View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by epublications@Marquette

Marquette University
e-Publications@Marquette

Computer Science Faculty Research and Publications
Computer Science, Department of

1-2013

A Down-to-Earth Educational Operating System for Up-in-the-Cloud Many-Core Architectures

Michael W. Ziwisky
Marquette University

Kyle Persohn
Marquette University, kyle.persohn@marquette.edu

Dennis Brylow
Marquette University, dennis.brylow@marquette.edu

Follow this and additional works at: https://epublications.marquette.edu/comp_fac

Recommended Citation
Ziwisky, Michael W.; Persohn, Kyle; and Brylow, Dennis, "A Down-to-Earth Educational Operating System for Up-in-the-Cloud Many-Core Architectures" (2013). *Computer Science Faculty Research and Publications*. 9.
https://epublications.marquette.edu/comp_fac/9

Marquette University

e-Publications@Marquette

Computer Science Faculty Research and Publications/College of Arts and Sciences

***This paper is NOT THE PUBLISHED VERSION;* but the author's final, peer-reviewed manuscript.** The published version may be accessed by following the link in the citation below.

ACM Transactions on Computing Education, Vol. 13, No. 1 (January 2013). DOI. This article is © ACM and permission has been granted for this version to appear in e-Publications@Marquette. ACM does not grant permission for this article to be further copied/distributed or hosted elsewhere without the express permission from ACM.

A Down-to-Earth Educational Operating System for Up-in-the-Cloud Many-Core Architectures

Michael Ziwisky
Department of Mathematics, Statistics, and Computer Science, Marquette University Milwaukee, WI
Kyle Persohn
Department of Mathematics, Statistics, and Computer Science, Marquette University Milwaukee, WI
Dennis Brylow
Department of Mathematics, Statistics, and Computer Science, Marquette University Milwaukee, WI

We present Xipx, the first port of a major educational operating system to a processor in the emerging class of many-core architectures. Through extensions to the proven Embedded Xinu operating system, Xipx gives students hands-on experience with system programming in a distributed message-passing environment. We expose the software primitives needed to maintain coherency between many cores in a system lacking specialized caching hardware. Our proposed series of laboratory assignments adds parallel thread execution and inter-core message passing communication to a well-established OS curriculum.

Key Words
Xipx, Embedded Xinu, message passing, concurrency, many-core, SCC

1. INTRODUCTION

With modern processor clock frequencies approaching their practical maximum, the trend among chip manufacturers is to scale up the number of processing cores on a die. An N-core processor can execute N simultaneous instructions, leading to an N-fold performance increase under ideal circumstances. A typical modern multicore processor features between two and eight cores with sophisticated hardware for managing cache coherency between cores. However, this level of parallelism is just the beginning.

Manufacturers are already moving from multicore processors with a handful of cores to many-core chips consisting of dozens or even hundreds of cores. Notable examples of many-core platforms include the 64-core Tilera TILE64 [Bell et al. 2008], the Tilera TILEPro64, and the 48-core Intel Single-Chip Cloud Computer (SCC) [Howard et al. 2010]. In addition to these manufactured platforms, there are unrealized architecture designs such as the MIT ATAC [Kurian et al. 2010] which focuses on how to handle intercore communication when the processor scales beyond 1,000 cores. These current and emerging architectures add new demands to operating systems, yet current undergraduate-level OS courses fail to cover the important topics.

This article focuses on using an emerging many-core architecture, the Intel SCC, as a platform for an educational parallel operating system suitable for an undergraduate laboratory course. Our Xipx operating system is based on the proven pedagogical design of Xinu, an operating system widely used for research and teaching on a variety of platforms. This article presents three novel contributions.

> —We describe Xipx, the first port of a major educational operating system to a platform in the emerging class of many-core architectures.
>
> —We outline Xipx's message-passing device interface, which provides a correct and efficient mechanism for scalable communication between the 48 cores of the Intel SCC platform within a code base that remains tractable in the context of an undergraduate systems course.
>
> —We detail enhancements to the existing sequence of Embedded Xinu laboratory assignments that leverage Xipx to explore the concurrency and coordination issues in a modern many-core platform.

In the remainder of this article, we elaborate on the details of our Xipx curriculum and on the value of giving students a hands-on experience in developing important components of an operating system for a massively parallel architecture.

