Here we discuss these features and how we exploit them to improve the efficiency of our MPB device.

The memory management capabilities of the GaussLake core include a new flag for page table entries. When this flag, called PMB, is set, the page to which the entry refers is considered "message passing buffer type" (MPBT) memory [26]. Accesses to such memory are cached by L1 cache only – L2 cache is bypassed. Each cache line in L1 includes a flag to represent whether the present line is MPBT or not. Since cache coherency for the shared MPB memory must be managed in software, MPBT lines must be explicitly invalidated at appropriate times. Using a heavy-handed `INVD` instruction, which invalidates the entire L1 cache, would degrade system performance by wiping out lines of private memory from L1 and consequently causing more accesses to L2 cache and/or RAM. Instead, the

GaussLake core is able to invalidate all MPBT cache lines without affecting non-MPBT lines via a new instruction called `CL1INVMB`. This instruction only invalidates cache lines – it does not flush them. There is no instruction to explicitly flush MPBT lines from L1.

The core additionally features a write-combining buffer (WCB) which is the size of a cache line and enhances the speed of writes to MPBT memory. When the core issues a write to an uncached MPBT line, the data is not committed to the MPB right away, but buffered in the WCB. Once a series of writes fills the WCB, the whole cache line is written to the MPB in a single burst. The WCB only buffers full cache lines, i.e., data that belongs to a cache-line-aligned set of contiguous addresses. The data in a partially filled WCB gets flushed back to the MPB if the core issues a write to an MPBT memory address that is not part of the cache line that the WCB is currently buffering. To fully exploit the WCB for fast writes to MPB memory, one must first issue a `CL1INVMB` to ensure that the locations to write are not cached in L1. Then one must write entire cache lines, or, if writing a partial cache line, be sure to follow it with a write to a different MPBT cache line to flush the partially filled WCB.

Underestimating the benefits of caching the relatively fast MPB memory, our naive first implementation of the MPB device disregarded these GaussLake features. MPB memory was simply mapped as uncached, and coherence was therefore not a concern. However, in our latest implementation, we found that message passing

bandwidth can be significantly improved by taking advantage of the burst reads and writes offered by L1 cache and the WCB. The benefits of our SCC-specific optimizations are presented quantitatively in Section 5.1.2.

4.4 Computation Migration

While efficient message passing is an important component of a distributed OS, its implementation as a peripheral device does not alter the nature of the Embedded Xinu kernel. On the other hand, enabling the migration of executing threads requires significant extensions to the kernel, and these extensions are the primary attributes that distinguish Xipx from Embedded Xinu. A new type of thread, virtual memory management, indirect device references, and a migration protocol are the enabling features discussed in this section.

4.4.1 User Threads

Traditional Embedded Xinu recognizes a single class of computational unit [83]. Although this unit does not fit perfectly with any one definition given in Section 2.2, it most closely resembles a thread. Each Xinu thread has a unique state including a stack, stack pointer, instruction pointer, and register values. While there are several research ports of Xinu that include virtual memory support, the default version does not, so all threads exist in the same memory space. Because there are no processes in Xinu, the code that a thread executes is contained in the kernel

image, and heap memory is shared among all threads in the system. However, Xinu threads are process-like in the sense that they do not necessarily all work together towards a unified purpose. Also, although Xinu threads can communicate via shared memory, a mailbox system exists in Xinu to deliver single-word messages from one thread to another in a manner that typically is used for inter-process communications. Although the design of Embedded Xinu makes assumptions about what a thread should and should not do (i.e., what kernel functions and resources should and should not be accessed by a thread), there is no actual protection built into the kernel. The kernel and all threads are vulnerable to corruption, as any thread can access and modify the entire memory space.

Besides the lack of protection, the problem with using Xinu threads in a many-core system is that they are not migratable. Xinu is a monolithic kernel, meaning threads get linked together with the kernel to form a single image. Each thread exists at a specific, unique location in memory, and correct execution of the binary code into which the thread gets compiled depends on it being located at that unique spot. If the code is relocated in memory, it will not execute properly. Therefore, a Xinu thread would only be able to migrate from one core to another if the exact memory space that the thread requires were available on the receiving core. In addition, any resources that the executing thread has allocated dynamically and still depends upon would have to migrate to their same locations in the

receiving core's memory space too. Given these restrictions, it is clear that a Xinu thread is not suitable for migration.

Xipx supports the traditional style Xinu thread described above and refers to it as a *kernel thread*. In addition to the kernel thread, Xipx introduces a new class of thread, called a *user thread*, which differs from a kernel thread in a few key ways. Most significantly, a user thread exists in a virtual memory space, meaning it may be located anywhere (with page-size granularity) in physical memory without jeopardizing correct execution. As a result, a core needs only to have *enough* memory available to receive a migrating thread – a much more relaxed restriction than requiring the availability of specific memory locations. Also unique from kernel threads, a user thread holds *indirect* references to kernel resources it may need, such as device IDs. Similar to the effect of virtual memory, this trait reduces migration restrictions based on device availability. Both of these properties are described in more detail in the following sections.

Note that in spite of their chosen names, the two threads described here differ in character from the traditional definitions [19] of user thread and kernel thread given in Section 2.2. A Xipx kernel thread is like a traditional one-to-one mapped thread that is not a part of a process. Another way to think of it is to view the kernel itself as the process-like entity that owns the Xipx kernel thread. On the other hand, a Xipx user thread is like a single-threaded process. In the remainder of

this thesis, the terms “user thread” and “kernel thread” refer to the Xipx versions unless otherwise noted.

4.4.2 Virtual Memory, Protection, and System Calls

In addition to making user threads migratable, virtual memory management adds the benefit of executing each user thread in isolation, thereby protecting the kernel and all other threads from being corrupted by malicious or erroneous threads. The organization of memory in Xipx is illustrated in Figure 4.1. The kernel sees a flat view of physical memory, while each user thread is isolated in its own virtual memory space. Thread code is mapped to the bottom of the memory space, and stack and heap memory are mapped higher up. A thread also has the Xipx kernel mapped flatly into its physical location so that handlers located in the kernel can be called when an interrupt occurs. However, during thread execution the CPU *privilege level* (discussed below) is such that the kernel is neither readable nor writable; only upon receiving an interrupt does the privilege level change so that the kernel can service the interrupt.

Thread isolation through virtual memory management is necessary for system protection, but it is not sufficient. In addition, the system must restrict the set of CPU instructions that is available to the thread. For example, there are instructions that modify the virtual memory mappings. If a thread were allowed to

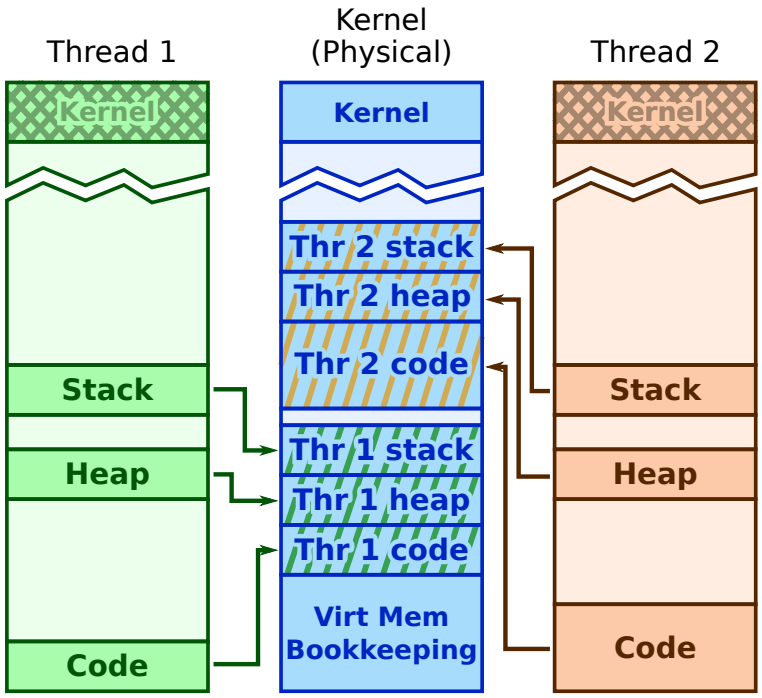

Figure 4.1: User threads are isolated in virtual memory spaces.

execute these instructions, it would be able to get out of isolation and potentially corrupt the kernel and/or other threads.

Since the SCC is based on the x86 architecture, each core of the chip has privilege level management capabilities [11]. At any time, a core is executing in one of four privilege levels. These levels, often referred to as "Rings," are numbered 0 through 3, with numerically greater Rings having lesser privileges. Some CPU instructions (e.g., those that modify virtual memory mappings) are only executable from Ring 0. Others (e.g., I/O accesses) are only executable in some configurable set of Rings. A common practice is for an operating system to only use two Rings – Ring 0 for kernel code and Ring 3 for user code. An x86 interrupt gate is a data

structure that defines how the processor should behave when an interrupt occurs. An interrupt gate can be configured to change the processor to Ring 0 when its associated interrupt is triggered from Ring 3. This is the feature that makes it possible to keep the kernel's interrupt handlers mapped in a thread's memory space so that they can be executed when needed, and yet not be in danger of being corrupted by the thread.

Keeping a user thread isolated in a lower privilege level is necessary to protect the system, but user threads often require access to more privileged system functions for tasks such as I/O accesses and inter-thread communication. For this reason, operating systems provide threads with a *system call* interface – a controlled gateway through which threads may request that the kernel carry out privileged operations on their behalf. A common implementation of system calls on an x86 architecture sets up an interrupt gate that can be software triggered from Ring 3. This gate is configured to change the privilege level of the CPU to Ring 0 and jump to a handler in the kernel that selects which system call to execute based on an identifier the user passes in a general purpose register. When the system call completes, the kernel puts the CPU back into Ring 3 before returning the result to the user thread. Appendix A describes the procedures and CPU configuration that Xipx uses for virtual memory management, privilege level handling, and system call operations, along with some subtleties involved in switching between different memory spaces.

4.4.3 Indirect Device References

As in Embedded Xinu, each instance of a Xipx device has a unique identifier. Typical device use in a Xinu thread starts with a call to the system function `getdev()` or to some other device-specific allocation function. The identifier for the device is returned, and the thread stores that identifier somewhere so that it may be used in future device calls such as `open()` and `write()`. In Xipx, this procedure works fine for kernel threads, but it is insufficient for migratable user threads. If a user thread on core A holds a direct reference to some device and then it migrates to core B, it will try using the device on B that has the same identifier as the one it was using on A. (Device identifiers are not universally unique – they are only unique with respect to the other devices on the same core.) That device on B is almost certainly not configured the way the old device was, and it may even be in use by some other thread. Embedded Xinu does not keep track of which threads are using which devices, so the kernel would be unable to tell that a conflict has arisen.

Xipx addresses this issue by keeping a record of each user thread's allocated devices and only exposing to the thread an indirect, thread-level device reference rather than the true, kernel-level identifier. When a user thread calls any device allocation function, the kernel stores the identifier of the allocated device in an array in the thread table, and it returns to the thread the index to which the identifier was stored. User threads use that index as an argument for any device

calls they make. The kernel then looks up the true device identifier and passes it to the device call along with the other arguments from the user thread. This indirect reference system is illustrated in Figure 4.2.

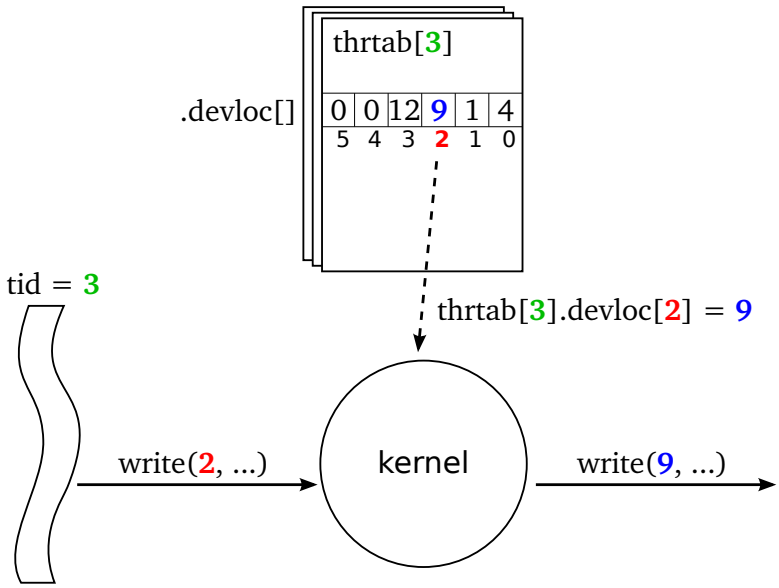

Figure 4.2: The kernel translates thread-local device IDs to system device IDs before calling device functions.

With a record of device usage kept in the kernel, a pair of cores can work out the necessary device dependencies before migrating a thread between them. Prior to migration, the receiving core is given the thread's device array along with sufficient information about all devices that it references. Before committing to receive the thread, the receiving core ensures it has the same types of devices that the thread needs. If it does, these devices are allocated and properly configured, and the entries in the thread's device array are updated to point to them. The thread

continues to use the same local device identifiers without knowing that the global identifiers to which they refer may have changed during migration. After offloading the thread, the originating core can free all of the devices that the migrating thread had been using.

We see that the use of indirect device references has a similar effect as the use of virtual memory in terms of reducing migratability restrictions. Namely, a device is restricted not by the availability of specific devices on the destination core, but only by the availability of the specific *types* of devices to which it holds references. For example, if a thread is using just one device which is of type T and has identifier I, it can migrate to any core with *any* T device available, regardless of that available device's identifier (provided the receiving core also has enough free memory to take on the thread).

4.4.4 User Thread Migration

As previously discussed, thanks to virtual memory management, a user thread may be located anywhere, with page-size granularity, in physical memory. Xipx creates user threads so that their code, stack, and local heap each align with a page boundary. Therefore, when a core requests to offload a thread to a "helper" core, the helper checks to make sure it has enough page-boundary-aligned free memory for each of these three entities. If it does, and it has all of the necessary devices available, then the helper core prepares a new thread with all of these

resources allocated to it. The helper then notifies the offloading core that it is committed to taking the thread, and data transfer can proceed. A detailed description of the current Xipx migration protocol is discussed near the end of this section.

