CAREER: Deterministic Shared Memory Multiprocessing: Vision, Architecture and Impact on Programmability

Project Summary

Developing multithreaded software has proven to be much more difficult than writing single-threaded code. One of the main reasons is that multithreaded code in current shared memory multicore systems can execute nondeterministically. Each time a multicore executes a multithreaded application, it can produce a different output even if supplied with the same input. This frustrates debugging efforts and limits the ability to properly test multithreaded code, becoming a major obstacle to widespread adoption of parallel programming.

Past efforts to addressing the problem of nondeterminism have primarily focused on deterministic replay and deterministic parallel programming models. The former is useful only for debugging, while the latter is typically domain-specific. In contrast, this proposal envisions that shared memory multiprocessor systems should always behave deterministically when executing *any* shared memory parallel program. The core idea is to make inter-thread communication appear to be fully deterministic by guaranteeing equivalence to a deterministic serialized execution. This can be made efficient by employing speculation and exploiting properties of memory sharing behavior of applications.

Intellectual Merit. This proposal poses broad intellectual questions with far-reaching implications in modern computer systems: *Can nondeterminism be removed from shared-memory multiprocessor systems without degrading performance? What are the trade-offs in designing deterministic multiprocessor systems? What are the implications and uses of deterministic behavior in programmability?* **This CAREER proposal seeks to answer these questions by devising efficient, general purpose, fully deterministic shared memory multiprocessor systems and demonstrating that they can enable significant changes in how parallel programs are written, tested and deployed**. In addition to the obvious benefits deterministic multiprocessing affords to debugging, i.e., repeatability, we argue that parallel programs should always execute deterministically in the field. This will add assurance to software testing, allow a more meaningful collection of crash information, and increase the reliability of multithreaded software deployed in the field. Interestingly, the same techniques we will develop to enable deterministic execution can also be used to improve the testing of multithreaded programs and proactively avoid concurrency bugs in the field. Specific research objectives include: (1) developing an architecture for efficient deterministic multiprocessing, from mechanisms to the hardware/software interface, (2) addressing system issues, e.g., executing an operating system in deterministic multiprocessors and supporting multiple deterministic processes, and (3) leveraging the deterministic multiprocessing hardware/software interface to create tools for debugging, testing and bug avoidance.

Broader Impact. How to popularize parallel programming is one of Computing Research Association's Grand Research Challenge for the systems community. Being able to leverage the full potential of multicore systems would put us back into exponential growth of usable performance as well as lead to significant power savings. Making multiprocessors deterministic would have a transformative effect in the IT industry, as it attacks a problem at the heart of the programmability issues in multiprocessor systems. The PI's prior work in the area of concurrency has helped develop many close contacts with industrial research labs, especially IBM Research, Microsoft Research and the newly formed AMD Research and Advanced Development Labs. The PI will leverage these contacts to solicit feedback on our efforts and increase the impact of the contributions that will result from this proposal.

An integral part of the concurrency challenge is teaching our students to "think parallel". **The high-level educational goal of this proposal, then, is to develop a graduate and undergraduate curriculum at the University of Washington that will teach students about concurrency principles and practical programming.** The key philosophy in both the undergraduate and graduate educational plan in this proposal is deep integration of research into teaching. This is especially natural in the problem space of this proposal, the concurrency revolution. As we experiment with new models and systems, we both teach students and learn the programmability benefits of research ideas. The PI will take advantage of the University of Washington's Minority Science and Engineering Program to realize his strong commitment to underrepresented minority outreach in his research and teaching activities.

CAREER: Deterministic Shared Memory Multiprocessing: Vision, Architecture and Impact on Programmability

1 Introduction

How to **popularize parallel programming** is a Grand Research Challenge for the systems community [1]. Being able to leverage the full potential of multicores would put us back into exponential growth of usable performance as well as lead to significant power savings. However, developing multithreaded software has proven to be much more difficult than writing singlethreaded code. One of the major challenges is that current multicores execute multithreaded code nondeterministically [32]: given the same input, threads can interleave their memory and I/O operations in different ways each time an application is executed.

Nondeterminism in multithreaded execution arises from small perturbations in the execution environment, e.g., other processes executing simultaneously, differences in the operating system resource allocation, state of caches, TLBs, bus and other micro-architectural structures. The ultimate consequence is a change in program behavior in each execution. Nondeterminism complicates the software development process significantly. Defective software might execute correctly hundreds of times before a subtle synchronization bug appears, and when it does, developers typically can not reproduce it during debugging. In addition, nondeterminism also makes it difficult to test multithreaded programs, as good coverage requires both a wide range of program inputs and wide range of possible interleavings. Moreover, if a program might behave differently each time it is run with the same input, it becomes hard to assess test coverage. For that reason, a significant fraction of the research on testing parallel programs is devoted to dealing with nondeterminism.

Past efforts to addressing the problem of nondeterminism have primarily focused on deterministic replay and deterministic parallel programming models. Deterministic replay [23, 25, 31, 40, 44, 54, 63] relies on a previously generated log to reproduce a given parallel execution and is very useful for debugging. Deterministic parallel programming models, such as stream programming languages [58] and implicitly parallel languages [24, 53], yield parallel programs that behave deterministically but tend to be domainspecific. **In contrast, this proposal envisions that shared memory multiprocessor systems should always behave deterministically when executing any shared memory parallel program.** Making multiprocessors deterministic would have a transformative effect in the IT industry, as it attacks a problem at the heart of the programmability issues in multiprocessor systems.

This CAREER proposal seeks to devise efficient, general purpose, fully deterministic shared memory multiprocessor systems and demonstrate that they can enable significant changes in how parallel programs are written, tested and deployed. Figure 1 shows the impact of determinism over the lifetime of multithreaded software. In addition to the obvious benefits deterministic multiprocessing affords to debugging, i.e., repeatability, we argue that parallel programs should always execute deterministically in the field. This will make software testing more assuring, allow a more meaningful collection of crash information, and increase the reliability of multithreaded software deployed in the field. Interestingly, the same techniques we will develop to enable deterministic execution can also be used to improve the testing of multithreaded programs and proactively avoid concurrency bugs in the field.