1.1. Prior and Related Work

As multicore and many-core architectures have become more common, there has been an explosion in research by computer science educators on when and how to introduce concurrent programming to undergraduate majors. Recent flagship ACM SIG conferences have included papers targeting educational aspects of many-core graphical processor units (GPUs) [Anderson et al. 2010] and heterogeneous clusters [Adams et al. 2011]. Most proposed curricula focus on the high-level language aspects of concurrent programming, although other emphases include correctness and testing of concurrent programs [Sadowski et al. 2011].

Whereas some advocate teaching concurrency to students using popular languages with built-in thread and synchronization support, such as Java, others have focused on using domain-specific languages, such as Erlang [Ortiz 2011], or on explicit modeling of coordination [Lin and Tatar 2011].

There is much discussion on when and where in the curriculum students should be introduced to concurrency [Ernst et al. 2009], with some arguing for early in the curriculum [Gross 2011], while others favor an incremental approach across all undergraduate levels [Brown and Shoop 2011].

Prior work has touched on using an educational operating system to support the addition of thread-level concurrency into specific systems courses such as compiler construction [Mallen and Brylow 2010]. Our work differs from this approach in its use of genuine hardware concurrency on a many-core system and in its focus on the undergraduate OS course.

The Xinu operating system [Comer 2011] has been ported to many platforms in the past 25 years and has a proven track record of both classroom and commercial usage. The modernized Embedded Xinu port [Brylow 2008; Brylow and Ramamurthy 2009] has seen a resurgence on inexpensive commodity hardware in courses including computer organization, operating systems, embedded systems, and networking.

Xipx differs from other recent Xinu ports in its focus on a many-core, Intel-based architecture.

1.2. The Intel SCC

The Intel SCC experimental processor is a “concept vehicle” created by Intel Labs as a platform for many-core software research [Howard et al. 2010]. It features 48 ×86-based cores organized into 24 tiles, each containing two cores, a 16kB “message-passing buffer” (MPB), and a router, as illustrated in Figure 1. This architecture foregoes cache coherency hardware for shared memory in order to reduce overhead and power consumption; coherency is managed entirely in software. Instead, the chip features hardware to accommodate an efficient message-passing system. Implementation of this message-passing paradigm makes for an illustrative series of laboratories to explore the challenges and requirements an OS must address to take full advantage of a many-core architecture.

Gaining access to many-core hardware can be a significant barrier to educators. While Intel’s SCC prototypes are in short supply, their eagerness to foster a thriving development community around this technology has made it possible for many universities around the world to gain remote access to SCC machines at no cost. The machines interface with an internet-connected workstation, called a management console PC (MCPC), to which users may remotely log in and gain control over the SCC. We have conducted all of our development and pilot experiments using remote access with no SCC of our own. In addition to our experiences, other universities have shown shared access for classroom purposes to be feasible [Strazdins 2012]. Requests for access to Intel’s Many-Core Architecture Research Community (MARC), including SCC platforms, can be sent to SCC Research Proposals@intel.com.

Fig. 1: Overview of the Intel SCC.

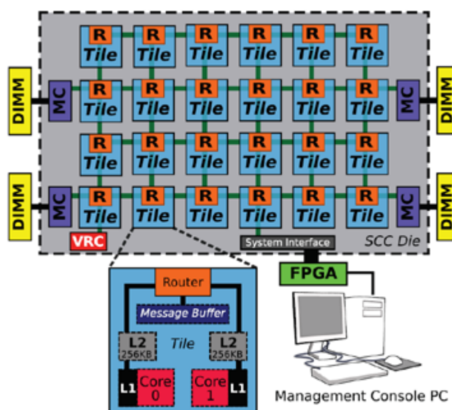

Fig. 1. Overview of the Intel SCC.

2. LABORATORY CURRICULUM

As outlined in Table I, our OS course emphasizing system-level parallelism is modeled on the established and proven Embedded Xinu curriculum [Brylow 2008] but includes new assignments for SCC serial communication (see Section 3), parallel execution speed-up (see Section 4), and intercore message passing (see Section 5).