The simple and generalized way of transferring the code, stack, and heap of the user thread is to send everything in packets via the message passing interface. This is generalized because it works in any distributed system with message-passing nodes. However, the SCC features some interesting hardware that can be exploited to transfer relatively large amounts of contiguous data between cores much more efficiently.

The SCC Lookup Table (LUT)

We have mentioned briefly in Section 2.5 that there is a lookup table (LUT) associated with each SCC core that maps each 32-bit (physical) core address to a 46-bit system address. Each LUT consists of 256 entries, each of which maps a 16 MiB segment of the associated core's 4 GiB memory space to some system address. Figure 4.3 illustrates address translation and the composition of a system address.

A complete system address consists of a single bypass bit, an 8-bit destination ID, a 3-bit sub-destination ID, and a 34-bit sub-address. A LUT entry holds the bypass bit, destination ID, sub-destination ID, and upper 10 bits of the

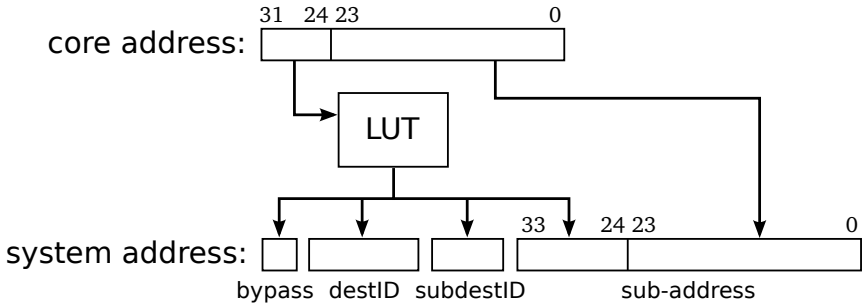

Figure 4.3: Address translation on the SCC (after [26]).

sub-address. The upper 8 bits of the core address index one of the 256 LUT entries, and the lower 24 bits of the core address provide the lower 24 bits of the system sub-address.

Each system address refers to a port of one of the SCC's 24 routers. The destination ID identifies the tile to which the target router belongs, and it is given in (y, x) format – i.e. the upper four bits indicate the y-coordinate and the lower four bits indicate the x-coordinate – where $(0, 0)$ represents the lower left tile, and $(3, 5)$ represents the upper right tile. The sub-destination ID selects which port of the router to access, which will determine whether we are addressing a memory controller, the VRC, the system interface, the configuration registers, or the message passing buffer. Memory controllers, the VRC, and the system interface are connected to specific ports on specific tiles as can be seen in Figure 2.9, but all tiles have router ports linked to configuration registers and the local MPB. Table 4.2 lists the ports corresponding to each of the 3-bit sub-destination IDs, and Figure 4.4

shows the default mapping of a core's 4 GiB address space for an SCC system with 32 GiB RAM.

Table 4.2: System address sub-destination ID ports [26].

subdestID	Port	Comment
0x0	Core0	Not a destination for memory R/W
0x1	Core1	Not a destination for memory R/W
0x2	CRB	Configuration Register
0x3	MPB	Message Passing Buffer
0x4	E_port	Memory Controller @ (0,5) and (2,5)
0x5	S_port	System IF @ (0,3); VRC @ (0,0)
0x6	W_port	Memory Controller @ (0,0) and (2,0)
0x7	N_port	Nothing is on this port of any edge router

Each of the four SCC memory controllers can support up to 16 GiB of RAM, which is completely covered by the 34-bit sub-address space. If the destination is something besides a memory controller, such as an MPB or configuration register, the sub-address provides the offset into these spaces as well. The intended use of the bypass bit is to lower the access latency for the local MPB. When this bit is set, the core bypasses the local router and directly accesses the local MPB. However, Intel researchers have discovered a hardware bug in which data corruption can occur on the MPB when this bit is set, so it is now recommended to never set the bypass bit [88].

LUT ENTRY #	CORE ADDRESS RANGE	MAPS TO SYSTEM ADDRESS SPACE
255	FF000000 – FFFFFFFF	Private RAM (16 MB)
:	:	:
251	FB000000 – FBFFFFFF	VRC
250	FA000000 - FAFFFFFF	MCPC Interface
:	:	:
247	F7000000 – F7FFFFFF	Config Registers – Tile 23
:	:	:
224	E0000000 – E0FFFFFF	Config Registers – Tile 0
:	:	:
215	D7000000 – D7FFFFFF	MPB – Tile 23
:	:	:
193	C1000000 – C1FFFFFF	MPB – Tile 0
:	:	:
131	83000000 – 83FFFFFF	Shared RAM (64MB)
:	:	:
128	80000000 – 80FFFFFF	
:	:	:
40	28000000 – 28FFFFFF	Private RAM (656 MB)
:	:	:
1	01000000 – 01FFFFFF	
0	00000000 – 00FFFFFF	

Figure 4.4: Default LUT configuration for an SCC system with 32 GiB RAM [26].

LUT-Based Data Transfer

The default LUT configuration for each SCC core includes mappings to all 48 of the LUTs themselves, meaning any core can reconfigure any LUT at runtime. This provides the interesting possibility of very rapidly swapping 16 MiB chunks of private memory between cores. Thus, rather than aligning user threads merely to

page boundaries (which are at 4 KiB intervals), Xipx aligns threads to LUT segment boundaries and stores the code, stack, and heap all within one or more segments. The data transfer part of migration is then performed by a few short LUT writes that swap segments between source and destination cores. One tradeoff for getting such simple and quick data transfers is that each user thread, no matter how small it may actually be, now takes up a minimum of 16 MiB of private RAM. However, with a typical hardware configuration having 32 GiB of system memory, this thread size is hardly excessive.

There is another disadvantage of LUT-based data transfers. Before a swap can take place, both of the involved cores must ensure that their cache contains no data from the segment they are about to give up. While the GaussLake cores of the SCC have an instruction to write back and invalidate L1 cache, they unfortunately have no direct control over their L2 cache. Instead, a software routine (described in Appendix B) was written to flush the L2 cache. As discussed in Section 5.2, the cost of this routine is rather low, and migration by LUT-swapping is less expensive than message-passing migration even for relatively small threads. A more thorough investigation of inter-core memory copy operations on the SCC using several different techniques is given in [89].

Updating Message Passing Channels

We have covered the problems of transferring a thread's state from one core to another in a way that is transparent to the migrated thread. However, we have not discussed how migration affects the other threads in a distributed system. In particular, we want to maintain transparency for any threads that have open communication channels with the migrated thread.

One strategy involves forwarding communications through the thread's originating core. However, such residual dependencies are undesirable for a number of reasons (see Section 3.2.2). Another option involves reconfiguring any remote devices that are being used to communicate with the migrating thread. A relatively simple, restrictive, and unoptimized solution of the latter type is used in the migration protocol developed for Xipx. We describe the LUT-swapping version of the protocol below along with an account of its limitations. We also discuss a few reasons why it is difficult to build a more robust solution, noting that this problem is an excellent candidate for future development.

Current Xipx Migration Protocol

We describe here the LUT-swapping version of the Xipx migration protocol, beginning with some terminology and relationships. Each core has a "communication manager daemon" running as a kernel thread and a "migration daemon" that comprises two kernel threads called the "request" thread and the

"helper" thread. The communication manager manipulates local devices based on communications it receives from remote cores. The migration daemon "request" thread initiates migrations of local user threads based on communications it receives from local sources. It does so by spawning an "offloader" thread for each migration request, and that offloader communicates with the helper thread on the core to which the thread will migrate. The "target thread" is the thread that is being migrated, and the "target core" is the core to which it is moving. We call its original home the "source core."

Our system enforces the restriction that migratable user threads are only allowed to open MPB devices in the ACTIVE mode. This ensures that each device communicates with a single end-point, which is necessary to allow the system to determine which external devices need to be updated during migration.

Once an offloader has been spawned with a target thread and a target core specified, communications between it, the target core's helper thread, and the system's communication managers proceed as follows. (See Figure 4.5 for a graphical representation of the protocol.)

The offloader sends to the helper a copy of the target thread's control block and information about the ports used by any MPB devices the thread has open. The helper uses this information to determine what kernel resources are needed by the target thread (e.g., a thread control block, one or more LUT segments, and zero or more MPB devices) and then attempts to allocate those same resources locally. If

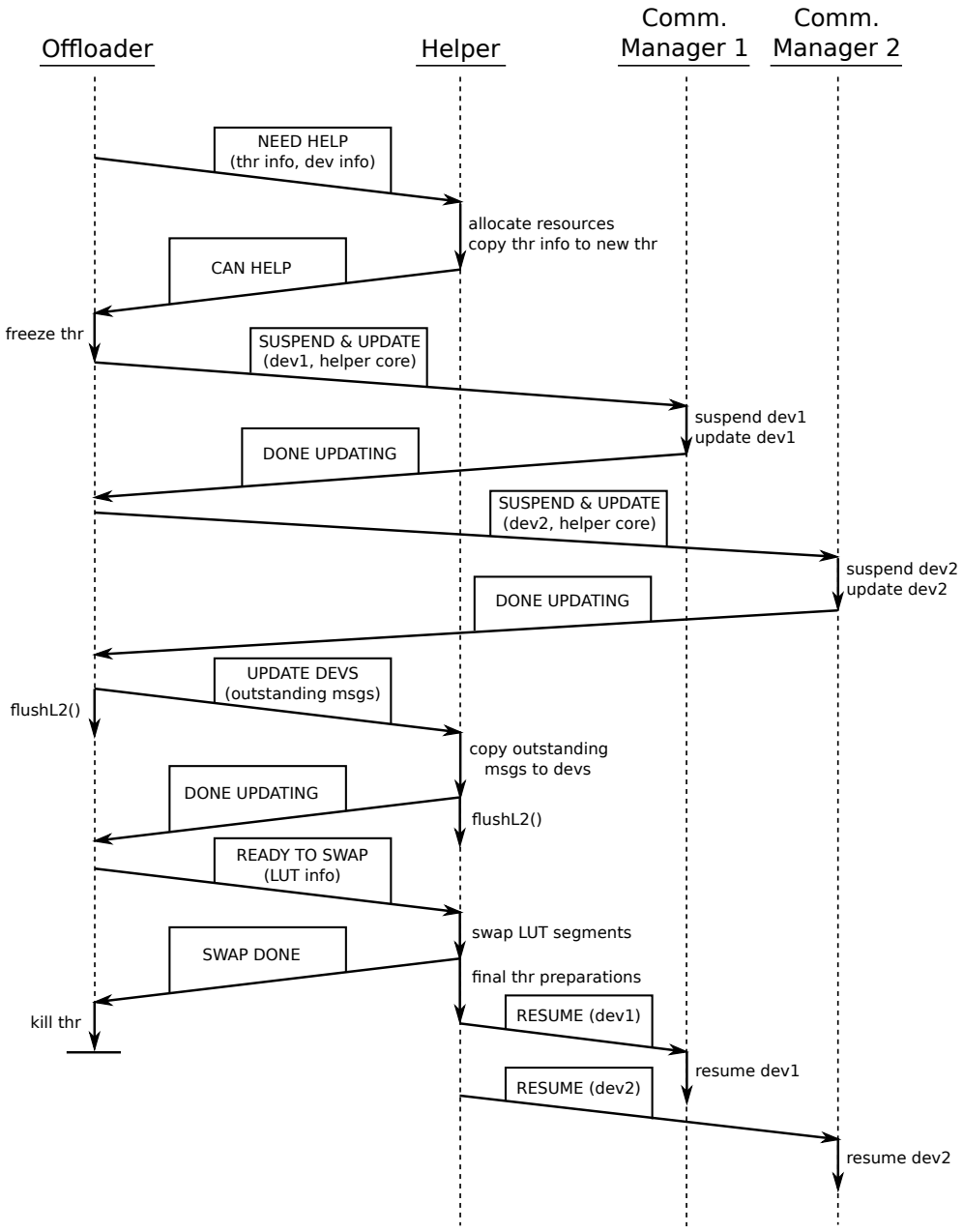

Figure 4.5: Control and communication flow of the Xipx thread migration protocol. Here, the migrating thread has open communication channels with two other cores, and each of those cores' communication managers are involved in the migration.

the local resources are allocated successfully, the helper copies much of the

information from the target thread's control block into the just-allocated local one, and then replies that it is able to help.

The offloader then freezes the target thread and checks which devices it has open. Since each one is an ACTIVE device, the offloader can identify the "partner" device with which it communicates, and it can identify on which core the partner resides. The offloader sends a message to the communication manager on the resident core of each partner. It asks the communication manager to suspend the partner device and update it to direct future communications to the target core. While a device is suspended, calls to read from it are permitted, but any attempts to write to it will block until the device is resumed. Communication managers reply to the offloader to let it know they completed their tasks.

Next, the offloader sends to the helper any buffered messages that the target thread has not yet retrieved. The helper stores these messages in the buffers of the locally allocated devices and notifies the offloader when it is finished.

At this point, the source core and target core are about to swap LUT segments, so both the offloader and helper call `flushL2()` to ensure they have written back from their L2 cache any data that belongs to the segment they are about to give up. Then the offloader sends information about the target thread's LUT segment to the helper, and the helper uses it to perform the swap. The helper notifies the offloader when the swap is complete, and then the offloader kills its local representation of the thread, freeing any resources it had been holding.

Meanwhile, the helper has been preparing the migrated thread for execution in its new environment. This includes building its page table, preparing its stack so that it resumes properly when it gets scheduled, and adding it to the local readylist. Finally, the helper restores communication channels by contacting all of the appropriate communication managers to ask them to resume the target thread's partner devices.