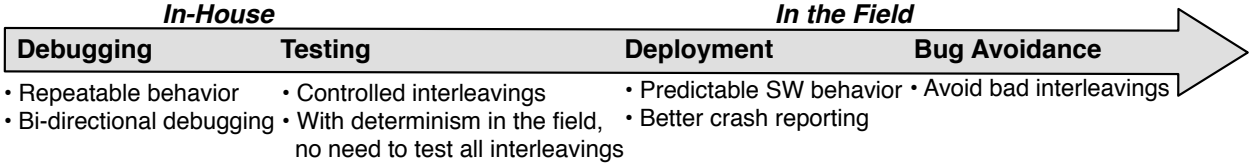

Figure 1: Impact of deterministic multiprocessing in the lifetime of multithreaded software.

The high-level educational goal of this proposal is to develop a graduate and undergraduate curriculum at the University of Washington that will teach students how to think parallel. This is especially natural in the problem space of this proposal, the concurrency revolution. As we experiment with new models and systems, we both teach students and learn the programmability benefits of research ideas.

1.1 Relevant Prior Research Results and PI Qualifications

I have been conducting research on architecture support, compiler techniques, and programming models to improve the programmability of multiprocessor systems for over six years and have co-authored over twenty-five papers in these areas.

My doctoral research [13] included the *Bulk* architectural primitive, which cost-effectively enables coarse-grain operations on sets of memory references. Bulk's key idea is to encode sets of memory accesses into a concise signature and then provide signature operations that efficiently process sets of addresses. This enables simple support for the execution of chunks of dynamic instructions atomically and in isolation by using signatures to encode their read and write sets. Bulk has been used as the cornerstone of a variety of programmability features, e.g., hardware support for transactional memory and thread-level speculation [15]. In addition, as part of my thesis, I proposed *Bulk Enforcement of Sequential Consistency (BulkSC)* [16], which used Bulk to support coarse-grain memory ordering in multiprocessors [59, 61]. BulkSC groups sets of consecutive dynamic instructions into large chunks and enforces sequential consistency (SC) at the coarse grain of chunks. By amortizing expensive operations with chunks over many instructions and employing speculative execution, BulkSC yields virtually the same performance as relaxed memory models while still providing the strongest and most intuitive memory consistency model, namely, sequential consistency. The idea of Bulk signatures has been leveraged by several research projects in the architecture community [23, 39, 56, 65].

The work on BulkSC is especially relevant to this proposal because we have expanded it in two ways. First, we developed efficient support for *deterministic replay* of multiprocessor systems using the chunk abstraction [40] by logging the size and commit order of chunks; this is sufficient to later deterministically replay the execution since it yields the same memory interleaving. We also proposed making both chunk size and interleaving more deterministic so that less information is required to reconstruct the original thread interleavings, reducing log size. Second, we developed a system called Atom-Aid [35], that can detect and survive a broad class of concurrency bugs. Concurrency bugs manifest nondeterministically, i.e., they depend upon the occurrence of certain bad interleavings. Interestingly, a given memory semantics exposed to the software can yield multiple valid global interleavings of memory operations. One can allow only a subset of these interleavings to avoid concurrency bugs while still exposing the same memory semantics to the software. Atom-Aid leverages this property and avoids concurrency bugs by monitoring when one is likely to happen and choosing a "less dangerous" interleaving. Atom-Aid is built upon a BulkSC system and controls memory access interleaving by carefully choosing chunk boundaries.

Collectively, the key unifying theme of these prior research projects is controlling communication via shared memory (or memory interleaving) for coarse-grain memory ordering, deterministic replay and bug avoidance. This proposal will integrate and leverage these insights to build a robust foundation for deterministic control of inter-thread interaction in shared memory multiprocessors.

2 Deterministic Shared Memory Multiprocessing (DMP)

This section describes the foundation of deterministic multiprocessing. It defines deterministic parallel execution in shared memory multiprocessors (Section 2.1) and quantitatively assesses the degree of nondeterminism in current systems (Section 2.2). Section 2.3 describes preliminary designs for mechanisms to efficiently enforce deterministic multiprocessing, and Section 2.4 shows a preliminary evaluation of both hardware and software implementations.

2.1 A Definition of Deterministic Parallel Execution

We define a deterministic shared memory multiprocessor (DMP) system as a computer system that: (1) executes multiple threads that communicate via shared memory and system calls, and (2) will produce the same program output if given the same program input. This definition implies that a parallel program running on a DMP system is as deterministic as a single-threaded program.

The most direct way to guarantee deterministic behavior is to preserve the same global interleaving of instructions in every execution of a parallel program. However, several aspects of this interleaving are irrelevant for ensuring deterministic behavior. It is not important which global interleaving is chosen, as

long as it is always the same. Also, if two instructions do not communicate, their order can be swapped with no observable effect on program behavior. What turns out to be the key to deterministic execution is that all communication between threads must be precisely the same for every execution. This guarantees that the program always behaves the same way if given the same input.

Guaranteeing deterministic communication between threads requires that each dynamic instance of an instruction (consumer) read data produced from the same dynamic instance of another instruction (producer). Producer and consumer need not be in the same thread, so this communication happens via shared memory or I/O. Interestingly, there are multiple global interleavings that lead to the same communication between instructions, they are called *communication-equivalent interleavings*, illustrated in Figure 2. In summary, any communication-equivalent interleaving will yield the same program behavior. To guarantee deterministic behavior, then, we need to carefully control only the behavior of load, store and I/O operations that cause communication between threads. This insight is key to efficient deterministic execution.