Laboratories cover fundamental concepts for traditional operating systems along with necessary constructs for providing parallel thread execution support on a manycore system. The first seven assignments are the same as those in the Embedded Xinu curriculum [Brylow 2008], modulo some hardware differences between the MIPS and x86/SCC processors. Students write thread context switch and thread creation routines to learn about basic multitasking. Preemption and priority scheduling are added to the OS, and then students use a real-time clock and delta queues to implement thread sleeping. Students implement malloc() and free() to manage heap memory and synchronization constructs, including semaphores and test-and-set locks, for interthread communication. After building up a solid foundation for a single-core OS, the latter assignments require students to implement support for parallel execution both of independent threads and of tasks involving concurrent threads communicating with one another via message passing.

Table I. Typical Embedded Xinu Laboratory Curriculum Compared with Our Presented Xipx Curriculum

Lab	Xinu curriculum	Xipx curriculum
1	C programming review	same
2	UART serial communication	SCC serial communication (Sect. 3)
3	Context switch & nonpreempt. sched.	same
4	Priority scheduling & preemption	same
5	Synchronization & interthread comm.	same
6	Sleep & delta queues	same
7	Dynamic heap memory allocation	same
8	Ultra-Tiny file system	Parallel execution speed-up (Sect. 4)
9	Basic networking (Ping)	Intercore message passing (Sect. 5)

3. SCC SERIAL COMMUNICATION

One of the first challenges an operating system developer faces on any new platform is getting feedback from the hardware. For the Embedded Xinu curriculum, in which development is done on Linksys WRT54GL wireless routers, this means figuring out how to write to the router's on-board serial port, which sends characters to a terminal program via a physical connection to the developer's workstation.

The SCC platform does not include standard serial ports, and it is not clear how such hardware would scale to 48 cores if it did. Instead, SCC prototypes interface with an MCPC via a packet-exchanging protocol. Through this interface, the MCPC is able to read and write to most of the SCC components including configuration registers, message passing buffers, and RAM attached to the memory controllers. Xipx reserves an area of shared RAM for each core to act as a circular buffer for one-way serial communication. SCC cores write to this buffer while a pseudoterminal program on the MCPC periodically polls it and displays any new characters it finds.

3.1. Assignment: Serial Communication

The first Xipx assignment to differ from the standard Embedded Xinu curriculum is the serial communication lab. We have several key learning outcomes associated with this assignment. Students will:

(1) Familiarize themselves with the tools for compiling, deploying, and monitoring code on the 48-core SCC;
(2) Navigate the organization of shared memory space on a many-core system;
(3) Improve low-level data handling skills in C; and
(4) Review data structures topics from previous terms (circular buffers).

Students begin with a basic SCC framework that includes platform initialization code; handling for processor exceptions, faults, and interrupts; an implementation of a subset of the standard C library; and code to set up and jump into a C execution environment. (This “bare metal” SCC framework has been released as an open-source tool, and is described in Ziwisky and Brylow [2012].) Upon this foundation, students incrementally build up a fully parallel distributed operating system, one component at a time, beginning with a serial I/O interface.

For the first assignment using SCC hardware, students are given working source code for the pseudoterminal program—the “consumer” in the communication system— and they are tasked with developing an SCC putc() function—the “producer.” The putc() function must read the head and tail from shared RAM and, if the buffer is not full, place a character in the slot pointed to by the head, and then increment the head. The producer–consumer problem is commonly used as a simple example for teaching about the use of synchronization constructs. Indeed, we revisit this problem in the synchronization assignment later in the course. However, it is valuable for students to use a circular buffer to solve the kind of real-world problem that it commonly solves, which is queueing data that is transferred between two entities. Students discover another practical use for this data structure in the message passing assignment, as discussed in Section 5.4.

Although writing to a circular buffer may not seem like a particularly challenging task, students have much to gain from several aspects of this assignment. This is their first exposure to the compilation, kernel deployment, and execution tools available for SCC development. In addition, students face the important challenge of coding “in the dark,” that is, without any sort of output from the CPU to help debug during development. This is a critical step in the initial stages of software development on a new hardware platform.

4. PARALLEL EXECUTION SPEED-UP

Students complete Assignments 3 through 7 from the standard Embedded Xinu sequence with only minor architecture-dependent variations on a single SCC core. Following Assignment 7, we begin exploring the parallelism of the SCC platform.

Brute-force search may be used to crack an encrypted password when the encrypting function is known. The brute-force procedure of testing every possible password is easily divided into parallel tasks that may execute independently of each other. With a well-balanced workload among p parallel threads, the time it takes to solve the problem approaches t/p, where t is the execution time of the serial solution.