Shortcomings

Although it is suitable under certain circumstances, the migration protocol described above has some significant shortcomings in the Xipx environment. First, we have already mentioned that user threads must only use MPB devices in the ACTIVE mode. Additionally, the protocol is not safe to use if a migrating thread has not already established communication links with all of its partners. Communication channels may be *updated* safely, but the way in which they are *established* currently depends upon the user thread knowing the location and listening port of its partner at runtime. This information may be hard-coded by the programmer, or it may be determined at runtime and shared between communicating partners when they have a parent-child relationship. However, in either case, if migration causes one of the threads to change locations before the MPB channel is established, then the threads' attempts to set up that channel will fail.

A potential solution for this problem is to establish a globally unique identifier for Xipx threads and use that ID, rather than a destination core and port, to specify the endpoint of an ACTIVE MPB device. Globally unique thread IDs would have to be established at runtime – the programmer would have no way of knowing them beforehand. Therefore, the IDs must be shared between threads before they have an MPB channel established, which implies that the threads must have a parent-child relationship. Independently created user threads would not be able to identify each other, so this solution restricts the way in which parallel programs must be written.

Furthermore, for a thread to establish a communication link solely based on its partner thread's ID, there needs to be a way to locate that partner in the system. An efficient and residual-dependency-free solution to this problem is not obvious. A "master core" could be used to keep track of thread locations in the system, requiring all threads to register with the master when they change hosts. However, centralized solutions like this are not scalable because the master becomes a bottleneck as the number of nodes or the migration volume grows. An example of a distributed solution is to broadcast a message asking every core if they host the thread in question, but requiring all cores to spend time searching their thread tables is clearly undesirable.

Xipx is in need of efficient, scalable solutions to lower its restrictions on thread migration. However, as a small, agile kernel, it is an excellent platform for

exploration of solutions that may require significant system-level changes. We leave the problem as an avenue for future work.

4.5 BareMichael and MikeTerm: A Bare-Metal Framework for SCC

The initial stages of porting an operating system to a new platform present significant challenges to developers. It may be difficult to establish a proper build environment for compiling, loading, and executing a system image, and the hardship is compounded by the fact that there usually are no means to deliver feedback to the developer until a sufficient amount of code is written "in the dark" to operate a serial output device. Such barriers took a significant amount of time to overcome in the early phases of the Xipx port, and little support for "bare-metal" (i.e., operating-system-free) SCC development existed.

As interest in bare-metal programming increased in the MARC community [90], we saw it as an opportunity to share our work so that the barriers would be lowered for those who were just starting out. We extracted from Xipx all of the essential parts that are needed to load and launch bare-metal C and assembly code on the SCC with supervisor-level access to all aspects of the chip. This includes the build environment to compile and load an image into memory, the initialization code to bring the SCC cores to 32-bit protected mode and set up a C code execution environment, a set of exception handlers that provide debugging information, and a subset of the standard C library. We also included some SCC-specific helper

functions and definitions to do things such as read the local core ID, read mesh and tile clock frequencies, address MPBs and configuration registers, acquire and release tile lock registers, and trigger inter-core interrupts. A one-way pseudo-terminal program named MikeTerm is used to display output from each SCC core.

We named our minimalistic framework BareMichael. As an open-source tool, every aspect of the package is exposed to the developer who is free to modify, remove, or reimplement its parts at will. The following sections provide technical details about the composition and execution flow of the framework. Their content has been extracted from [91].

4.5.1 Platform Initialization

The following is a brief walkthrough of the code path BareMichael steps through to initialize an SCC core. This description, accurate for the latest versions (4, 5, 6, and 7) of the framework, illuminates the BareMichael startup process so that a developer may understand both how it works and how it may be modified to suit particular needs. Paragraph headers identify the location of the code being discussed, and a schematic representation of the entire process is illustrated in Figure 4.6.

boot/reset_vector.S Based on the Intel P54C, each SCC core boots in "real mode," and consequently has access to just a 20-bit address space. In spite of this

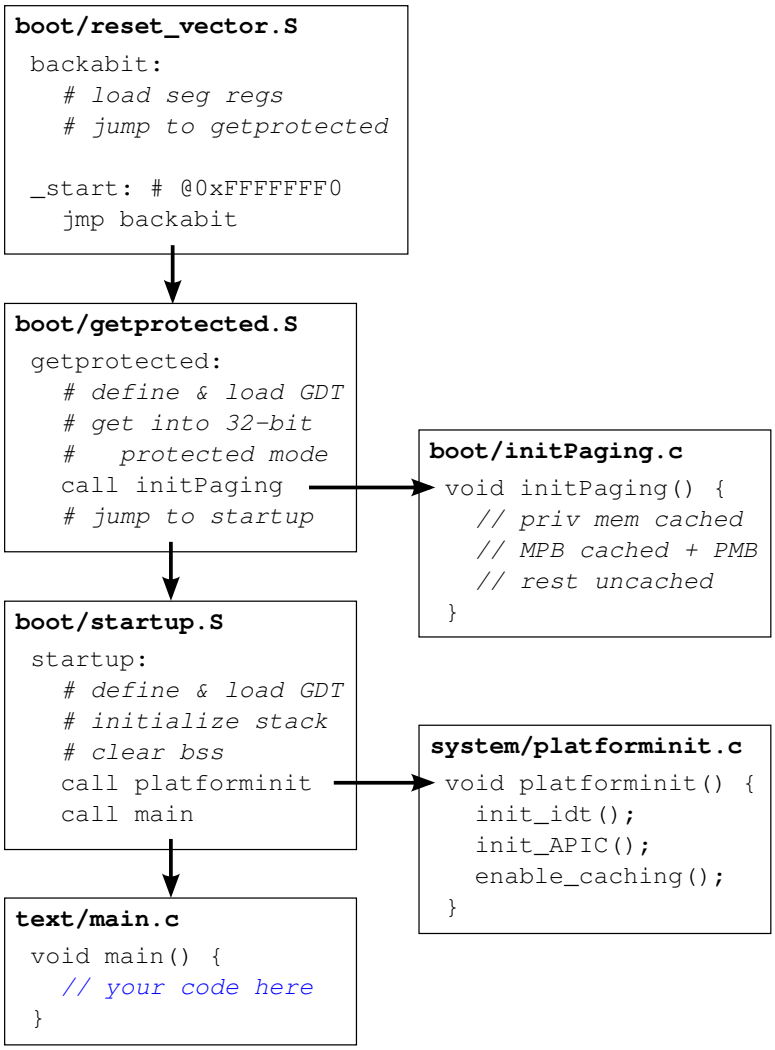

Figure 4.6: Per-core initialization procedure of BareMichael.

limitation, the first instruction a core executes after its reset pin is released is loaded from memory address 0xFFFFFFF0, sixteen bytes from the end of a 32-bit address space. We put a short relative jump instruction here, which takes us back just far enough to initialize the core's segment registers and stack pointer, then far-jump down to a `getprotected()` routine located within the first mebibyte of memory.

boot/getprotected.S The `getprotected()` routine takes the processor into 32-bit "protected mode" by setting up the necessary CPU configuration data structures and registers, including a global descriptor table (GDT) to define flat code and data segments. Then a page table is created for virtual memory management.

boot/initPaging.c The default look-up table (LUT) for an SCC core, which maps core addresses into a larger system address space, splits the core's address space into sections including private memory, shared memory, message passing buffer space, and configuration register space. Our page table flatly maps all of this space with cache disabled for all but private RAM and message passing buffers. Message passing buffer pages also have the PMB flag set to enable special caching features of the SCC [26]. With the page table configured and enabled, the core jumps to the `startup()` routine.

boot/startup.S The `startup()` code gets linked together with `libxc` and the rest of the developer's bare-metal code to create the main image, which may be located in private memory wherever the developer chooses (specified via a Makefile variable). The `startup()` routine defines and loads a new (but identical) GDT within the main image to allow for easier addressing of the data structure should the developer wish to access it later. Space then is allocated for an interrupt

descriptor table (IDT) which will be loaded with descriptors momentarily. After initializing a stack, clearing the `bss` section of the image, and initializing the floating point unit, the core calls `platforminit()`.

system/platforminit.c Among the duties of the `platforminit()` routine are calls to initialize and enable the local advanced programmable interrupt controller (APIC), load the IDT with some default descriptors, and enable caching. As of version 3, the framework includes real-time clock support using the local APIC timer. If this feature is enabled (via a definition in `include/conf.h`), its initialization function is called here. Interrupt vectors 0x00 through 0x1F are reserved for CPU faults and exceptions, and the default handlers BareMichael assigns to these vectors print out information about the state that the system was in when the interrupt occurred. Such information is useful for debugging. After `platforminit()` returns, BareMichael calls the `main()` function in `text/main.c`, which is assumed to be the starting point of the developer's code.

To summarize, we now describe the state of an SCC core after BareMichael initialization. The setup routine brings the SCC core to 32-bit protected mode at privilege level 0 (supervisor level). Virtual memory management is enabled with page table entries present only for the core addresses that are mapped to actual system addresses by the default LUT configuration. Private memory is configured to have cache enabled, MPB-mapped pages have cache enabled and the

SCC-specific PMB flag set, and all other sections have cache disabled. The local APIC is enabled and, by default, its periodic timer is set up to trigger a handler (found in `system/clock.c`) every millisecond. If the framework is configured for RCCE support (see Section 3.2.1), the core's MPB space is initialized to zeros, and a heap is initialized to allow dynamic management of private memory.

4.5.2 MikeTerm

BareMichael applications can print text back to the MCPC through a call to `printf()`. This function simply writes data to a circular buffer in memory where it can be seen and retrieved by the MCPC via the SCC's system interface. Each core has a different buffer allocated for this purpose. Running on the MCPC, a utility called *MikeTerm* acts as a one-way pseudo terminal, periodically polling each of the 48 buffers and printing any text found therein. All output from MikeTerm is preceded by a core identifier. Because MikeTerm scans the shared memory buffers sequentially, it is not guaranteed that its output will be printed in the order in which the cores wrote to their respective buffers. The output from any given core will be delivered in the order in which the core printed it, but ordering of output between any two cores is not necessarily preserved. Additionally, if a core is writing to its buffer faster than MikeTerm is retrieving it, old data will be overwritten and lost without being printed. No protections are built in to prevent this. The default configuration of the framework allocates 64 KiB buffers which get polled by

MikeTerm roughly once per second, so data is likely to be lost when output rates are greater than about 64,000 characters per second. BareMichael currently offers no mechanism for interacting with running SCC programs by feeding data in the other direction, from the MCPC to the chip.

```
[00]: Hello, World -- I'm core 0!

[01]: Hello, World -- I'm core 1!

[05]: Hello, World -- I'm core 5!

[24]: Hello, World -- I'm core 24!

[47]: Hello, World -- I'm core 47!

[00]: I'm going to trigger core 47's LINT0 now.

[47]: I've been interrupted!
[47]: (SCC has been booted for 2 seconds)

[00]: Now I'm toggling core 47's LINT1.

[47]: Another interruption!
[47]: (SCC has been booted for 5 seconds)
^C
Thanks for flying MikeTerm!
```

Figure 4.7: Sample output from MikeTerm. In this sample program, each booted core says "Hello." Then, after a short delay, core 0 toggles each of core 47's APIC interrupt pins with a delay in between. Core 47 has set these interrupt vectors to point to handlers that print out the total time passed since boot up. That time is kept track of by the real-time clock which operates based on the APIC timer and the tile clock frequency.

4.5.3 Build Environment and Dependencies

BareMichael is a flexible framework with few dependencies. This section enumerates the dependencies and describes the build environment, concluding with

a discussion of the ways in which the environment may be modified for extended functionality.

Dependencies

BareMichael leverages some open-source utilities for image compilation, image loading, and delivering output through MikeTerm. The framework uses the `i386-unknown-linux-gnu` cross-compiler tools from gcc version 3.4.5 to produce flat binary object files. *sccKit* is a suite of utilities, provided by Intel, that run on the MCPC and interact with the SCC. BareMichael is compatible with sccKit version 1.4.1, and it uses the `bin2obj`, `sccMerge`, `sccBoot`, and `sccReset` tools for loading binaries into SCC memory and toggling reset pins of individual cores. MikeTerm uses `sccDump` and `sccWrite` to access print buffers in shared memory.

Compilation and Execution

Compilation of both MikeTerm and the SCC image is managed using Makefiles written for the GNU `make` utility. MikeTerm is written in C++ and located in the `miketerm` directory. To compile it, simply change to that directory and invoke `make`.

BareMichael expects the directory containing sccKit binaries to be included in the user's `PATH` environment variable. Paths to the cross-compiler and `bin2obj` tool must be specified in the framework's Makefile, which is located at

`compile/Makefile`. The Makefile also includes a configuration variable for specifying a list of cores to boot. After defining these few variables, compiling and running a bare-metal application is very simple and straightforward. The default `make` target builds the image; the `run` target loads that image into SCC memory and releases the resets of the specified cores. The `main()` function in `test/main.c` is the entry point for the developer's code, and if all of the developer's code is contained in that file (or in any set of files already in the framework), a simple `'make; make run'` is all that is needed to get the code running on the SCC. Follow it up with `'../miketerm/miketerm'` to view output from the cores. If additional source files need to be linked, one must add them to one of two lists in the Makefile: C source files get added to the `C_FILES` list, while assembly files belong in the `S_FILES` list.

Advanced Capabilities

Although most developers probably will be satisfied with the default configuration of the build environment, additional customization is possible. One simple example is changing the memory address to which the main image gets loaded onto the core. This is easy to modify as it is already defined by a variable (`IMG_ADDR`) in the Makefile. However, the framework has other potential capabilities – such as loading and booting different images on different cores – that are possible to realize but not as simple to exploit. For this reason, we disclose the roles of a few files that the build process creates along the way to creating a loadable SCC image.