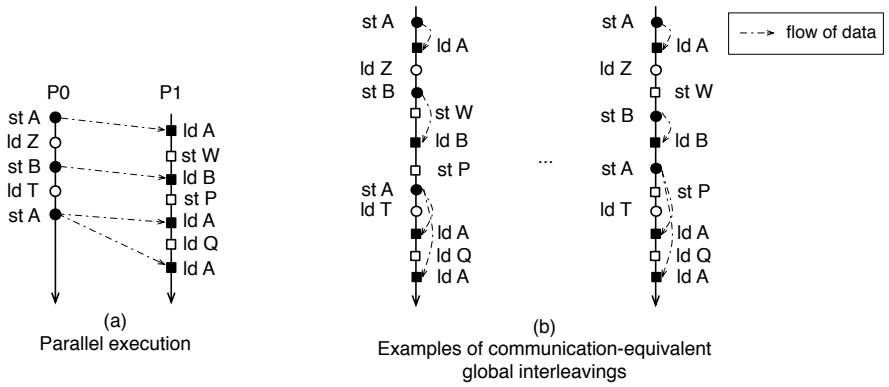

Figure 2: An illustration of a parallel execution (a) and two of its multiple communication-equivalent interleavings (b). Solid markers represent communicating instructions, while hollow markers represent instructions that do not communicate between threads.

2.2 Nondeterminism in Existing Systems

Current generation software and hardware systems are not built to behave deterministically. Multiprocessor systems execute programs nondeterministically because the software environment and the non-ISA micro-architectural state change from execution to execution. These effects manifest themselves as perturbations in the timing between events in different threads of execution, causing the final global interleaving of memory operations in the system to be different. Ultimately, the effect on the application is a change in which dynamic instance of a store instruction produces data for a dynamic instance of a load instruction. Once this occurs, program execution behavior may diverge from previous executions at the ISA level, and, consequently, program output may vary.

2.2.1 Sources of nondeterminism

The following are some of the software and hardware sources of nondeterminism.

Software Sources. Several aspects of the software environment create nondeterminism: other processes executing concurrently and competing for resources; the state of memory pages, power savings mode, disk and I/O buffers; and the state of global data structures in the OS. In addition, several operating system calls have interfaces with legitimate nondeterministic behavior. For example, the *read* system call can legitimately take a variable amount of time to complete and return a variable amount of data.

Hardware Sources. A number of non-ISA-visible components vary from program run to program run. Among them are the state of any caches, predictor tables and bus priority controllers, in short, any micro-architectural structure. In fact, certain hardware components, such as bus arbiters, can change their outcome with each execution purely due to environmental factors; e.g., if the choice of priority is based on which signal is detected first, the outcome could vary with differing temperature and load characteristics.

2.2.2 Quantifying nondeterminism

As mentioned previously, nondeterminism in program execution occurs when a particular dynamic instance of a load reads data created from a different dynamic instance of a store. To measure this degree of nondeterminism, we start by building a set containing all pairs of communicating dynamic instructions (dynamic store $\rightarrow$ dynamic load) that occurred in an execution. We call this the *communication set*, or C. A dynamic memory operation is uniquely identified by its instruction address, its instance number[1], and the id of the thread that executed it. Assume there are two executions, $ex1$ and $ex2$, whose communicating sets are C_{ex1} and C_{ex2}, respectively. The symmetric difference between the communication sets, $C_{ex1} \triangle C_{ex2} = (C_{ex1} \cup C_{ex2}) - (C_{ex1} \cap C_{ex2})$, yields the communicating pairs that were *not* present in both executions, which quantifies the execution difference between $ex1$ and $ex2$. Finally, we define the amount of nondeterminism between $ex1$ and $ex2$ as: $ND_{ex1,ex2} = \frac{|C_{ex1} \triangle C_{ex2}|}{|C_{ex1}|+|C_{ex2}|}$ (Eq. 1), which represents the proportion of all communicating pairs that were *not* in common between the two executions. Note that this number correctly accounts for extra instructions spent in synchronization, as they will form some communicating pair. We developed a tool using PIN [36] that profiles the execution of a multithreaded program and builds the communication sets.

Figure 3 shows two examples of the results (*barnes*, *ocean-contig*) [62] running on an Intel Core 2 Duo machine. The plots show that programs have significant nondeterministic behavior. Moreover, they reveal two interesting properties. First, both plots depict phases of execution where nondeterminism drops to nearly zero. These are created by barrier operations that synchronize the threads and then make subsequent execution more deterministic. Second, *ocean-contig* never shows 100% nondeterminism, and, in fact, a significant fraction of pairs are the same, i.e., typically those from private data accesses. These aspects of program execution are significant and can be exploited in designing a system that is actually deterministic.

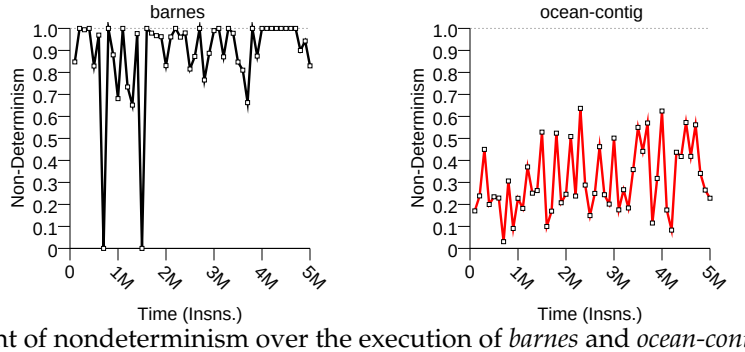

Figure 3: Amount of nondeterminism over the execution of *barnes* and *ocean-contig*. The x axis is the position in the execution where each sample of 100,000 instructions was taken. The y axis is ND (Eq. 1) computed for each sample in the execution.

2.3 Enforcing Deterministic Shared Memory Multiprocessing

We have developed initial ideas on how to build a DMP system. This section focuses on the key mechanisms without discussing specific implementations. We begin with a basic naive approach, and then refine this simple technique into progressively more efficient organizations.

2.3.1 Basic Idea

As seen earlier, making multiprocessors deterministic depends upon ensuring that the communication between threads is deterministic. The easiest way to accomplish this is to allow only one processor at a time to access memory in a deterministic order. This process can be conceptualized as a memory access token being deterministically passed among processors. We call this *deterministic serialization* of a parallel execution, shown in Figure 4(b). Deterministic serialization guarantees that inter-thread communication is deterministic by preserving all pairs of communicating memory instructions.

[1]The instance number is obtained by keeping a per-thread counter of the number of dynamic memory operations executed by each instruction pointer.