4.1. Assignment: Parallel Speed-UP

Our first parallel programming assignment involves password cracking by brute-force attack. Working on this assignment, students will:

(1) access hardware-dependent core numbering mechanisms and use this information to organize core coordination and I/O,
(2) divide a problem into independent tasks that are scheduled for parallel execution, and
(3) quantitatively measure performance gains of implemented parallel solutions.

Now that multiple cores will be involved in computations, the first task for students to complete is writing get my coreid(). This function reads MYTILEID, an SCC configuration register, and returns a number 0 through 47 corresponding to the ID of the core that called the function. The function finds its first use as a demultiplexer for terminal output. Students are given an updated SCC pseudoterminal that now checks 48 different buffers in shared memory, one for each SCC core, and prepends each printed line with a number [00] through [47] corresponding to the buffer from which the line was retrieved. To take advantage of the new terminal, students must modify the putc() function written in the earlier assignment so that it uses their new get my coreid() to determine which buffer to modify.

```
function SERIAL_SEARCH(encryptedPW)
    for All possible 5-letter test strings do
        Apply Hill cypher to test string;
        if The result matches encryptedPW then
            PRINT the test string;
        end if
    end for
end function

function PARALLEL_SEARCH(encryptedPW, p)
    Calculate search domain based on p (the number of
participating cores) and the rank of this core;
    for All 5-letter test strings in this core's domain do
        Apply Hill cypher to test string;
        if The result matches encryptedPW then
            PRINT the test string;
        end if
    end for
end function
```


```
function SERIAL_SEARCH(encryptedPW)
    for All possible 5-letter test strings do
        Apply Hill cypher to test string;
        if The result matches encryptedPW then
            PRINT the test string;
        end if
    end for
end function

function PARALLEL_SEARCH(encryptedPW, p)
    Calculate search domain based on p (the number of
participating cores) and the rank of this core;
    for All 5-letter test strings in this core's domain do
        Apply Hill cypher to test string;
        if The result matches encryptedPW then
            PRINT the test string;
        end if
    end for
end function
```


Fig. 2. Serial and parallel brute-force search algorithms for cracking Hill-enciphered passwords consisting of five uppercase alphabetic characters.

Students then implement a serial version and a parallel version of brute-force search for password cracking. Given an encrypted password and the encrypting function, each possible password must be run through the encrypting function to test whether it produces the encrypted password. We use the Hill cipher [Hill 1929] as a simple encrypting function and limit the search space to the set of passwords consisting of exactly five uppercase alphabetic characters. Although the simple cipher has obvious flaws for modern cryptographic use, we find this limited problem to be well scoped for a pedagogical setting; a serial search of the full space takes roughly 1 minute on the SCC hardware, and a typical p-core parallel implementation was measured to have a

speed-up of about 0.995p. Encryption collisions are possible with this scheme, and students are asked to find all possible passwords that match the given encrypted one. This necessitates a search of the entire space no matter how early a matching password is encountered. Example implementations of the two brute-force search routines are shown in Figure 2.

5. PARALLEL ALGORITHMS WITH SHARED DATA

Brute-force password cracking is an example of an easily distributed workload because it incorporates a loop with many independent iterations, which can be farmed out to independent threads. More complex parallel algorithms require data to be shared between threads.

Our OS course engages students with diverse background knowledge from several academic departments. Many domain-specific problems such as medical imaging, cryptography, and computer graphics involve computationally intense matrix multiplication. In our parallel systems curriculum, we demonstrate a matrix algorithm relevant to many students' fields, with enough abstraction to minimize prerequisite knowledge from a particular discipline.

It is often desirable for matrix multiplication to leverage the power of parallel and distributed systems as data sets scale beyond the processing capabilities of a traditional single-core workstation. Cannon's algorithm is a memory-efficient matrix multiplication technique for distributed computer architectures. For a detailed discussion of this algorithm, see Lee et al. 1997.

5.1. Shared Memory

To implement Cannon's parallel data-sharing algorithm, data blocks must be passed between cores. A first idea may be to designate regions of RAM as shared memory and coordinate the cores so that interacting pairs agree on where blocks will be read and written. For synchronization, a status flag is designated for each sharing pair that indicates whether the shared data is new or old. After a core passes a block to its neighbor, it sets the flag to indicate that it is finished and new data can be read. After the neighbor reads and no longer needs the block in shared memory, it clears the flag to indicate that old data can be overwritten.