Initially, the source is compiled into three flat binary object files: the reset vector, the "get protected" and paging initialization code, and the main image. The file `compile/load.map` is created and populated with the names of these three objects, each preceded by the memory address (32-bit core address, not a memory controller address) to which it is to be loaded. This file serves as the input to the `bin2obj` tool, which creates a text file, `compile/battle.obj`, that represents a composite of the three objects. The `sccMerge` tool decides where to load the composite image into SCC memory and how to set initial core LUT configurations. The tool makes these decisions based on three arguments: the number of cores to be served by each memory controller (12 by default), the size per memory controller in GiB (8 by default), and the contents of a `.mt` input file. BareMichael creates the file `compile/battle.mt` and populates it with 48 lines, each of which identifies a core, a memory controller, a "memory slot" (between 0 and 47, inclusive), and a `.obj` file. By default, this file assigns to each core: the nearest memory controller; a memory slot between 0 and 12, which is assigned in increasing numerical order to the 12 cores sharing a memory controller; and the object file that was built earlier, `compile/battle.obj`. The output of `sccMerge` is a directory, `compile/obj/`, and files therein that define the SCC memory contents and the LUT configurations. This directory is provided as an argument to `sccBoot`, which does the actual loading of SCC memory and configuring of LUTs. Finally, the framework issues the command `'sccReset -r <list of cores>'` to release the reset pins of the desired cores.

Clearly, the build procedure may be altered in a few ways – most notably through modifications to the `.mt` file – to customize how SCC memory gets loaded and distributed among cores. As an example, one may arbitrarily assign `.obj` files to cores in the `.mt` file to boot heterogeneous images among the cores. Of course, this requires building multiple `.obj` images, so multiple load maps must be defined and fed to `bin2obj`. Implementation of such alterations is left to the interested developer.

4.6 Integration with RCCE

RCCE [56] is a message-passing software library that Intel Labs designed and implemented in conjunction with the SCC hardware. The current version of the library, V2.0, may be compiled for use in SCC Linux [41], a kernel port also supplied by Intel, or for use in a bare-metal environment. However, because bare-metal RCCE is a library and not an environment itself, it does not provide the execution framework needed to run bare-metal applications on its own. In addition to a CPU initialization process, RCCE demands:

- POSIX functions `mmap()` and `munmap()` for virtual memory management,
- file operations such as `open()`, `flush()`, and `fprintf()`,
- `malloc()` and `free()` for dynamic memory management, and
- various additional C library functions.

These gaps are filled by BareMichael, allowing the developer to use the unmodified bare-metal RCCE library with BareMichael "out of the box." While some features such as dynamic memory management are properly implemented for general use, others, including virtual memory management functions and file operations, are tailored to be compatible with RCCE, though not fully implemented to fulfil their intended duties. These functions are not necessarily safe for use outside of the purpose of supporting RCCE V2.0.

Performance measurements of the bare-metal RCCE message-passing library are given in Section 5.1.

4.6.1 Availability

We provide the BareMichael framework as an open-source package in the hope that it will lower the entry barrier for others wishing to develop and run bare-metal applications on the Intel SCC. The framework is available to download at `http://marcbug.scc-dc.com/svn/repository/trunk/baremetal/baremichael/`.

CHAPTER 5

Performance Analysis

This thesis focuses on the foundational work that has brought Xipx to life on the many-core Intel SCC. Achieving state-of-the-art performance in message passing bandwidth and migration latency is beyond the scope of this work. Nonetheless, it is important to demonstrate reasonable performance in order to justify the claim that Xipx is a suitable platform to support a programming model that is centered around concurrent, migratable, promise-based threads. We disclose in this chapter the measured bandwidth of the Xipx MPB device with a comparison to Intel's user-space message passing library, RCCE [41]. We also exhibit the performance of two different thread migration implementations; one is portable to any message passing system, while the other takes advantage of unique SCC hardware features to achieve a dramatic reduction in latency for large threads. Since the SCC has configurable frequency domains, we note that all tests presented here were performed with all tile, router, and RAM clocks configured to 533 MHz, 800 MHz, and 800 MHz, respectively. Unless otherwise noted, cores 0 and 1 were used in the tests that involve two interacting cores.

5.1 Message Passing

We begin with a report on the measured bandwidth of the Xipx MPB device. A discussion of the device's design, along with details of SCC architectural features that enhance MPB access performance, may be found in Section 4.3.2. The benefits gained by use of these features are quantified below.

5.1.1 Comparison to RCCE

RCCE [41] is a user-space message passing library that Intel Labs designed and implemented in conjunction with the SCC hardware (see Sections 3.2.1 and 4.6). Like the Xipx MPB device, RCCE uses the SCC's MPB space for passing messages between cores. However, unlike the Xipx device, the basic RCCE API only offers *synchronous* (blocking) send and receive functions. As a result, a call to `RCCE_send()` will not return until after a corresponding call to `RCCE_recv()` occurs at the receiver side. A primary design goal for Xipx is the ability to support simultaneous execution of multiple programs that each consist of a number of concurrent threads. Threads of different programs must be able to share a single processing core and each perform message passing communication with remote threads without interfering with one another. Such sharing of the MPB space is not safe with the RCCE library because unpredictable preemptions may cause a thread to intercept messages inadvertently that were intended to be read by a different

thread that is sharing the core. Indeed, the RCCE release specification [56] states that use of the library is constrained to a single parallel program that executes on all or a subset of the SCC cores. In contrast to Intel's offering, the Xipx MPB device is an asynchronous, interrupt-driven device that meets our goal of supporting multiple parallel applications concurrently.

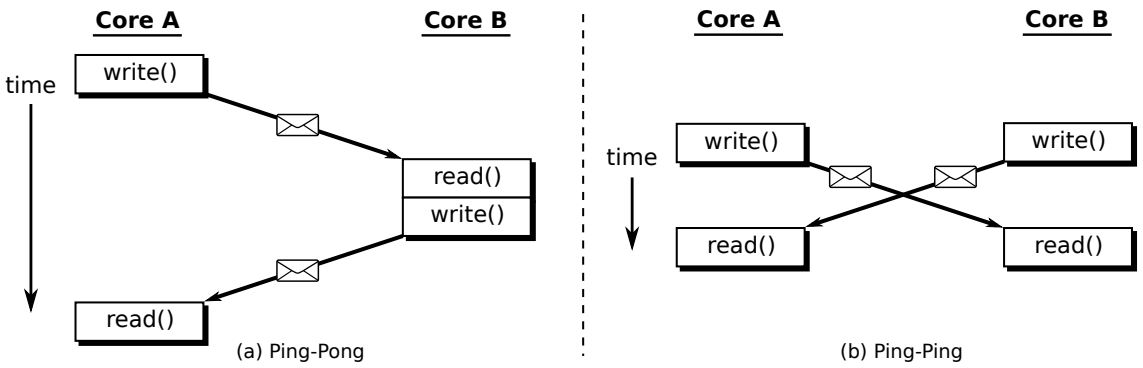

Figure 5.1: Comparison of the communication patterns for (a) the ping-pong benchmark and (b) the ping-ping benchmark (after [57]).

Two benchmarks, whose communication patterns are visualized in Figure 5.1, were used to assess the bandwidth of the Xipx MPB device. The simple "ping-pong" benchmark [41] incorporates two cores, Core A and Core B, each running a single thread. The thread running on Core A sends a message to the thread on Core B; then the thread on Core B sends a message back to the thread on Core A. Both messages have the same payload length, L, which we vary from 1 byte to 4 KiB. We measure the total time that this transaction takes, T, beginning with

the call to `write()` on Core A and ending with the return from `read()` on Core A. Bandwidth is calculated as $2L/T$.

The "ping-*ping*" benchmark [57] involves both Core A and Core B sending a message to each other simultaneously. Again, the time of the transaction is measured from the beginning of the `write()` call on Core A to the end of the `read()` call on the same core. Bandwidth again is calculated as $2L/T$. This benchmark demonstrates a very basic communication pattern that is not achievable with the synchronous primitives of RCCE. If each of two RCCE threads try to send to each other simultaneously, neither will return from their call to `RCCE_send()` because each one will block and wait for the other to call `RCCE_recv()`. On the other hand, the asynchronous semantics of the `iRCCE_isend()` and `iRCCE_irecv()` functions from the iRCCE library are able to implement the ping-ping benchmark.

Benchmark results are seen in Figure 5.2. RCCE and iRCCE measurements were performed on bare-metal implementations supported by the BareMichael framework (Section 4.5), but the respective implementations for SCC Linux performed identically. The ping-pong benchmark was implemented and executed both with Xipx user threads and with Xipx kernel threads (Section 4.4.1). The lower performance of the user thread implementation is due to the overhead that user threads incur from the Xipx system call interface (described in Appendix A). The proportional difference is generally smaller for larger messages because the relatively constant number of extra cycles spent passing through a system call gate

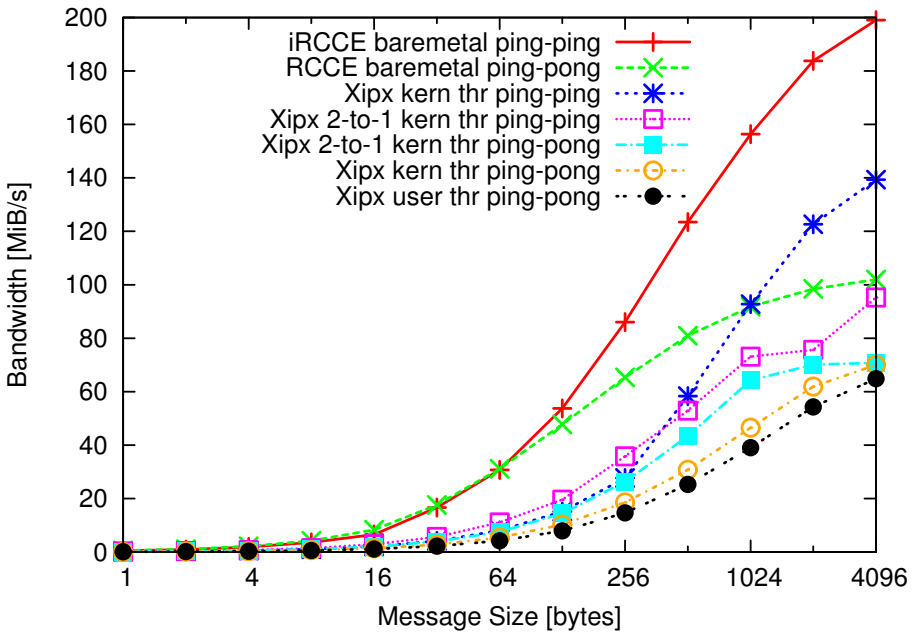

Figure 5.2: Benchmark performance of the asynchronous Xipx MPB device.

is a lower proportion of the growing amount of total work being done per system call. We find the speedup of kernel threads over user threads is between 8% (for 4-KiB messages) and 29% (for 1-byte messages). Xipx kernel threads achieved about 17% of RCCE's ping-pong performance for messages between 1 and 64 bytes long. The relative performance of Xipx increased as messages got longer, up to 69% of RCCE's bandwidth for 4 KiB messages.

One of the reasons the asynchronous Xipx driver does not achieve the same bandwidth as RCCE is because it performs an extra `memcpy()` on the receiving side. Each Xipx MPB device has a buffer pool associated with it, and messages sent to a device immediately get copied off of the MPB and into a buffer from the pool. In addition to the overhead of moving the data, this design can degrade performance

by causing extra cache ejections. However, it facilitates asynchrony in the device and prevents saturation of the MPB space, both of which are important for supporting concurrent parallel applications.

The ping-ping benchmark implemented as a pair of Xipx kernel threads exhibits a 44% to 99% gain in bandwidth over the Xipx kernel thread ping-pong results. This higher bandwidth is achieved by allowing communications between the pair of cores to travel in both directions concurrently. Ping-ping communications in Xipx surpass RCCE's ping-pong bandwidth for message payloads of 1 KiB and higher. A bandwidth improvement of 37% over RCCE is measured for 4-KiB messages. The iRCCE ping-ping implementation outperformed that of Xipx by 43% for 4 KiB messages.

In a two-to-one test, core 0 runs two concurrent benchmarks, each with a different partner core (cores 1 and 2). The measured bandwidth is calculated based on the total data flow in and out of the shared core. Both the RCCE and iRCCE libraries are incapable of running concurrent parallel applications, so these benchmarks were only performed in Xipx. Two-to-one ping-pong showed between 1% and 44% increased bandwidth over its one-to-one counterpart. The two-to-one ping-ping benchmark outperformed its one-to-one counterpart for messages of 256 bytes or shorter, but it displayed a lower bandwidth for larger messages. The lower performance may be due to reduced L1 cache efficiency on the shared core, contention for MPB space, and/or overhead from context switching.

5.1.2 SCC-Specific Device Optimization

An initial implementation of the MPB device neglected to exploit the special caching facilities that the GaussLake core provides for MPB memory (see Section 4.3.2). Furthermore, it used the standard, portable `memcpy()` routine from libxc, Embedded Xinu's subset of the standard C library. The x86 architecture features special instructions for copying arrays of words (`MOVSL`) or arrays of bytes (`MOVSB`) from one memory location to another. However, the compiled libxc `memcpy()` does not take advantage of these instructions.

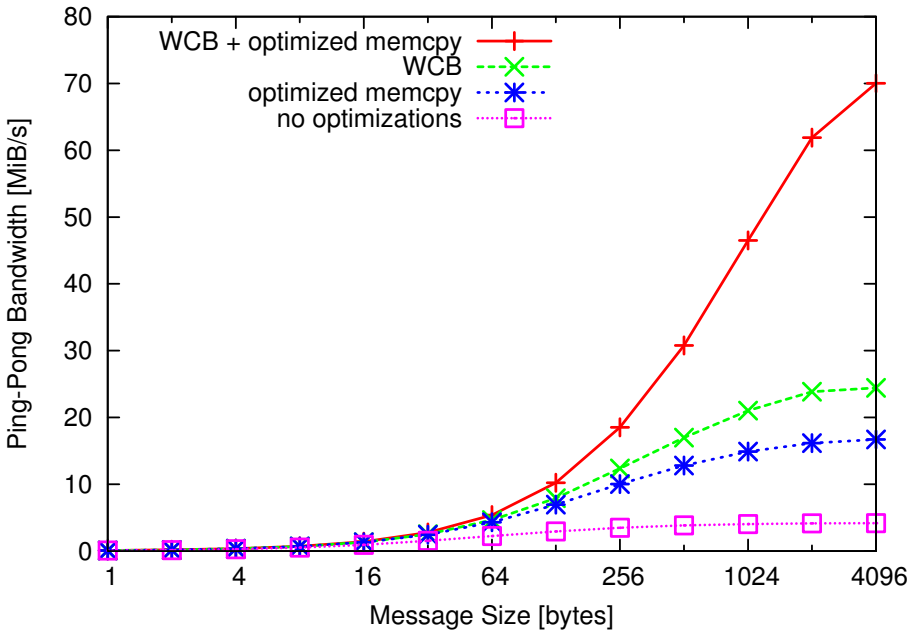

Figure 5.3: The impact of SCC-specific optimizations on message-passing device bandwidth.