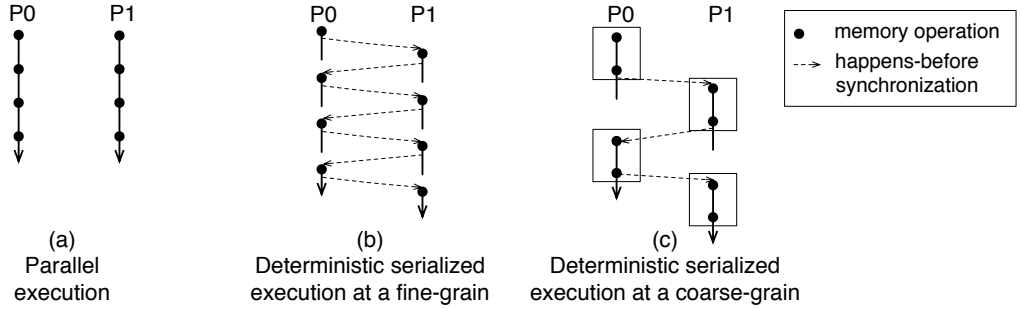

Figure 4: Deterministic serialization of memory operations.

The simplest way to implement such serialization is to have each processor obtain the memory access token (henceforth called *deterministic token*) and, when the memory operation is completed, pass it to the next processor in the deterministic order. A processor blocks whenever it needs to access memory but does not have the deterministic token.

Waiting for the token at every memory operation is likely to be expensive and will cause significant performance degradation when compared to the original parallel execution (Figure 4(a)). Performance degradation stems from overhead introduced by waiting and passing the deterministic token and from the serialization itself, which removes the benefits of parallel execution. Synchronization overhead can be mitigated by synchronizing at a coarser granularity (Figure 4(c)), allowing each processor to execute a finite, deterministic number of instructions, or a *quantum*, before passing the token to the next processor. A system with serialization at the granularity of quanta is called *DMP-Serial*. The process of dividing the execution into quanta is called *quantum building*: the simplest way to build a quantum is to break execution up into fixed instruction counts (e.g. 10,000).

2.3.2 Recovering Parallelism by Leveraging Communication-Free Execution

The performance of deterministic parallel execution can be improved by leveraging the observation that threads do not communicate all the time. Periods of the execution that do not communicate can execute in parallel with other threads. Thread communication, however, must happen deterministically. With *DMP-ShTab*, we achieve this by falling back to deterministic serialization only while threads communicate.

Inter-thread communication occurs when a thread writes to shared (or non-private) pieces of data. In this case, the system must guarantee that all threads observe that write at a deterministic point in their execution. DMP-ShTab does precisely that. Figure 5 illustrates how this works. There are two important cases: (1) reading data held private by a remote processor, and (2) writing to shared data (privatizing it). Case (1) is shown in Figure 5(a): when quantum 2 attempts to read data that is held private by a remote processor P0, it blocks and waits for the deterministic token. This is necessary to guarantee that quantum 2 always gets the same data, since quantum 1 might still write to A before it completes executing. Case (2) is shown in Figure 5(b): when quantum 1 — which already holds the deterministic token — attempts to write to a piece of shared data, it must wait until other processors reach a deterministic point in their execution, namely, the end of quantum 2. This is necessary to guarantee that all processors observe the change of the state of A (from shared to privately held by a remote processor) at a deterministic point in their execution.

To detect writes that cause communication, DMP-ShTab needs a global data-structure to keep track of the sharing state of memory positions. A *sharing table* is a conceptual data structure that contains sharing information for each memory position; it can be kept at different granularities, e.g., line or page. Figure 6 shows a flowchart of how the sharing table is used. A thread can access its own private data without holding the deterministic token (1). It can also read shared data without holding the token (2). However, in order to write to shared data or read data regarded as private by another thread, a thread needs to hold the token and wait until all other threads are blocked also waiting for the token (3). This guarantees that the sharing information is kept consistent and its state transitions are deterministic. When a thread writes to a piece of data, it becomes the owner of the data (4). Similarly, when a thread reads data not yet read by any thread, it becomes the owner of the data. Finally, when a thread reads data owned by another thread, the data becomes shared (5).

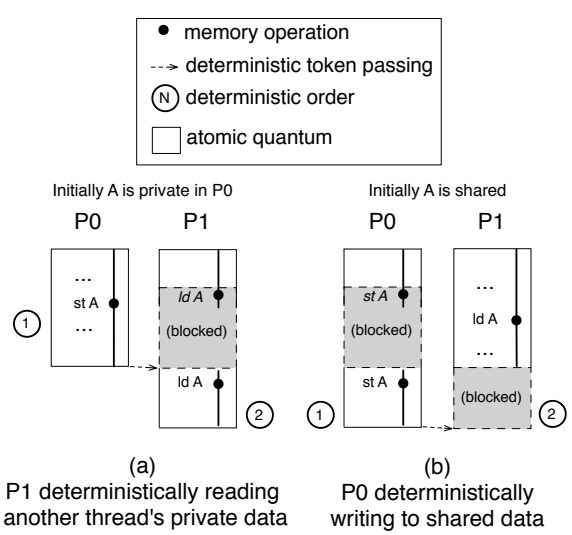

Figure 5: Recovering parallelism by overlapping communication-free execution.

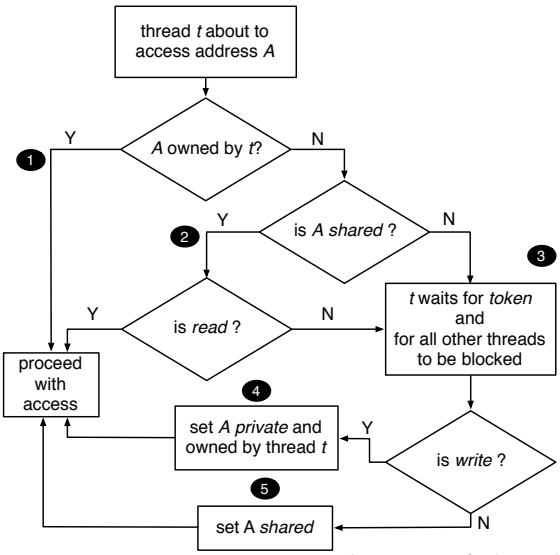

Figure 6: Deterministic serialization of shared memory communication only.