In the classroom, this shared memory scheme is explained and then demonstrated by executing an implementation of Cannon's algorithm that uses it. When this demonstration is moved from a cache-coherent multicore system to the SCC, it does not work as expected. In fact, the calculation never completes. Why is this? The reason is that the many cores of the SCC are not cache coherent.

5.2. Cache Coherency

To understand why our shared memory implementation does not work without cache coherence, students must understand what cache memory is, why it is used, and how it can affect the operation of a multi- or many-core CPU. Understanding caching is already a primary learning outcome for an OS course, even without many-core hardware. A deep understanding of why and how computers cache, along with how various kinds of cache coherency impact a system, is a fundamental requirement for students who will face a future full of many-core and distributed computing systems. Thus, while some of the material in this unit may seem overly technical or tied to a particular emerging architecture, the pedagogical focus must remain on providing students with an impactful encounter with the effects of caching.

Cache memory is divided into lines (one line may be 32 bytes in a typical system), and data copies between RAM and cache occur one entire line at a time. The first time a processor wants to read from a given memory location, it finds that location in RAM and loads a line that includes that location into cache so that the data there can be accessed quickly if it is needed again. If the caching policy is write-back, subsequent writes to that location only update cache, and RAM will contain inaccurate data. The CPU notes this condition by marking the

cache line as dirty. Because there is so much more RAM than cache, a cache line must be shared between many line-sized sections of RAM. Eventually the cache line will need to be overwritten by data from a different section of RAM. When this happens and the line is dirty, the cached data is written back to RAM. Alternatively, a write executed on write-through cache causes RAM to be updated immediately.

The existence of multiple processing cores adds additional complexity to the system. Each core in a multicore system may have its own cache and, therefore, has its own "idea" of what is in RAM. If a core changes its copy of a shared location, then all others caching that location must be notified of the change. In a cache-coherent multicore system, hardware exists to ensure that when one core modifies a cached resource, all other cores that also have that resource cached are notified of the event. If the resource is to be used by any of those cores again, they will first update their copy of it 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.

Cache coherency hardware alleviates the system programmer of much of the burden involved in dealing with cache and, therefore, makes its existence fairly transparent. Unfortunately, current techniques for maintaining coherency have not scaled beyond a handful of cores because of the required hardware overhead, bus usage, and power consumption. Intel SCC does not include coherency hardware and leaves it to the system programmer to take care of coherency in software. This is a significant burden for programmers who are intent on using a shared-memory approach to coordinate work between cores; the alternative approach is to disallow shared memory in favor of other distributed paradigms.

Each of the SCC's 24 tiles has 16kB of on-die memory—referred to as a message passing buffer (MPB)—which can be accessed by any of the 48 cores. Cores have direct access to their local MPB and also to remote MPBs via the high-speed router mesh of the SCC. The MPB is essentially a small, fast shared memory. The limited size and exceptional speed of the MPB and router mesh makes them well suited as a backbone for an on-chip message passing system.

5.3. Message Passing

Understanding message passing and other interthread communication mechanisms is already a primary learning outcome for traditional single-core OS courses. In our coverage of this topic, we are able to illustrate many aspects of the general concept of message passing with the specific hardware implementation found on the SCC.

Message passing differs fundamentally from shared memory on the SCC because the MPB hardware allows message data to be copied directly into the private memory space of the recipient. Cache coherence is neither assumed nor required, making message passing a likely communication paradigm for future many-core architectures.

In an OS course, we examine how the system provides the abstraction of OS message passing on top of tricky hardware details like buffering messages and intercore signaling. The API provides two important functions: send msg() and recv msg().

5.4. Implementation of Message Passing

Among the details of the message passing framework is a need to ensure that no message gets lost when multiple messages are sent to a core before it gets a chance to start retrieving them. Implementing a circular buffer in the MPB provides a way to hold multiple messages in one mailbox without having unread messages overwritten. But what happens if two cores try to send a message to the same mailbox at the same time? If both cores read the head before either of them write their message and update the head, then they will both be trying to write to the same memory address and one or both of the messages are transmitted incorrectly. This is known as a data race, and it is one type of synchronization problem that can occur in a multiprocessor

environment. To prevent this problem, the system should enforce mutual exclusion on the MPB, which is to say that each MPB can only be accessed by one core at a time.