As seen in Figure 5.3, this unoptimized implementation running the ping-pong benchmark achieves about 4 MiB/s bandwidth or less with the tested

message lengths. When the libxc `memcpy()` was replaced with an assembly coded routine that takes advantage of the x86 array copying instructions, performance increased up to four fold. The plot marked "WCB" (for write-combining buffer) was produced from reverting to the libxc `memcpy()` and enabling the GaussLake MPB caching features. Finally, both optimizations were combined to obtain our final device implementation, which beats the original by an order of magnitude for large messages. Comparing the fully optimized device to the one that only incorporated an optimized `memcpy()`, we see a four-fold increase in bandwidth for large messages. The impact of the MPB caching hardware is clearly significant for our use of the MPB space.

5.2 Thread Migration

In addition to message passing bandwidth, another important performance measure for a distributed system is thread migration latency. In Section 4.4.4, we identify two methods for thread migration on the SCC, namely message-passing based and LUT swapping. We evaluate the performance of these two techniques by measuring the "freeze time" of a migrating user thread, which is the amount of time that the thread must be suspended during migration. In our experiments, we migrate a thread with code and stack sizes of 470 bytes and 64 KiB, respectively, and a heap size that is varied between 16 KiB and 256 MiB. Results are shown in Figure 5.4.

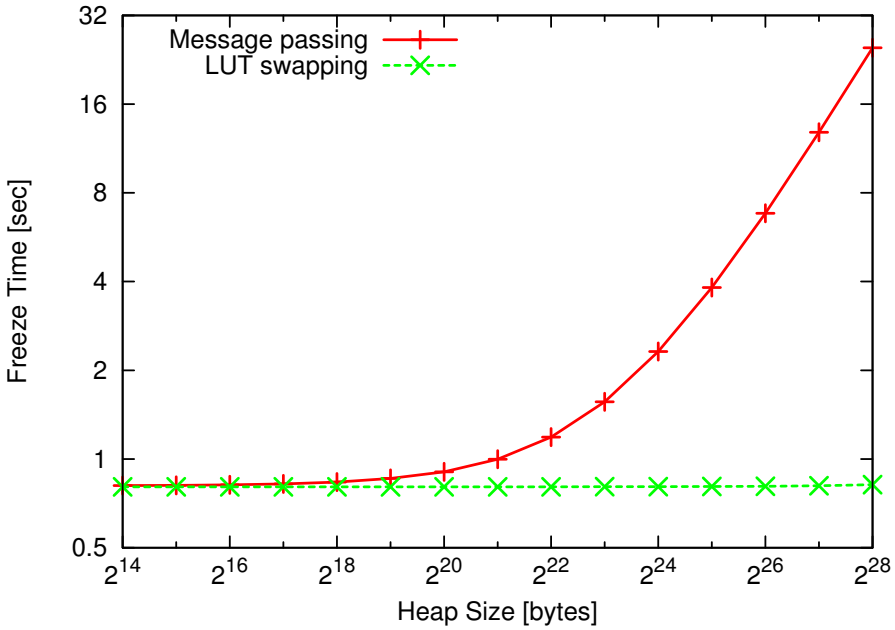

Figure 5.4: Freeze time of migrating Xipx user threads.

We see that no matter the heap size, LUT-based migration exhibits a lower latency than message-passing, although the two are very similar for small threads. By profiling the LUT-swapping method, we find that the procedure is dominated by the creation of a new page table for the incoming thread. This rather expensive routine takes about 804 ms in Xipx, and it is executed every time a user thread is created or received in a migration. Page table creation accounts for about 99.8% of the LUT-based freeze time regardless of the size of the incoming thread. The second most expensive procedure is the L2 cache flush which takes about 1.18 ms, or 0.15% of the freeze time. While our migration routine suspends the migrating thread during the time that the destination core prepares its page table, a more

sophisticated implementation could dramatically lower the freeze time by allowing the thread to continue executing on the source core during this period.

Message-passing based migration suffers from the same baseline freeze time of about 804 ms for page table creation. The bulk of the remaining freeze time is consumed by data transmission via the Xipx MPB device. For a thread with a 16 KiB heap (about 80 KiB including stack and code), data transmission takes about 10.8 ms, which is about 1.3% of the total 815 ms freeze time. With a 256 MiB heap, the largest migrating thread we tested spent 24.07 s, 96.7% of its freeze time, on data transmission.

We were surprised to see LUT-based migrations outperform message-passing for small threads because we expected the L2 flush routine to be a more expensive procedure than it turned out to be. Note, however, that the execution time of this routine varies depending on the state of L2 cache when it is called. Since L2 is configured to be write-back, it may contain a varied number of dirty lines at any time. The greater the number of dirty lines, the more RAM accesses are needed for a flush, and the longer the routine takes. An explanation of the L2 flush routine is given in Appendix B.

CHAPTER 6

Summary and Future Work

This thesis introduced the Xipx operating system, an x86 port of the Embedded Xinu OS tailored to the many-core Intel Single-chip Cloud Computer (SCC). An asynchronous Xipx device that multiplexes the SCC's message passing hardware among many concurrent threads has been presented and has exhibited good performance. Measurements of the device's bandwidth with and without platform-specific optimizations demonstrate the benefits provided by the SCC's unique cache-related hardware features.

Xipx extends Embedded Xinu with a new type of thread that is migratable between separate instances of the OS. A limited framework and protocol for thread migrations has been realized both in a generalized implementation, suitable for any homogeneous distributed environment, and in an implementation that takes advantage of SCC-specific capabilities for drastically reduced data transfer latency.

A significant pragmatic contribution of this thesis is an SCC bare-metal programming framework named BareMichael. As a tool that lowers the barriers to operating-system-free programming on the SCC, the open-source, minimalistic framework has generated interest from several members of the Intel Many-core Applications Research Community.

6.1 Future Work

Though the Xipx MPB device meets the goal of providing fast, scalable message passing channels for concurrent threads, it leaves room for improvement in performance. While our protocol is general enough to support arbitrary networks of message-passing threads, different communication patterns (e.g., collective communications such as broadcast or reduce) have been shown to benefit from more tailored designs. Additionally, our experiments in thread migration suggest that sufficiently large messages might be sent faster by LUT-swapping than by MPB use.

The SCC features dynamic voltage and frequency scaling (DVFS) capabilities. These features provide software control of the chip's speed and power usage, but they are not currently explored in Xipx. With a rapidly growing mobile device market and increased development of large-scale power-hungry clusters and servers, energy-efficient computing has become an increasingly important problem.

Thread migration in Xipx currently lacks robustness. Transparency is compromised, for unexpected migrations may cause threads to fail at establishing communication channels. Providing an efficient and robust migration algorithm is not only a challenging problem in its own right, it also enables exploration of other interesting problems that involve deciding when and where to move threads to achieve gains in performance or energy efficiency. A better migration solution might involve fundamental kernel changes such as the use of globally unique resource

identifiers to keep track of mobile threads. With a very manageable codebase, Xipx is a convenient platform for such system-level modifications.

As it stands, Xipx provides a solid foundation for pursuing research avenues in distributed and many-core computing. However, the kernel would benefit from some optimizations, particularly in dealing with user threads. Currently, Xipx performs very basic virtual memory management. By not leveraging x86 segmentation features, the kernel has to build for each user thread a page table that spans the entire 32-bit memory space. This is inefficient both in space and in time, and it is by far the most expensive operation in user thread creation and LUT-based migration.

Finally, although we believe Xipx features all of the components necessary to support the share-nothing promise-based concurrent threads of the upcoming extensions to the Tamarin virtual machine, experimental validation of this belief has not been possible because those extensions are not yet available to the public. Once the new VM is released, we hope to use it to develop more insight into how an operating system can best host its concurrent threads on a many-core platform for which they are so naturally suited.

BIBLIOGRAPHY

[1] Dennis Brylow, "An experimental laboratory environment for teaching embedded operating systems", in *SIGCSE 2008: Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, New York, NY, USA, 2008, vol. 40, pp. 192–196, ACM.

[2] Dennis Brylow and Bina Ramamurthy, "Nexos: A next generation embedded systems laboratory", *SIGBED Review*, vol. 6, no. 1, Jan. 2009, URL `http://sigbed.seas.upenn.edu/`.

[3] J. Howard, S. Dighe, S. R. Vangal, G. Ruhl, N. Borkar, S. Jain, V. Erraguntla, M. Konow, M. Riepen M. Gries, G.Droege, T. Lund-Larsen, S. Steibl, S. Borkar, V. K. De, and R. Van der Wijngaart, "A 48-core IA-32 processor in 45 nm CMOS using on-die message-passing and DVFS for performance and power scaling", *IEEE Journal of Solid-State Circuits*, vol. 46, no. 1, pp. 173–183, Jan. 2011.

[4] "Tamarin - MDN", Online, Accessed on 28 May 2012. URL `https://developer.mozilla.org/en/Tamarin`.

[5] Jason Williams and Krzysztof Palacz, "Concurrency in Flash runtimes", Online video clip, 2011, Accessed on 31 January 2012. URL `http://tv.adobe.com/watch/max-2011-develop/concurrency-in-flash-runtimes/`.

[6] B. Liskov and L. Shrira, "Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems", in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, New York, NY, USA, 1988, PLDI '88, pp. 260–267, ACM.

[7] Ido Green, *Web Workers: Multithreaded Programs in JavaScript*, O'Reilly, Sebastopol, CA, 2012.

[8] Herman H. Goldstine, *The Computer: From Pascal to von Neumann*, Princeton University Press, Princeton, NJ, 1972.

[9] Jim Handy, *The Cache Memory Book*, Academic Press, Inc., 1250 Sixth Avenue, San Diego, CA 92101-4311, 2nd edition, 1998.

[10] John L. Hennessy and David A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, San Francisco, CA, 2nd edition, 1996.

[11] Intel, *Intel Architecture Software Developer's Manual, Volume 3: System Programming*, 1999.

[12] David A. Patterson and John L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann Publishers, San Francisco, CA, 3rd edition, 2005.

[13] Mostafa Abd-El-Barr and Hesham El-Rewini, *Fundamentals of Computer Organization and Architecture*, Wiley, Hoboken, NJ, 2005.

[14] Lionel Torres, Pascal Benoit, Gilles Sassatelli, Michel Robert, Fabien Clermidy, and Diego Puschini, *An Introduction to Multi-Core System on Chip – Trends and Challenges*, Springer, New York, NY, 2011.

[15] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann Publishers, San Francisco, CA, 1999.

[16] Rakesh Kumar, Timothy G. Mattson, Gilles Pokam, and Rob Van Der Wijngaart, "The case for message passing on many-core chips", in *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*, Michael Hübner and Jürgen Becker, Eds. Springer, New York, 2011.

[17] Charlie Demerjian, "Tilera releases a second 64-core chip", Sept. 2008, Accessed on 5 May 2012. URL `http://www.theinquirer.net/inquirer/news/1006963/tilera-releases-core-chip`.

[18] Jason Miller, James Psota, George Kurian, Nathan Beckmann, Jonathan Eastep, Jifeng Liu, Mark Beals, Jurgen Michel, Lionel Kimerling, and Anant Agarwal, "ATAC: A manycore processor with on-chip optical network", Tech. Rep., Massachusetts Institute of Technology, May 2009.

[19] Avi Silberschatz, Peter Baer Galvin, and Greg Gagne, *Operating Systems Concepts*, Wiley, Hoboken, NJ, 8th edition, 2009.

[20] Scott J. Norton and Mark D. Dipasquale, *Threadtime: The Multithreaded Programming Guide*, Prentice Hall, Upper Saddle River, NJ, 1997.

[21] Thomas W. Doeppner, *Operating Systems in Depth*, Wiley, Hoboken, NJ, 2011.

[22] Thomas Rauber and Gudula Rünger, *Parallel Programming for Multicore and Cluster Systems*, Springer, New York, 2010.

[23] András Vajda, *Programming Many-Core Chips*, Springer, New York, 2011.

[24] Herb Sutter and James Larus, "Software and the concurrency revolution", *Queue*, vol. 3, no. 7, pp. 54–62, Sept. 2005.

[25] Edward A. Lee, "The problem with threads", *Computer*, vol. 39, no. 5, pp. 33–42, 2006.

[26] Intel Corporation, *SCC External Architecture Specification (EAS)*, Nov. 2010, Revision 1.1.

[27] D. Wentzlaff, P. Griffin, H. Hoffmann, Liewei Bao, B. Edwards, C. Ramey, M. Mattina, Chyi-Chang Miao, J.F. Brown, and A. Agarwal, "On-chip interconnection architecture of the tile processor", *Micro, IEEE*, vol. 27, no. 5, pp. 15 –31, Oct. 2007.

[28] Shane Bell, Bruce Edwards, John Amann, Rich Conlin, Kevin Joyce, Vince

Leung, John MacKay, Mike Reif, Liewei Bao, John Brown, Matthew Mattina, Chyi-Chang Miao, Carl Ramey, David Wentzlaff, Walker Anderson, Ethan Berger, Nat Fairbanks, Durlov Khan, Froilan Montenegro, Jay Stickney, and John Zook, "TILE64 processor: A 64-core SoC with mesh interconnect", in *Proceedings of the IEEE International Solid-State Circuits Conference*, Feb. 2008.

[29] George Kurian, Jason Miller, James Psota, Jonathan Eastep, Jifeng Liu, Jurgen Michel, Lionel Kimerling, and Anant Agarwal, "ATAC: A 1000-core cache-coherent processor with on-chip optical network", in *Proceedings of Parallel Architectures and Compilation Techniques*. Sept. 2010, ACM.