In summary, DMP-ShTab lets threads run concurrently as long as they are not communicating. As soon as they attempt to communicate, the sharing table deterministically serializes communication.

2.3.3 Recovering Parallelism by Leveraging Support for Transactional Memory (TM)

Executing quanta atomically and in isolation in a deterministic total order is equivalent to deterministic serialization of memory operations. To see why, consider a quantum executed atomically and in isolation as a single instruction in the deterministic total order, which is the same as DMP-Serial. Transactional Memory [20, 22] can be leveraged to make quanta *appear* to execute atomically and in isolation. This, coupled with a deterministic commit order, makes execution equivalent to deterministic serialization while recovering parallelism.

We use TM support by encapsulating each quantum inside a transaction, making it appear to execute atomically and in isolation. In addition, we need a mechanism to form quanta deterministically and another to enforce a deterministic commit order. As Figure 7(a) illustrates, speculation allows a quantum to run concurrently with other quanta in the system as long as there are no overlapping memory accesses that would violate the original deterministic serialization of memory operations. In case of conflict, the quantum later in the deterministic total order gets squashed and re-executed (2). Note that the deterministic total order of quantum commits is a key component in guaranteeing deterministic serialization of memory operations. We call this system *DMP-TM*.

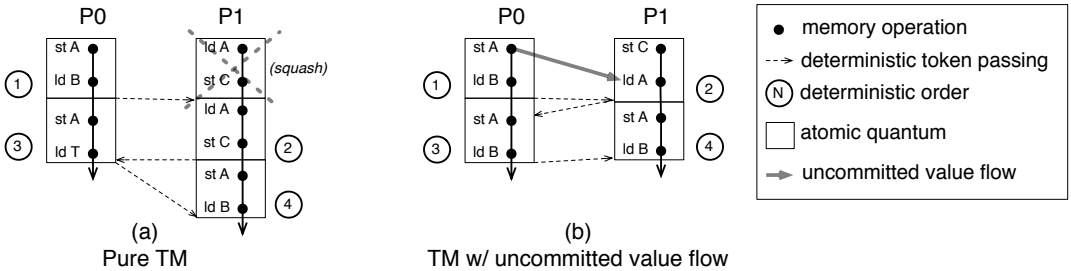

Figure 7: Recovering parallelism by executing quanta as memory transactions (a). Avoiding unnecessary squashes with un-committed data forwarding (b).

We can go further than just allowing quanta to execute atomically and in isolation in parallel with other quanta. Having a deterministic commit order allows isolation to be selectively relaxed, enabling uncommitted (or speculative) data to be forwarded between quanta. This can potentially save a large number of squashes in applications that have more inter-thread communication. To do so, we allow a quantum

to fetch speculative data from another uncommitted quantum earlier in the deterministic order. This is illustrated in Figure 7(b), where quantum 2 fetches an uncommitted version of A from quantum 1. Note that without support for forwarding, quantum 2 would have been squashed. To guarantee correctness, if a quantum that provided data to other quanta is squashed, all subsequent quanta must also be squashed since they might have consumed incorrect data. We call a DMP system that leverages support for TM with forwarding *DMP-TMFwd*.

Another interesting effect of pre-defined commit ordering is that memory renaming can be employed to avoid squashes on write-write conflicts. For example, in Figure 7(a), if quanta 3 and 4 execute concurrently, the store to A in (3) need not squash quantum 4 despite their write-write conflict.

2.4 Preliminary Evaluation

We now provide a preliminary evaluation of both hardware and software implementations of a DMP system based on the techniques discussed so far. For the hardware experiments, or Hw-DMP, we developed a simulator using PIN [36]. Our simulator monitors the execution of an application on a real system, builds quanta based on the instructions that are executed, and then computes a schedule for their execution on the various DMP system implementation choices. For software-only DMP experiments, called Sw-DMP, we developed a compiler pass for LLVM v2.2 [30]. The compiler pass inserts code that implements the various DMP techniques, which potentially requires instrumenting every single memory operation. We used applications from the SPLASH2 [62] and PARSEC [12] benchmarks suites.

We first show the scalability of the hardware implementations. Figure 8 shows performance normalized to the nondeterministic parallel baseline for 4, 8 and 16 threads. As one would expect, Hw-DMP-Serial exhibits slowdown nearly linear with the number of threads. The degradation can be sublinear because DMP affects only the parallel components of an application's execution. Hw-DMP-ShTab has 38% overhead on average with 16 threads, and in the few cases where Hw-DMP-ShTab has larger overheads, Hw-DMP-TM provides much better performance. For an additional cost in hardware complexity, Hw-DMP-TMFwd, with an average overhead of only 21%, provides a consistent performance improvement over Hw-DMP-TM. Hw-DMP-ShTab and the TM-based schemes all scale sublinearly with the number of processors. The overhead for TM-based schemes is flat for most benchmarks, suggesting that a TM-based system would be ideal for larger DMP systems. Thus, with the right hardware support, the performance of deterministic execution can be very competitive with nondeterministic parallel execution.

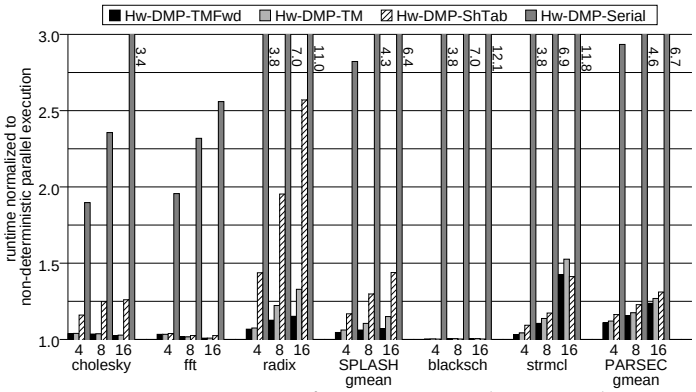

Figure 8: Runtime of Hw-DMP with 4, 8 and 16 threads relative to nondeterministic parallel execution.