Students first explore mutual exclusion in the context of multiple threads executing on a single core in the assignment on synchronization and interthread communication. This assignment is a staple of a traditional OS curriculum, and the SCC features classic hardware mechanisms for experimenting with the conceptual topics it covers. Because this article does not discuss that assignment, the relevant details from it are disclosed here. The aforementioned hardware mechanisms featured on the SCC are a set of test-and-set registers, one per core, which provide a simple mechanism for mutual exclusion. If a sending core wants to write to some receiving core's MPB, it must own the lock of the receiver. To ask for the lock, the sender reads the receiver's test-and-set register. If a "0" is read, the lock is already owned by another core, and the sender does not have permission to write to the MPB. If a "1" is read, the sender has acquired the lock, and any subsequent reads from any core will return a "0." To release the lock, the owner only has to issue a write to the test-and-set register. This resets its value to "1" so that it is given up again the next time it gets read. Part of the earlier assignment on synchronization and interthread communication is to write two functions—acquire lock(int coreid) and release lock(int coreid)—to access the lock registers for any core.

The final important decision to consider in implementing send msg() is how to delimit messages on the MPBs. One option is to require that all messages be of a fixed length, but this may be inefficient in memory usage due to excessive internal fragmentation. Instead, a simple header containing the message length is tacked onto the front of each message. The source core ID is also included in the header for demultiplexing, and the destination core ID is included for debugging purposes. Combining the buffering, synchronization, and header construction, the implementation of send msg() is shown in Figure 3. Note that this implementation is left incomplete in the interest of illustrating the key aspects—a full implementation would use the length of message and the head position to determine if the message must wrap around the end of the buffer, and it would perform the actual wrapping if necessary. These are trivial additions to the code. When a user thread calls recv msg(), it passes in a pointer to a location to which the message will be copied. If a message is already waiting in the MPB, recv msg() will (i) immediately copy the message to the requested location, (ii) free up the space on the MPB by incrementing the tail of the circular buffer, and (iii) return the length of the retrieved message. If a message has not yet been sent to the receiving core, then recv msg() will put the calling thread to sleep until a message arrives. How does the thread know whether a message is waiting, and how does a sleeping thread get notified when a message arrives? Another function on the receiving core cooperates with recv msg() behind the scenes—interrupt handler handle msg() reacts to the call to interrupt core() at the end of the send msg() routine.

Interaction between recv msg() and handle msg() is regulated by a classic semaphore, a topic covered in the earlier standard assignment on synchronization and interthread communication.

```
function SEND_MSG(toCore, msg, len)
    Acquire lock of toCore;
    Construct header containing senderCore, toCore, and len;
    Read head pointer for toCore's MPB;
    Copy header followed by msg to toCore's head pointer;
    Update head pointer;
    Release lock of toCore;
    Interrupt toCore;
end function
```

```
function SEND_MSG(toCore, msg, len)
    Acquire lock of toCore;
    Construct header containing senderCore, toCore, and len;
    Read head pointer for toCore's MPB;
    Copy header followed by msg to toCore's head pointer;
    Update head pointer;
    Release lock of toCore;
    Interrupt toCore;
end function
```

Fig. 3: A basic version of the message sending routine.

5.5. Assignment: Intercore Message Passing

The final Xipx assignment provides students with valuable problem-solving experience in managing a parallel execution environment with many concurrent, interacting threads. Learning outcomes associated with this assignment include:

—considering strengths and weaknesses of design decisions in the context of a problem with a many-dimensional design space;

—expanding a synchronization mechanism originally developed for a single multitasking core into one that is suitable for coordinating multiple concurrent cores; and

—building a framework for asynchronous communications between arbitrary pairs of processing cores.

Students' primary goals for the message passing assignment are to implement the three key functions of the message passing system—send msg(), recv msg(), and handle msg()—and to fix the cache-coherent shared memory implementation of Cannon's algorithm by modifying it to utilize the new functions. In addition, two functions—acquire lock() and release lock()—originally written in the assignment on synchronization and interthread communication to access the lock register for the local core only, must now be modified to access the lock register of any core.