[30] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich, "An analysis of Linux scalability to many cores", in *9th USENIX Symposium on Operating System Design and Implementation, OSDI10*, Oct. 2010.

[31] Intel Corporation, *MultiProcessor Specification*, May 1997.

[32] Jan-Arne Sobania, Peter Tröget, and Andreas Polze, "Towards symmetric multi-processing support for operating systems on the SCC", in *Proceedings of the 4th Many-core Applications Research Community Symposium*. Dec. 2011, Hasso Plattner Institute at the University of Potsdam.

[33] "IBM systems: Virtualization", White Paper, IBM, Dec. 2005.

[34] Stefan Lankes, Pablo Reble, Carsten Clauss, and Oliver Sinnen, "The path to MetalSVM: Shared virtual memory for the SCC", in *Proceedings of the 4th Many-core Applications Research Community Symposium*. Dec. 2011, Hasso Plattner Institute at the University of Potsdam.

[35] Pablo Reble, Stefan Lankes, Carsten Clauss, and Thomas Bemmerl, "A fast inter-kernel communication and synchronization layer for MetalSVM", in *Proceedings of the 3rd Many-core Applications Research Community Symposium*. July 2011, KIT Scientific Publishing.

[36] Stefan Lankes, Pablo Reble, Carsten Clauss, and Oliver Sinnen, "Revisiting shared virtual memory systems for non-coherent memory-coupled cores", in *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM 2012)*, Feb. 2012.

[37] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania, "The multikernel: A new OS architecture for scalable multicore systems", in *Proceedings of the 22nd SOSP*, Oct. 2009.

[38] Simon Peter, Adrian Schüpbach, Dominik Menzi, and Timothy Roscoe, "Early experience with the Barrelfish OS and the Single-Chip Cloud Computer", in *Proceedings of the 3rd Many-core Applications Research Community Symposium*. July 2011, KIT Scientific Publishing.

[39] David Wentzlaff, Charles Gruenwald III, Nathan Beckmann, Kevin Modzelewski,

Adam Belay, Lamia Youseff, Jason Miller, and Anant Agarwal, "A unified operating system for clouds and manycore: fos", in *Proceedings of the 1st Workshop on Computer Architecture and Operating Systems co-design (CAOS)*, Jan. 2010.

[40] Adam Belay, "Message passing in a factored OS", Master's thesis, MIT, 2011.

[41] Timothy G. Mattson, Michael Riepen, Thomas Lehnig, Paul Brett, Werner Haas, Patrick Kennedy, Jason Howard, Sriram Vangal, Nitin Borkar, Greg Ruhl, and Saurabh Dighe, "The 48-core SCC processor: The programmer's view", in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Washington, DC, USA, 2010, SC '10, pp. 1–11, IEEE Computer Society.

[42] Jan-Arne Sobania, Peter Tröger, and Andreas Polze, "Linux operating system support for the SCC platform - an analysis", in *Proceedings of the 3rd Many-core Applications Research Community Symposium*. July 2011, KIT Scientific Publishing.

[43] Michael J. Flynn, "Very high-speed computing systems", *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1901–1909, 1966.

[44] Vijay Vasudevan, David G. Andersen, and Michael Kaminsky, "The case for VOS: The Vector Operating System", in *Proceedings of HotOS XIII*, May 2011.

[45] Per Brinch Hansen, "The nucleus of a multiprogramming system", *Communications of the ACM*, vol. 13, no. 4, pp. 238–241, Apr. 1970.

[46] J E White, "A high level framework for network-based resource sharing", in *Proceedings of the National Computer Conference*. June 1976, AFIPS Press.

[47] Marc Snir, Steve W. Otto, David W. Walker, Jack Dongarra, and Steven Huss-Lederman, *MPI: The Complete Reference*, MIT Press, Cambridge, MA, USA, 1995.

[48] Stephen F. Siegel and Ganesh Gopalakrishnan, "Formal analysis of message passing", in *Verification, Model Checking, and Abstract Interpretation*, Ranjit Jhala and David Schmidt, Eds., vol. 6538 of *Lecture Notes in Computer Science*, pp. 2–18. Springer Berlin / Heidelbert, 2011.

[49] Isaías Comprés Ureña, Michael Riepen, and Michael Konow, "RCKMPI – lightweight MPI implementation for Intel's Single-chip Cloud Computer (SCC)", in *Recent Advances in the Message Passing Interface*, Yiannis Cotronis, Anthony Danalis, Dimitrios Nikolopoulos, and Jack Dongarra, Eds., vol. 6960 of *Lecture Notes in Computer Science*, pp. 208–217. Springer Berlin / Heidelberg, 2011.

[50] William Gropp, "MPICH2: A new start for MPI implementations", in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Dieter Kranzlmúller, Jens Volkert, Peter Kacsuk, and Jack Dongarra, Eds., vol. 2474 of *Lecture Notes in Computer Science*, pp. 37–42. Springer Berlin / Heidelberg, 2002.

[51] Steffen Christgau, Bettina Schnor, and Simon Kiertscher, "The benefit of topology-awareness of MPI applications on the SCC", in *Proceedings of the 3rd Many-core Applications Research Community Symposium*. July 2011, KIT Scientific Publishing.

[52] Isaías A. Comprés Ureña and Michael Gerndt, "Improved RCKMPI's SCCMPB channel: Scaling and dynamic process support", in *Proceedings of the 4th Many-core Applications Research Community Symposium*. Dec. 2011, Hasso Plattner Institute at the University of Potsdam.

[53] Boris Bierbaum, Carsten Clauss, Rainer Finocchiaro, Martin Pöppe, Silke Schuch, and Joachim Worringen, *MP-MPICH – User Documentation and Technical Notes*, Chair for Operating Systems, RWTH Aachen University, 2009.

[54] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl, "Evaluation and improvements of programming models for the Intel SCC many-core processor", in *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, July 2011, pp. 525–532.

[55] Carsten Clauss, Stefan Lankes, Pablo Reble, and Thomas Bemmerl, "Recent advances and future prospects in iRCCE and SCC-MPICH", Poster at the 3rd Many-core Applications Research Community Symposium, July 2011.

[56] Tim Mattson and Rob van der Wijngaart, "RCCE: A small library for many-core communication", Jan. 2011, Software Version 2.0-release.

[57] Carsten Clauss, Stefan Lankes, Thomas Bemmerl, Jacek Galowicz, and Simon Pickartz, *iRCCE: A Non-blocking Communication Extention to the RCCE Communication Library for the Intel Single-Chip Cloud Computer – User Manual*, Chair for Operating Systems, RWTH Aachen University, June 2011.

[58] Aparna Chandramowlishwaran and Richard Vuduc, "Performance modeling on SCC's on-chip interconnect", Poster at the 2nd Many-core Applications Research Community Symposium, Mar. 2011.

[59] Randolf Rotta, "On efficient message passing on the Intel SCC", in *Proceedings of the 3rd Many-core Applications Research Community Symposium*. July 2011, KIT Scientific Publishing.

[60] M.J. Litzkow, M. Livny, and M.W. Mutka, "Condor – a hunter of idle workstations", in *Proceedings of the 8th International Conference on Distributed Computing Systems*, June 1988, pp. 104–111.

[61] Dario Simone, "Power management in a manycore operating system", Master's thesis, Swiss Federal Institute of Technology Zurich, 2009.

[62] Mohamed Gomaa, Michael D. Powell, and T. N. Vijaykumar, "Heat-and-run: Leveraging SMT and CMP to manage power density through the operating system", in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 2004, ASPLOS-XI, pp. 260–270, ACM.

[63] K. Thitikamol and P. Keleher, "Thread migration and communication minimization in DSM systems", *Proceedings of the IEEE*, vol. 87, no. 3, pp. 487–497, Mar. 1999.

[64] Sayantan Chakravorty, Celso Mendes, and Laxmikant Kalé, "Proactive fault tolerance in MPI applications via task migration", in *High Performance Computing - HiPC 2006*, Yves Robert, Manish Parashar, Ramamurthy Badrinath, and Viktor Prasanna, Eds., vol. 4297 of *Lecture Notes in Computer Science*, pp. 485–496. Springer Berlin / Heidelberg, 2006.

[65] Gengbin Zheng, L.V. Kale, and O.S. Lawlor, "Multiple flows of control in migratable parallel programs", in *International Conference on Parallel Processing Workshops, ICPP 2006 Workshops*. 2006, IEEE.

[66] Dejan S. Milojíčić, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou, "Process migration", *ACM Comput. Surv.*, vol. 32, no. 3, pp. 241–299, Sept. 2000.

[67] Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton, "Preemptable remote execution facilities for the V-system", in *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, New York, NY, USA, 1985, SOSP '85, pp. 2–12, ACM.

[68] E. Zayas, "Attacking the process migration bottleneck", in *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, New York, NY, USA, 1987, SOSP '87, pp. 13–24, ACM.

[69] Fred Douglis and John Ousterhout, "Transparent process migration: Design alternatives and the Sprite implementation", *Software: Practice and Experience*, vol. 21, no. 8, pp. 757–785, 1991.

[70] Amnon Barak, Oren Laden, and Yuval Yarom, "NOW MOSIX and its preemptive process migration scheme", *Bulletin of the IEEE Technical Committee on Operating Systems and Application Environments*, vol. 7, no. 2, pp. 5–11, 1995.

[71] E. Steketee, Wei Ping Zhu, and P. Moseley, "Implementation of process migration in Amoeba", in *Proceedings of the 14th International Conference on Distributed Computing Systems*, June 1994, pp. 194–201.

[72] M. Frans Kaashoek, Robbert van Renesse, Hans van Staveren, and Andrew S. Tanenbaum, "FLIP: An internetwork protocol for supporting distributed systems", *ACM Trans. Comput. Syst.*, vol. 11, no. 1, pp. 73–106, Feb. 1993.

[73] Gerald J. Popek and Bruce J. Walker, *The LOCUS Distributed System Architecture*, MIT Press, Boston, MA, 1985.

[74] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield, "Live migration of virtual machines", in *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, Berkeley, CA, USA, 2005, NSDI'05, pp. 273–286, USENIX Association.

[75] Michael Nelson, Beng-Hong Lim, and Greg Hutchins, "Fast transparent migration for virtual machines", in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, Berkeley, CA, USA, 2005, ATEC '05, pp. 25–25, USENIX Association.

[76] Wei Huang, Qi Gao, Jiuxing Liu, and D.K. Panda, "High performance virtual machine migration with RDMA over modern interconnects", in *Cluster Computing, 2007 IEEE International Conference on*, Sept. 2007, pp. 11–20.

[77] Robert Bradford, Evangelos Kotsovinos, Anja Feldmann, and Harald Schiöberg, "Live wide-area migration of virtual machines including local persistent state", in *Proceedings of the 3rd International Conference on Virtual Execution Environments*, New York, NY, USA, 2007, VEE '07, pp. 169–179, ACM.

[78] R.D. Blumofe and C.E. Leiserson, "Scheduling multithreaded computations by work stealing", in *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, Nov. 1994, pp. 356–368.

[79] D. Majeti, "Lightweight dynamic task creation and scheduling on the Intel Single Chip Cloud (SCC) processor", in *Proceedings of the Fourth Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software*, 2011, pp. 35–42.

[80] Andreas Prell and Thomas Rauber, "Task parallelism on the SCC", Poster at the 3rd Many-core Applications Research Community Symposium, July 2011.

[81] Georgios Varisteas, Mats Brorsson, and Karl-Filip Faxèn, "Resource management for task-based parallel programs over a multi-kernel", in *Proceedings of RESoLVE 2012*. Mar. 2012, ACM.

[82] S.A. Brandt, S. Banachowski, Caixue Lin, and T. Bisson, "Dynamic integrated scheduling of hard real-time, soft real-time, and non-real-time processes", in *Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE*, Dec. 2003, pp. 396–407.

[83] Douglas E. Comer, *Operating System Design: The XINU Approach*, Prentice Hall, 1984.

[84] Zachary D. Lund, "A VoIP implementation on an embedded platform", Master's thesis, Marquette University, 2010.

[85] Kyle Persohn and Dennis Brylow, "Interactive real-time embedded systems education infused with applied internet telephony", in *Computer Software and Applications Conference (COMPSAC), 2011 IEEE 35th Annual*, July 2011, pp. 199–204.

[86] Michael J. Schultz, "Using software transactional memory in interrupt-driven systems", Master's thesis, Marquette University, 2009.

[87] "Many-core applications research community", An online community of users of the SCC processor. `http://communities.intel.com/community/marc/`.

[88] "Bug 46: Bypass causes data corruption in the MPB", Many-core Applications Research Community (MARC) Bugzilla, Aug. 2010, Accessed on 28 May 2012. URL `http://marcbug.scc-dc.com/bugzilla3/show_bug.cgi?id=46`.

[89] Michiel W. van Tol, Roy Bakker, Merijn Verstraaten, Clemens Grelck, and Chris R. Jesshope, "Efficient memory copy operations on the 48-core Intel SCC processor", in *Proceedings of the 3rd Many-core Applications Research Community Symposium*. July 2011, KIT Scientific Publishing.

[90] "Is there any documentation for running applications on baremetal", Intel MARC forums, July 2011, Accessed on 15 March 2012. URL `http://communitites.intel.com/thread/23765/`.

[91] Michael W. Ziwisky and Dennis W. Brylow, "BareMichael: A minimalistic bare-metal framework for the Intel SCC", in *Proceedings of the* 6^{th} *Many-core Applications Research Community (MARC) Symposium*, Éric Noulard and Simon Vernhes, Eds. July 2012, ONERA, The French Aerospace Lab, `http://sites.onera.fr/marconera2012`.

[92] "Bug 195: Cache flushing with shared memory unreliable", Many-core Applications Research Community (MARC) Bugzilla, Apr. 2011, Accessed on 28 May 2012. URL `http://marcbug.scc-dc.com/bugzilla3/show_bug.cgi?id=195`.