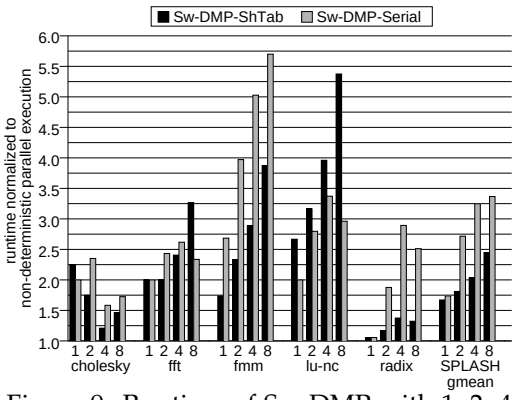

Figure 9: Runtime of Sw-DMP with 1, 2, 4 and 8 threads relative to nondeterministic parallel execution.

Figure 9 shows the performance and scalability of Sw-DMP-ShTab and Sw-DMP-Serial compared to the parallel baseline. As expected, the performance of Sw-DMP-Serial degrades with the number of threads. With the exception of `lu-nc`, all benchmarks recovered enough parallelism to overcome the overhead of tracking memory references. Interestingly, for Sw-DMP-ShTab, we see two classes of trends: slowdowns that increase with the number of threads (e.g. `lu-nc`), and slowdowns that do not increase much with the number of threads (e.g. `radix`). Broadly, these are derived from the sharing pattern of the benchmarks and the change in communication-to-computation ratio based on the number of processors. These numbers

show that while there is much room for improvement, Sw-DMP-ShTab is fast enough to be useful for debugging, and potentially even deployment in limited cases.

3 Research Plan

The preceding preliminary research on techniques to build DMP systems, together with our previous work on deterministic replay [40] and concurrency bug survival [35], opened up several exciting ideas. Broadly, this proposal has two main directions: (1) developing architecture and system software support for efficient DMP systems, and (2) exploring the benefits and uses of DMP systems in improving programmability. Figure 10 shows an overview of the research items and their place in the system stack. We first summarize the proposed research items and then elaborate on each. We end with our experimental infrastructure plans.

Software Development	**Compiler/ Binary Instrumentation**	• Supporting deployment of deterministic execution • Supporting better field crash reporting • Supporting bug avoidance • Supporting testing • Supporting interference-free debugging
System	**OS**	• Understanding and reducing sources of non-determinism • Resolving issues in executing kernel code deterministically • Mitigating the effects of inherently non-deterministic I/O
Architecture	**ISA**	• Making DMP state visible to software • Conveying deterministic token order information and quantum creation • Simultaneously supporting multiple deterministic processes
	Hardware	• Mechanisms for supporting quantum creation and deterministic token passing • Supporting multiple determinism domains • Reducing the performance impact of deterministic execution

Figure 10: Proposed research items and their place in the system stack.

Architecture. We will investigate the right balance between hardware and software support, using basic mechanisms in hardware and policies in software. One key goal is to develop a rich hardware/software interface for DMP systems. Also, preliminary results in Section 2 showed that the performance impact of a DMP system is on the order of 20%. Ideally, we would like to bring this number down to 0%. Our current ideas to do so include exploring the use of information about sharing behavior, synchronization and thread progress to further reduce the performance impact of determinism.

System Issues. OS and I/O pose special challenges to deterministic execution since several system calls are nondeterministic, executing OS speculatively is typically problematic, and some forms of I/O are inherently nondeterministic. We will address the OS challenges by: (1) investigating the sources of nondeterminism in each syscall and attempting to reduce them, and (2) exploring ways to execute OS code on a DMP system without relying on speculation (e.g., DMP-ShTab), while allowing user code to execute speculatively (e.g., DMP-TM). Finally, there is some fundamental nondeterminism in I/O, such as networks and interface with humans, that must be addressed. We will investigate the use of extra synchronization to mitigate the effect of fundamental nondeterminism in DMP by exploring partial determinism.

Support for Software Development (Programmability). We will explore how DMP systems can enhance the software development process, from debugging to testing to deployment. Debugging instrumentation might change the behavior of a parallel program even when running on a DMP system. We will develop mechanisms to avoid this interference. On the testing front, we will leverage DMP system mechanisms to provide support for effective testing of parallel programs. They key aspect in this item will be support for adaptive testing, which will be able to monitor a parallel execution and steer it to more interesting testing scenarios. Our vision includes deploying deterministic execution in the field. Therefore, we will investigate the challenges and opportunities of deploying DMP systems, such as portability, or making a program behave the same way in different DMP systems. One opportunity is to develop better crash reporting mechanisms for parallel code running deterministically in the field. Finally, we will explore the use of DMP

mechanisms to avoid concurrency bugs in legacy binaries by watching for dangerous interleavings and routing the deterministic token in a way that avoids exposing the bug.

3.1 Architecture

The architecture design space ranges from pure software to pure hardware implementations. However, our focus will be on hybrid approaches, with the key mechanisms in hardware and policies in software. Our goal is to develop low complexity hardware mechanisms with a generic HW/SW interface to enable the range of applications we describe later in this section.

3.1.1 Mechanisms

A DMP system has two main mechanisms, one to break execution into quanta and one to enforce the semantics of deterministic serialization. We address our plan for each below.

Quantum Building. The only requirement to break threads into quanta is that quanta boundaries must be produced deterministically. This process can be done via hardware or software. We will explore both alternatives. One challenge of a hardware-only approach is making the process deterministic, so it has to rely only on non-speculative events. A software solution would require code changes to make use of the DMP system. We will explore both compiler and binary-instrumentation software mechanisms to build quanta. The software approach has the advantage of better portability across different DMP systems.

Deterministic Serialization Enforcement. We plan to implement the sharing table (DMP-ShTab) in hardware by leveraging a standard MESI [45] coherence protocol and reuse the cache line state to determine the sharing status of data. Challenges include guaranteeing deterministic transition of the states in the sharing table. For example, keeping track of the state of lines not present in any cache and making sure speculative instructions do not change the state of the sharing table nondeterministically.