Along with the cache-coherent Cannon code, students receive a working setupLINT() function for registering an interrupt handler with an external interrupt signal, and an interrupt core() function that signals the external interrupt pin of a remote core. Successful completion of this assignment depends on proper usage of these functions and of correct semaphore and locking functions.

It is important to note that the design space for a message passing framework is vast [Rotta 2011]. Design considerations include placement of messages, allocation of MPB memory, and new-message notification techniques. Our chosen method—putting messages on the recipient's MPB, allowing any core to write to any location of the buffer, using a header to identify the sender and message length, and triggering a single interrupt handler to notify the recipient of a new message—is not the only correct solution. Other MPB designs entail different trade-offs but present a rich space for further exploration and laboratory assignment variations.

6. EVALUATION AND DISCUSSION

Our initial pilot experiment putting the Xipx curriculum in student hands was too small to draw strong conclusions from; a larger empirical study is planned for the next iteration of the course. However, the pilot run has confirmed that these laboratory experiments are feasible, can be successfully undertaken by upper-division undergraduates with no prior parallel systems background, and anecdotally seem to increase familiarity with parallel and distributed concepts. Several improvements to the curriculum materials were suggested and have been implemented for the next iteration, including a more thorough description of the SCC debugging tools.

Student comments from the pilot run included the following.

— "It's cool to see how an algorithm can actually be distributed."

— "There's always something in the back of your mind when you're working on this thing (the SCC). You're so used to thinking in a single timeline, keeping track of just one instruction pointer, even when the single-core machine is using interrupts and time-slicing. With the SCC, you always wonder, where are those other 40-some instruction pointers?"

— "I don't know that it's harder; it's just . . . different."

Although it was agreed that Xipx assignments on the SCC were "cool" and helped demonstrate difficult parallel systems concepts, it remains future work to determine whether there is quantitative improvement in learning outcomes from this sphere.

Although most of the assignments in the Xipx track were considered comparable to their single-core Embedded Xinu counterparts, modulo some hardware differences, we noted that at least two of the assignments took notably longer to complete than expected. We have several possible causes for this that we would like to explore in greater depth for the next run. First, as alluded to in the second student comment, the mere presence of the other cores may be advancing the apparent cognitive load of these laboratory assignments. Our experience has been that many students are able to proceed (perhaps unwisely) with little regard for concurrency issues when programming single-core assignments, even in the presence of interrupt-driven multitasking. The SCC may make students more cognizant of these perils, even though comparable perils often exist in the single-core variant. Second, the complexity of SCC debugging facilities seemed to detract from student productivity in some circumstances. Finally, the SCC cores are CISC machines, which makes for an awkward comparison with our default single-core RISC platforms in some cases.

7. CONCLUSIONS AND FUTURE WORK

We have presented Xipx, an educational operating system supporting concurrent, distributed thread execution on an emerging many-core architecture. The Xipx curriculum is based on the successful Embedded Xinu curriculum, with extensive modifications to embrace this new class of complex hardware with growing importance. Students lacking knowledge and experience with the system-level mechanisms of these architectures will be unprepared for the world they enter after college. Knowledge of the inner workings of many-core and distributed systems are useful not only for system developers but also for application programmers; programmers are more capable of taking advantage of the parallel systems for which they are developing if they understand how the system uses their code.

There are several pressing aspects of future work to attend to. The first is to quantitatively evaluate the effectiveness of our new laboratory activities in an active operating systems course. Limited student access to experimental many-core platforms in the past has made this a challenge, but our pilot experiment suggests that the barrier is lowering, and the effort will be worth the rewards. A larger pseudoexperiment is planned for our next offering of our course. We will evaluate and report on the learning outcomes of these Xipx students with respect to our objectives and in comparison to the outcomes of the students following the traditional Embedded Xinu route.

On the technology side, an implementation of a thread migration protocol for Xipx is currently under development. Thread migration is an important part of a distributed OS environment like Xipx because it allows the system to redistribute workloads across the many cores of the SCC for greater throughput and flexibility. Our implementation follows the minimalist, elegant instructional design of the Xipx system to ensure that it is well suited for the classroom. We intend to incorporate this work into future iterations of the Xipx curriculum.

The open-source Xipx code and descriptions of the Xipx track of our OS laboratory assignments are available at the Embedded Xinu site, http://xinu.mscs.mu.edu/.

We believe that undergraduate computer science and engineering majors can and should understand the primary system structures that underlie programming in the many-core world. Our motto, in summary, is: “There is no magic in the multicore box!”