APPENDIX A

Implementation Details:
Virtual Memory, Privilege Levels, and System Calls

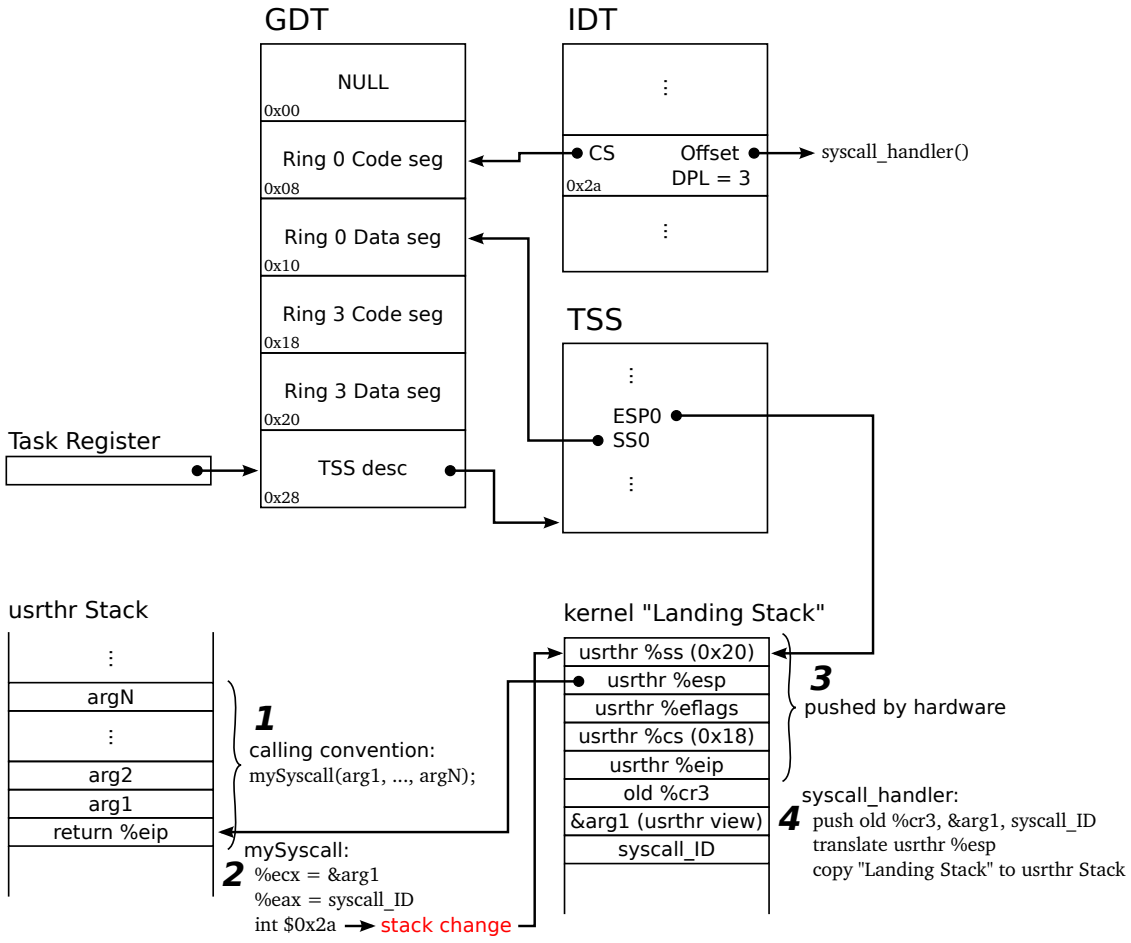

Figure A.1: Xipx CPU configuration and system call handling on an x86 architecture.

Figure A.1 illustrates some details of the data structures that Xipx creates to configure the CPU to use multiple privilege levels on an x86 system. It also depicts the beginning of the flow of events that occurs when a user thread executes a

system call. The purpose of this Appendix is not to describe how the x86 architecture works, but to describe how Xipx uses the architecture. For details of the former, see [11].

Xipx uses a static global descriptor table (GDT) with a minimal setup to support two privilege levels. The GDT includes code and data segment descriptors configured to span the entire 4 GiB memory space for both Ring 0 and Ring 3. It also includes a task segment selector (TSS) descriptor, which is a necessary for privilege level management.

Task segment selectors generally are used to define and configure hardware tasks on an x86 processor, but hardware task management is not a common architecture feature outside of x86. For reasons that include maximizing the portability of kernel code, Xipx performs task management in software. However, as discussed below, at least one TSS is required to use x86 privilege level features. Therefore, Xipx creates a single TSS in memory and references it by a TSS descriptor in the GDT. The Task Register indicates the current TSS – the *only* TSS for Xipx – by pointing to the TSS descriptor in the GDT.

The interrupt descriptor table (IDT) created by Xipx holds a descriptor (at index 0x2a) that points to the function `syscall_handler()` and to the Ring 0 code segment descriptor, with a descriptor privilege level (DPL) of 3. The DPL defines the highest current privilege level (CPL) at which the interrupt may be triggered. (Triggering the interrupt when the CPL is greater than the DPL results in a general

protection exception.) Therefore, the DPL of 3 means the interrupt handler may execute from any CPL, and the CS field pointing to the Ring 0 descriptor means that the CPL changes to (or remains at) 0 upon handler execution.

Xipx needs to define a TSS because when an x86 interrupt causes a privilege level change, a mandatory stack change takes place. The hardware automatically changes the stack segment and stack pointer to the values defined by the SS# and ESP# fields, respectively, of the current TSS, where # is the privilege level to which the CPU has changed. Because all interrupt descriptors in Xipx point to the Ring 0 code segment descriptor, interrupt-induced stack changes only occur when going from Ring 3 to Ring 0. Therefore, there is no need to define the TSS SS/ESP fields for Rings 1–3. The SS0 field points to the Ring 0 data segment descriptor, and the ESP0 field points to a space we refer to as the kernel *landing stack*.

The x86 architecture mandates a stack change when changing privilege levels so that the kernel may use a controlled stack, rather than the stack of the previously running thread, while handling the interrupt. Some of the reasons this may be desirable are to avoid overflowing a thread stack that happens to be nearly full, and to avoid exposing to the thread any data that the OS places on the stack lest that data be exploitable. However, using a single landing stack for system calls from all threads is dangerous. Some system calls in Xipx may reschedule before returning, which could allow another user thread to run and execute another system call. If ESP0 is unchanged before this time, the new system call will corrupt the

previous thread by overwriting the data it stored on the landing stack. One solution is to allocate a new stack and set ESP0 to point to it whenever a system call yields, but such dynamic memory management would introduce a varying amount of processing overhead to system calls.

Xipx copies everything from the landing stack onto the stack of the current thread immediately and then uses the thread stack to process the system call. Although this solution ignores the concerns of using a thread stack for kernel work, it would be a trivial matter to preallocate a stack per user thread in kernel space and copy from the landing stack to that location instead.

We will now discuss in detail the actions that take place when a user thread performs a system call. In these paragraphs, we refer to Steps 1 through 4, which correspond to the enumerated details in Figure A.1.

Step 1 When a user thread invokes a system call, arguments get pushed onto the stack in right-to-left order as is expected by the `_cdecl` calling convention Xipx uses. The x86 `call` instruction then pushes the return address before jumping to the called function.

Step 2 Several system calls are defined and linked into each user thread, and each one is a short and simple function. Although system call declarations list the

parameters required by the kernel function that will ultimately execute, the defined functions in user space simply:

1. load register ECX with the address of the first argument pushed,
2. load register EAX with a unique system call identifier, and
3. trigger a software interrupt for vector 0x2a.

As described above, this vector is configured to switch to CPL 0 (thereby changing SS/ESP to the kernel landing stack) and execute the `syscall_handler()` function in the kernel.

Step 3 With a stack change, the x86 hardware pushes the following five items onto the landing stack:

1. the old stack segment selector,
2. the old stack pointer,
3. the old flags register,
4. the old code segment selector, and
5. the old instruction pointer.

These values are needed later on to return, by the `iret` instruction, from the interrupt handler in Ring 0 to the user space system call function in Ring 3. When calling `iret`, the change from Ring 0 to Ring 3 does not involve the SS3 and ESP3

entries in the current TSS – rather, it pops the new SS and ESP off the stack. With the landing stack primed for an `iret`, control is given to the `syscall_handler()` function.

Step 4 The `syscall_handler()` function begins by pushing the CR3, ECX, and EAX registers. The CR3 register, also known as the page-directory base register, holds the base physical address of the current page-directory and therefore controls the virtual memory mapping of the processor. As stated in Step 2 above, the ECX and EAX registers contain a pointer to the first argument of the system call and the identifier of the system call to execute, respectively. With these three values on the stack, `syscall_handler()` switches to the kernel's flat-mapped memory space by loading CR3 with the kernel page-directory base address, locates the physical address of the user thread stack by reading the pushed ESP off of the landing stack and translating it, and then copies all of the contents from the landing stack to the user thread stack.

Finally, the system call identifier is used to determine how many arguments to pass to the actual system call function in the kernel, and the location of those arguments is determined by translating the argument pointer. After the system call executes and returns, the old stack pointer that was pushed by the hardware in step 3 is saved to a register, then the user thread's view of memory is restored by setting CR3. Then the old stack pointer is used to set ESP to point to the base of the five

values that were pushed by the hardware in step 3 (on the user thread stack now rather than the kernel stack), and finally an `iret` instruction brings the CPU back to Ring 3 while returning control to the user thread.

Note that system call handling is a special case of interrupt handling in general, and the procedure followed for interrupt handling is similar to that just described. While system calls only execute when they are called from within user threads, interrupts may occur at any time. For this reason, an interrupt handler first saves all general purpose registers, then determines whether a stack change occurred and whether the kernel's (flat) view of memory is in place. System calls always come with a stack change, meaning values must be copied from the landing stack to the user thread stack. They also always occur from a non-flat view of memory, meaning some pointer translation is needed. Either or both of these things may be untrue when an interrupt occurs, so there is some conditional control flow that decides whether stack copies and pointer translations are done. The code listing for interrupt handling is given in Figure A.2, and the `_ut_to_k_view()` macro that it references is given in Figure A.3.

```
#define EXCEPTION(num, ecode)                                          \
        .globl _Xint##num;                                             \
_Xint##num:                                                            \
    pushal;                                                            \
                                                                       \
    movl    %cr3, %eax;                                                \
    pushl   %eax;                                                      \
                                                                       \
    _ut_to_k_view(9+ecode);                                            \
                                                                       \
    movl    %esp, %ecx;                                                \
    cmp     $0x0, %ebx;                                                \
    je      1f;                                                        \
           /* change pushed %esp to what it would have been   */       \
           /* w/o stk change (and prior to changing to cr3=0) */       \
    movl    ((12+ecode)*4)(%esp), %ecx; /*ecx = old esp, ut view*/     \
    subl    $(0x14 + 4*ecode), %ecx;  /*move to after intr stuff*/     \
    movl    %ecx, 0x10(%esp);   /* replace in stack */                 \
    subl    $0x24, %ecx;        /* move to after pushal & cr3 */       \
                                                                       \
    /* save physical %esp in thrtab (needed for thr migration */       \
 1: movl    thrcurrent, %eax;                                          \
    movl    $THRENTSIZE, %edx;                                         \
    mull    %edx;                                                      \
    movl    $thrtab, %edx;                                             \
    movl    %esp, STKDIVOFFSET(%edx, %eax, 1);                         \
                                                                       \
    pushl   %ebx;           /* change_stk */                           \
    pushl   %ecx;           /* stk_ptr */                              \
    pushl   $num;           /* exc_num */                              \
    call    dispatch;                                                  \
    addl    $3*4, %esp;                                                \
                                                                       \
    movl    0x10(%esp), %ecx; /* ecx = user esp before pushal */       \
    subl    $0x20, %ecx;      /* ecx = user esp after pushal */        \
    popl    %edx;                                                      \
    movl    %edx, %cr3;         /* to usrthr view of mem */            \
    movl    %ecx, %esp;         /* restore esp */                      \
                                                                       \
    popal;                                                             \
    iret;

/* Create the individual exception handlers */
EXCEPTION(0x00, 0)
EXCEPTION(0x01, 0)
EXCEPTION(0x02, 0)
/* ... etc. */
```

Figure A.2: Code listing for the set of default interrupt handlers.

```
#define _ut_to_k_view(nelem)                                              \
    movl    ((nelem+1)*4)(%esp), %edx; /* edx = old CS */                 \
    andl    $~0xffff0007, %edx; /* forget about RPL and hi bits*/          \
    mov     %cs, %cx;           /* cx = curr CS */                         \
    andl    $~0xffff0007, %ecx;                                            \
    cmp     %ecx, %edx;         /* did we change stacks? */                \
    je      1f;                 /* if not, jump */                         \
    movl    $0x1, %ebx;  /* from now on, ebx = 1 if we changed */          \
    movl    ((nelem+3)*4)(%esp), %ecx;  /* user esp to ecx */              \
    jmp     2f;                                                            \
 1: movl    $0x0, %ebx;  /* ebx = 0 if we didn't change stacks */          \
    movl    %esp, %ecx;  /* current esp in ecx */                          \
                                                                           \
 2: cmp     $0x0, %eax;  /* if cr3 is already 0x00000000... */             \
    je      3f;          /* ...then skip the translation. */               \
    xorl    %edx, %edx;                                                    \
    movl    %edx, %cr3;         /* now in kernel view of mem */            \
    movl    %ecx, %edx;                                                    \
    shrl    $0x16, %edx;        /* edx = pdindex of user esp */            \
    shll    $0x2, %edx;                                                    \
    addl    %edx, %eax;         /* eax = &(pdir[pdindex]) */               \
    movl    (%eax), %eax;       /* eax = pdir[pdindex] */                  \
    andl    $0xfffff000, %eax;  /* eax = &ptab */                          \
    movl    %ecx, %edx;                                                    \
    shrl    $0xc, %edx;                                                    \
    andl    $0x3ff, %edx;       /* edx = ptindex of user esp */            \
    shll    $0x2, %edx;                                                    \
    addl    %edx, %eax;         /* eax = &(ptab[ptindex]) */               \
    movl    (%eax), %eax;       /* eax = ptab[ptindex] */                  \
    andl    $0xfffff000, %eax;                                             \
    movl    %ecx, %edx;                                                    \
    andl    $0xfff, %edx;                                                  \
    addl    %edx, %eax;         /* eax = phy addr of user esp */           \
    movl    %esp, %esi;  /* going to be copying from kern stk */           \
    movl    %eax, %esp;         /* now we're on user stack! */             \
 3: cmp     $0x0, %ebx;                                                    \
    je      4f;  /* if no stack change, then nothing to copy */            \
                 /* otherwise, copy elements to usrthr stack */            \
    subl    $((nelem+5)*4), %esp; /* make room for copies */               \
    movl    $(nelem+5), %ecx;     /* copy n+5 doublewords... */            \
    movl    %esp, %edi;           /* ...to usrthr stack. */                \
    cld;                                                                   \
    rep;                                                                   \
    movsl;                        /* do the copying! */                    \
 4: \
```

Figure A.3: Code listing for a macro that an interrupt handler uses to ensure the CPU is in a flat-mapped view of memory and to undo the effects of an interrupt-induced stack change.