We will develop a DMP-TM system by building on top of the multiple alternatives of TM support recently developed by the architecture community [20, 22, 41, 49]. We must additionally provide the ability to enforce a particular commit order of transactions. DMP-TMFwd requires more elaborate TM support to allow speculative data to flow from uncommitted quanta earlier in the deterministic order. We will implement this by making the coherence protocol aware of the quanta's data version, as in versioning protocols used by Thread-Level Speculation (TLS) systems [18, 19, 29].

3.1.2 Hardware/Software Interface

To make the DMP system as flexible and versatile as possible, we need to develop a rich hardware/software interface that allows the software to control and monitor the system behavior. We plan to investigate ISA extensions to: (1) convey quantum boundaries to the DMP hardware, (2) control the order of deterministic token passing, (3) group threads into deterministic domains, with each domain having its own deterministic token, and (4) expose the sharing table state in DMP-ShTab to software.

Item (3) allows multiple independent deterministic processes to execute simultaneously but not be deterministic with respect to each other. This also opens up the possibility of forming deterministic groups of threads within a process, very useful in exploiting the benefits of deterministic thread interleaving even in applications that are inherently nondeterministic, such as reactive applications (e.g, a web server).

In addition, we will investigate necessary ISA extensions that allow the operating system to multiplex the use of DMP mechanisms, i.e., to make the DMP mechanisms virtualizable. Finally, we will investigate the implications of quantum boundaries in memory consistency models.

3.1.3 Reducing the Performance Impact of Deterministic Execution

As we have seen in Section 2.4, the performance cost of deterministic execution remains relatively high, with a geometric mean of 20%. We foresee no fundamental barriers to creating DMP systems that are as fast as nondeterministic ones, and we intend to demonstrate that. The key to reducing the performance impact is lengthening the critical path of the execution as little as possible. We will improve DMP system

performance by investigating better ways to build quanta and relaxing the round-robin token passing order to avoid unnecessarily lengthening an application's critical path.

Using Data Sharing and Synchronization Information. We have experimented with quantum building based on naive instruction count and following synchronization operations. Our initial experiments showed that following synchronization improved performance about 10% compared to naive quantum building. This indicates that attempting to identify the critical path in the application's execution and creating quanta accordingly can be very profitable. Data sharing is likely to be a good indicator of the critical path in multi-threaded applications as well because a thread is likely to either become or stop being critical when it communicates with other threads. Based on that intuition, we will explore using information about data sharing and synchronization to develop heuristics to better form quanta. We will also investigate enforcing a non-round-robin order of the deterministic token based on sharing information. Note that whatever heuristics developed will still need to yield a deterministic decision of quantum boundaries and a deterministic token passing order.

Deterministically Estimating Thread Progress. Threads are unlikely to have the same rates of progress because their IPCs are not likely to be the same. This is one more reason why uniform quantum size and strict round-robin token passing might be suboptimal. We will develop techniques to determine thread progress without relying on elapsed time, which would be nondeterministic. Our approach will be to analyze data dependences and types of instructions in the code to determine the likely rate of progress of a thread and then chunk or pass the deterministic token accordingly.

3.2 System Issues

The operating system interface and I/O present special challenges to deterministic execution. For example, several system calls behave nondeterministically, and I/O can be inherently nondeterministic. We will address these challenges as follows.

Deterministic POSIX-compatible Interface. We will investigate the sources of nondeterminism in each POSIX system call and attempt to reduce them, e.g., making `read()` always return the same amount of data. The ultimate goal is to create a deterministic POSIX-compatible interface.

Executing an OS Deterministically on a DMP System. We will explore ways to execute kernel code on a DMP system. An interesting challenge here is that parts of the operating system code cannot be executed speculatively. Fortunately, DMP-TM, DMP-ShTab and DMP-Serial can coexist in the same system. One easy way to implement co-existence is to switch modes at a deterministic boundary in the program (e.g., the boundary of a quantum). Alternatively, we will explore how to have multiple modes co-exist simultaneously. This would let the system choose the most convenient approach depending on what code it is running. For example, a DMP system could use speculation (DMP-TM) in user code and avoid it (DMP-ShTab) in kernel code.

Dealing with Inherently Nondeterministic I/O. Ultimately, programs interact with remote systems and users, all nondeterministic factors that might affect thread interleaving. As a result, it may appear that there is no hope of building a deterministic multiprocessor system. This is not the case. Synchronization in multithreaded code (e.g., via barrier or locks), provides an opportunity to be deterministic from that point on since the threads involved in the synchronization are in a known state. Once the system is deterministic and its interaction with the external world is considered part of the input, it is much easier to reason about system behavior and to debug and deploy reliable software. This will likely encourage programmers to insert synchronization around I/O in order to make their applications more deterministic and hence more reliable. We will investigate the use of extra synchronization to mitigate the effect of inherent nondeterminism.

3.3 Support for Software Development (Programmability)

Below we describe in more detail our plan on leveraging DMP systems over the lifetime of multithreaded software development. We start with debugging and then go on to testing, deployment, crash reporting and finally bug avoidance, which aims at increasing reliability of deployed code.

3.3.1 Interference-free Deterministic Debugging of Parallel Programs

A DMP system is directly useful for debugging parallel programs because it offers repeatability by default if the program is given the same input and the code remains the same. However, to make debugging useful, we need a way to preserve the interleavings of the original execution while still allowing the user to instrument the code for debugging. To accomplish this, we will develop mechanisms that allow a software development system to mark the instrumentation code inserted and ensure that it does not affect the interleaving of the original program.

Our preliminary evaluation of DMP using a simple first-order simulator found that a sharing table implementation at the granularity of pages performs reasonably well. This suggests that it is possible to use page-level protection mechanisms to implement a hybrid hardware/software version of DMP-ShTab. Doing so will yield a debugging environment that provides deterministic execution in current off-the-shelf systems more efficiently.

The sharing table implementation can be easily extended to hold information about the instructions that wrote the data. This would allow precise monitoring of communicating pairs of instructions in an execution. We will explore how to analyze this information to find bugs. For example, we could mine the communication graph for uncommon behavior [21, 55].