ACKNOWLEDGMENTS

The authors thank members of the Intel MARC community, particularly Ted Kubaska and Jan-Arne Sobania, for prompt and clear responses to questions arising during development on the SCC.

REFERENCES

ADAMS, J. C., HOOBEBOOM, K., AND WALZ, J. 2011. A cluster for CS education in the manycore era. In Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE’11). ACM, New York, 27–32.

ANDERSON, N., MACHE, J., AND WATSON, W. 2010. Learning CUDA: Lab exercises and experiences. In Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (SPLASH’10). ACM, New York, 183–188.

BELL, S., EDWARDS, B., AMANN, J., CONLIN, R., JOYCE, K., LEUNG, V., MACKAY, J., REIF, M., BAO, L., ET AL. 2008. Tile64 processor: A 64-core SoC with mesh interconnect. In Proceedings of the International Solid-State Circuits Conference.

BROWN, R. AND SHOOP, E. 2011. Modules in community: Injecting more parallelism into computer science curricula. In Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE’11). ACM, New York, 447–452.

BRYLOW, D. 2008. An experimental laboratory environment for teaching embedded operating systems. In Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education (SIGCSE’08). ACM, New York, 192–196.

BRYLOW, D. AND RAMAMURTHY, B. 2009. Nexos: A next generation embedded systems laboratory. SIGBED Rev. 6, 1.

COMER, D. E. 2011. Operating System Design: The XINU Approach Linksys Version. CRC Press.

ERNST, D., WITTMAN, B., HARVEY, B., MURPHY, T., AND WRINN, M. 2009. Preparing students for ubiquitous parallelism. In Proceedings of the 40th ACM Technical Symposium on Computer Science Education (SIGCSE’09). ACM, New York, 136–137.

GROSS, T. R. 2011. Breadth in depth: A 1st year introduction to parallel programming. In Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE’11). ACM, New York, 435–440.

HILL, L. S. 1929. Cryptography in an algebraic alphabet. Amer. Math. Mon. 36, 6, 306–312.

HOWARD, J., DIGHE, S., HOSKOTE, Y., VANGAL, S., FINAN, D., RUHL, G., JENKINS, D., WILSON, H., BORKAR, N., ET AL. 2010. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In Proceedings of the International Solid-State Circuits Conference.

KURIAN, G., MILLER, J., PSOTA, J., EASTEP, J., LIU, J., MICHEL, J., KIMERLING, L., AND AGARWAL, A. 2010. ATAC: A 1000-core cache-coherent processor with on-chip optical network. In Proceedings of Parallel Architectures and Compilation Techniques.

LEE, H.-J., ROBERTSON, J. P., AND FORTES, J. A. B. 1997. Generalized Cannon’s algorithm for parallel matrix multiplication. In Proceedings of the 11th International Conference on Supercomputing (ICS’97). ACM, New York, 44–51.

LIN, S. AND TATAR, D. 2011. Encouraging parallel thinking through explicit coordination modeling. In Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE’11). ACM, New York, 441–446.

MALLEN, A. AND BRYLOW, D. 2010. Compiler construction with a dash of concurrency and an embedded twist. In Proceedings of the 25th ACM SIGPLAN Symposium on Systems, Programming, Languages, and Applications: Software for Humanity. 161–168.

ORTIZ, A. 2011. Teaching concurrency-oriented programming with Erlang. In Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE'11). ACM, New York, 195–200.

ROTTA, R. 2011. On efficient message passing on the Intel SCC. In Proceedings of the 3rd Many-core Applications Research Community (MARC) Symposium

SADOWSKI, C., BALL, T., BISHOP, J., BURCKHARDT, S., GOPALAKRISHNAN, G., MAYO, J., MUSUVATHI, M., QADEER, S., AND TOUB, S. 2011. Practical parallel and concurrent programming. In Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE'11). ACM, New York, 189–194.

STRAZDINS, P. E. 2012. Experiences in teaching a specialty multicore computing course. In Proceedings of the IEEE International Parallel & Distributed Processing Symposium Workshops. 1271–1276.

ZIWISKY, M. W. AND BRYLOW, D. W. 2012. BareMichael: A minimalistic bare-metal framework for the Intel SCC. In Proceedings of the 6th Many-core Applications Research Community (MARC) Symposium.