APPENDIX B

SCC L2 Cache Flush Routine

This appendix explains in detail the operation of L2 cache on the Intel SCC. It also describes a software routine to flush the contents of L2 cache to RAM. Much credit for the concepts that guided the development of our `flushL2()` routine is due to the Many-core Applications Research Community (MARC) members – particularly Michiel van Tol and Werner Haas – whose contributions on the topic are archived in [92].

Each core of the SCC has associated with it a 256 KiB, 4-way set associative L2 cache [26] that uses a pseudo-least-recently-used (PLRU) replacement policy. It is useful in a non-cache-coherent system to be able to explicitly flush the entire L2 cache. However, this cache is off-core and therefore cannot be controlled directly by the CPU. The x86 instruction set includes the `WBINVD` instructions to write back and invalidate all L1 cache lines, but there is no analogous instructions to control L2 cache. Therefore, we must achieve this end through more roundabout means. Using knowledge of how the L2 cache operates, we have built a routine that flushes the entire L2 cache by performing a sequence of reads from memory that evict and replace each line in succession.

B.1 The PLRU Replacement Policy

To describe our L2 flush routine, we must first describe how the L2 cache controller decides which of the four ways is replaced when a new line is loaded. The PLRU policy makes this decision based on its current state, which is stored by three bits for each set. We refer to a set's PLRU state as `t|lr` where `t`, `l`, and `r` are the values of the top, left, and right nodes of the corresponding binary tree. In our discussion, an `x` in the state means it does not matter what that bit's value is. The binary decision tree, showing which way to replace based on the current PLRU state, is depicted in Figure B.1. There is no significance to the numbers we use to distinguish the ways – they are numbered arbitrarily.

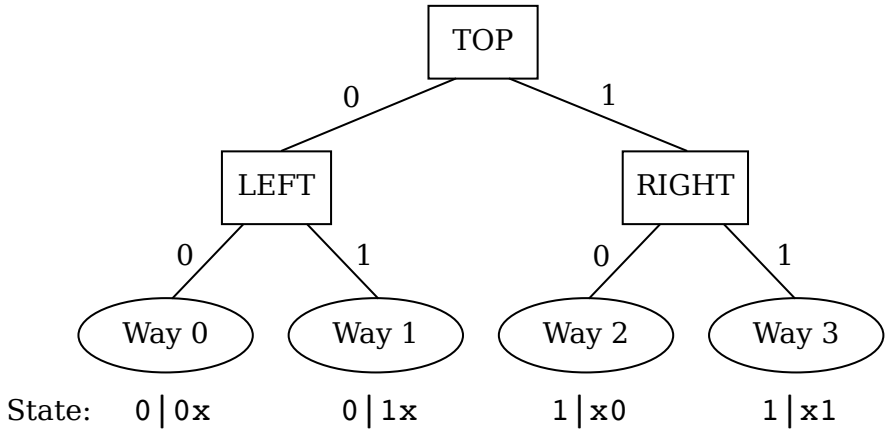

Figure B.1: Binary decision tree for a PLRU cache replacement policy (after [92]).

Bit `l` always indicates the least-recently used way of the two in the "left half" (ways 0 and 1 in Figure B.1). Similarly, bit `r` indicates the least-recently used

way of the two in the "right half" (ways 2 and 3). Bit `t` indicates the least-recently used half, i.e., the *most*-recently used way is *not* in the half to which `t` points. Knowing these facts, one is able to derive the rules for transitioning between PLRU states based on cache hits and cache misses that cause replacements. A state transition diagram is given in Figure B.2.

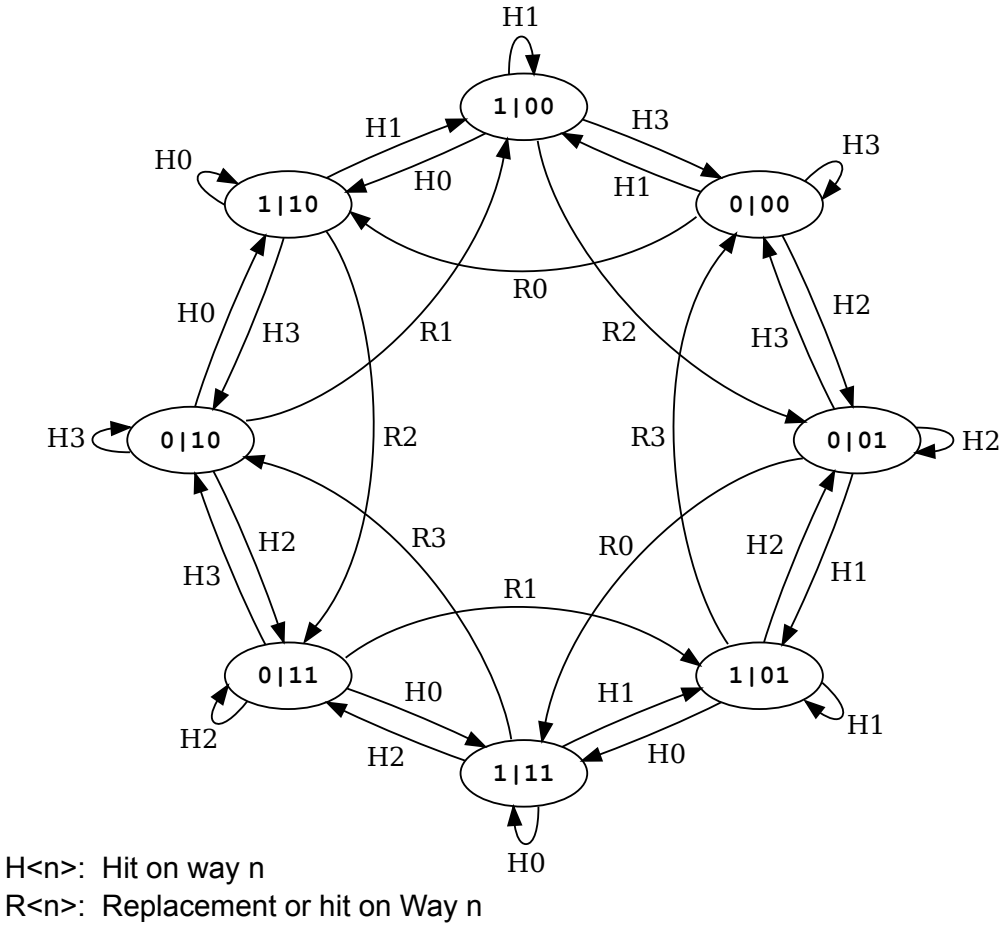

Figure B.2: State machine for a PLRU cache (after [92]).

B.2 Analysis of the L2 Cache Behavior

Based on the discussion in [92], our strategy for flushing the L2 cache involves issuing a series of reads of "dummy data" that fill up every line and, in the process, flush to RAM all dirty lines that were cached before the procedure began. We allocate two 256-KiB sections of address space as dummy data, each used solely for L2 flushing. If we can fill the entire L2 cache with one of these sections, we will have successfully flushed all other data back to RAM. We alternate which section is used each time the `flushL2()` routine is called. This way, we can guarantee that none of the dummy data we are using for the flush is already cached – i.e., every read of dummy data will be a cache miss. If we issue four consecutive reads of dummy data from the same set, that entire set will be evicted. An analysis in support of this claim follows.

Consider addresses A, B, C, and D as the four addresses in our dummy data section that each map to the same set in L2. We start with unknown data, denoted by a `*`, in each of the set's four ways. Without loss in generality, we may assume the initial state of the set is:

L2 set	PLRU bits
****	0|00

We then read our four dummy data addresses sequentially, the effect of which is illustrated below.

step	L2 set	PLRU bits	Action
0	`****`	`0|00`	read A
1	`A***`	`1|10`	read B
2	`A*B*`	`0|11`	read C
3	`ACB*`	`1|01`	read D
4	`ACBD`	`0|00`	*RESULT: set flushed*

These four sequential reads, under the assumption that none of the dummy data is already cached, do in fact replace the entire set in L2. Therefore, our procedure consists of repeatedly issuing groups of four reads, one group per set, until we have flushed every L2 set back to RAM. However, prior to beginning this series of reads, a `WBINVD` instruction must be issued. The justification for this requirement is provided in the following section.

B.3 Effect of L1 Write-Backs

Each SCC core has an on-die L1 data cache that is 16 KiB in size and 4-way set associative. The L1 cache also uses a PLRU policy, and its state is separate from that of L2. Our analysis thus far has neglected interactions between the L1 and L2 cache. However, consider the following scenario. We observe a particular situation in which some stale data that we want flushed to RAM exists in both L1 and L2 cache. The stale data is represented below as `#`. From an initial L1 and L2 state, we perform a series of four reads as follows.

step	L1 set	L1 PLRU	L2 set	L2 PLRU	Action
0	`*#**`	`1|11`	`*#**`	`0|00`	read A
1	`*#*A`	`0|10`	`A#**`	`1|10`	read B
2	`*B*A`	`1|00`	`A#B*`	`0|11`	*L1: write-back* `#`
3	`*B*A`	`1|00`	`A#B*`	`1|01`	read C
4	`*BCA`	`0|01`	`A#BC`	`0|00`	read D
5	`DBCA`	`1|11`	`D#BC`	`1|10`	*RESULT: stale data* `#` *not flushed*

At step 1, we read B from our dummy data. Since B is not present in either L1 or L2, it first gets copied to L2, which causes the L2 PLRU bits to update. Data B then gets copied into L1, which causes an eviction of the stale data, and the stale data gets written back to L2 at step 2. The write-back changes the L2 PLRU bits in such a way that the subsequent reads of C and D fail to flush the stale data from L2.

We see that this problem occurs only when L1 contains a copy of the stale data that we wish to eject from L2 (and the states of the two caches have a certain unfortunate relationship). However, all cache lines in L1 always can be found also in L2, so this vulnerability is always present if L1 contains stale data. Therefore, to prevent the unintended changing of the L2 PLRU bits, we must ensure L1 does not contain any stale data before we perform our dummy data reads. This is done by issuing the `WBINVD` instruction just prior to beginning the sequence of reads.

B.4 The Routine

The assembly language source of our `flushL2()` routine is presented in Figure B.3. We refer to the dummy data locations as "flush areas," and we set the base of the first flush area to the address just after the end of the local MPB

memory space. Each core's local MPB space is mapped to a single LUT entry, which covers 16 MiB of address space. However, the MPB itself is only 16 KiB long. It turns out that reading an address in this segment that is beyond the initial 16 KiB is a valid operation. The address space of the segment wraps after every 16 KiB, so the requested read address modulo 16384 gives the MPB location from which the data is read. However, to the core issuing the read, no address manipulation is observed, so even if the data at the truncated MPB address is present in L2, the read still causes a cache miss. Since reads from the local MPB space are much faster than reads from RAM, we can use these upper MPB address spaces as flush areas to achieve high performance.

```
#define L2_CACHE_SIZE  (256*1024)
#define L2_LINE_SIZE   32
#define L2_WAYS        4
#define L2_WAY_SIZE    (L2_CACHE_SIZE / L2_WAYS)
#define L2_SETS        (L2_WAY_SIZE / L2_LINE_SIZE)

#define BASE_FLUSH_AREA (0xd8000000 + 0x4000) /*just after MPB_OWN*/

    .globl flushL2
    .extern disable
    .extern restore

    .align 4
flusharea:
    .long BASE_FLUSH_AREA

flushL2:
    push    %ebp
    movl    %esp, %ebp
    subl    $0x4, %esp /* make room for irqmask */

    /* flip-flop between two flush areas */
    cmpl    $BASE_FLUSH_AREA, flusharea
    jne     1f
    movl    $(BASE_FLUSH_AREA + L2_CACHE_SIZE), flusharea
    jmp     2f
1:  movl    $BASE_FLUSH_AREA, flusharea
2:  movl    flusharea, %edx /* edx is the flusharea to use */

    call    disable
    movl    %eax, -0x4(%ebp) /* store irqmask */

    wbinvd
    /* begin loop over sets */
    xorl    %ecx, %ecx                  /* i = 0 */
    /* fill all four ways of the set */
3:  movl    (0x0*L2_WAY_SIZE)(%edx, %ecx, 1), %eax
    movl    (0x1*L2_WAY_SIZE)(%edx, %ecx, 1), %eax
    movl    (0x2*L2_WAY_SIZE)(%edx, %ecx, 1), %eax
    movl    (0x3*L2_WAY_SIZE)(%edx, %ecx, 1), %eax
    /* move to next set */
    addl    $L2_LINE_SIZE, %ecx  /* i += L2_LINE_SIZE */
    cmpl    $L2_WAY_SIZE, %ecx
    jl      3b              /* loop if i < L2_WAY_SIZE */

    pushl   -0x4(%ebp)
    call    restore
    addl    $0x4, %esp
    leave
    ret
```

Figure B.3: Software routine to flush the L2 cache.