3.3.2 Effective Testing of Parallel Programs

When testing a multithreaded program [17, 42, 47], it is insufficient to test only one program input; it is also necessary to test *multiple different interleavings*. Hence, testing multithreaded programs needs nondeterminism, as opposed to determinism. Interestingly, the same DMP primitives used to control memory interleaving for determinism can be leveraged to efficiently support software testing. We will investigate how to use the DMP hardware/software interface to control quantum formation and deterministic token passing to make testing more effective. In addition, we will explore using a software-visible sharing table implementation to watch the interleavings and resulting inter-thread communication. This will allow the testing infrastructure to explore the testing space more efficiently than exhaustively and consequently find concurrency bugs faster. Similar approaches for testing have been explored only in software [43], but their applicability was limited due to state explosion and low performance. In contrast, leveraging a DMP system for software testing would avoid the performance problem by using hardware mechanisms; it would avoid the state explosion problem by enabling more adaptive testing techniques. In summary, we believe a DMP system is an ideal architecture to test multithreaded code as it can enable both fine grain control of thread interleaving *and* provide reproducibility when a bug is found.

3.3.3 Deployment of Deterministic Execution and Better Crash Reporting

We contend that deterministic systems should be used not just for development, but for deployment as well. Systems in the field should behave similarly to systems used for testing for two primary reasons. First, developers would be more confident that their programs will work correctly once deployed. Second, if the program did crash in the field, then deterministic execution would provide a meaningful way to collect and replay crash history data.

Supporting deterministic execution across different physical machines places additional constraints on the implementation. Quanta must be built in the same way across all systems. This means that machine-specific effects cannot be used to end quanta (e.g., a full cache-set for bounded TM-based implementations). Furthermore, the deterministic token passing order among processors must be the same across all systems. This suggests that DMP hardware should provide the core mechanisms, while software should control quanta building and deterministic token scheduling. We will investigate the requirements for portable determinism.

Deterministic execution in the field has the potential to improve crash reports of multithreaded code. A checkpoint can be taken by holding the deterministic token and therefore not allowing other threads to proceed. This would be equivalent to taking a checkpoint at a point in a sequential execution. In addition, since the threads interleave deterministically, we can potentially reconstruct the recent past of the execution prior to the crash. We will investigate the synergy between crash reporting and execution in a deployed DMP system.

3.3.4 Concurrency Bug Avoidance in Legacy Binaries

Concurrency bugs manifest themselves when a particular interleaving of memory operations occurs. An alternate way of looking at the problem is that a bug manifests itself when incorrect synchronization causes a particular erroneous communication between threads. By avoiding the particular interleaving or inter-thread communication, it is possible to prevent the bug from crashing the program in the field [48, 52, 64]. Again, the same DMP primitives used to control memory communication for determinism can be leveraged to accomplish bug avoidance. However, this case is the opposite of testing, since the goal of bug avoidance is to prevent the bug from manifesting itself.

We will explore ways to monitor the dynamic communication between threads to detect when a bug is likely to occur. When such a situation is detected, the system will manipulate deterministic token passing order and quantum boundaries to prevent the potentially dangerous memory communication. For detection, we will build on top of our prior work on bug avoidance using dynamic atomicity [35] and on recent studies on the common patterns of concurrency bugs [34]. Our goal is to leverage DMP mechanisms to be able to survive a broader category of bugs more effectively. This opens up an interesting space of architecture research: designing systems that more reliably execute faulty software.

3.4 Experimental Infrastructure

The experimental infrastructure we will integrate includes a detailed architecture simulator (and possibly an FPGA prototype) and system software, i.e., binary instrumentation, compiler and operating system. We will build our architecture simulator on top of SESC [50], a fast, detailed multiprocessor simulator available to the community as open-source. I have been involved in the development of SESC since I was a junior graduate student and I am now one of its SourceForge project managers. Our architecture experiments will also possibly include an FPGA prototype leveraging the RAMP infrastructure [8]. Prof. Mark Oskin, my colleague and close collaborator at UW-CSE, is one of RAMP's founding members and has extensive experience with using FPGAs for architecture exploration. For code transformations, we will use the PIN binary instrumentation framework from Intel [36] and the LLVM compiler infrastructure [30] from the University of Illinois. PIN and LLVM are very mature frameworks; my graduate students, collaborators and myself have significant experience with both.

We will use Linux as the experimental OS platform. I also have significant experience with this OS, dating back to my time at IBM Research working on the Blue Gene/L project, when I helped port it to Blue Gene/L's processor architecture. As workloads, we intend to use real-world software, such as MySQL, Firefox, Apache, as well as emerging multicore benchmark suites such as PARSEC [12].

The architecture group at UW-CSE has a large cluster available to run experiments. In addition, I have recently received gifts from Intel and Microsoft to further improve this infrastructure.

4 Education Plan

An integral part of the concurrency challenge is educating those students and professionals who will be writing threaded code for games, servers, desktops or even embedded systems. Computer science students at all levels must understand concurrency principles and be proficient parallel programmers in several different problem domains, models and languages. *The high-level educational goal of this proposal, then, is to develop the curriculum at the University of Washington to teach students how to "think parallel"*. In addition, this proposal also includes plans to develop tools to assist in teaching concurrency to students without experience in parallelism.

The core target audience of this education plan is undergraduates, graduate students and professionals from the local technology industry. At the undergraduate level, the focus will be pragmatic but it will certainly include the basic principles. At the graduate level, the focus will be on research issues, with the goal of enabling students to become successful researchers and educators in the area. The target audience will also include students enrolled in the Professional Masters Program (PMP) at the University of Washington's Computer Science and Engineering Department. PMP is a well-established program that caters to professionals with 5-6 years of experience in local companies, such as Amazon, Microsoft and Boeing. Overall, the core target audience forms a unique and diverse body of students with vastly different backgrounds.

The key philosophy in the educational plan in this proposal is deep integration of research into teaching. This is especially natural in the problem space of this proposal, the concurrency revolution. As we experiment with new models and systems, we both teach students and learn the programmability benefits of research ideas. Specifically, classes will include projects with research components such as keeping track of concurrency bugs, or decomposing non-trivial algorithms into parallel tasks, etc. Such research significantly increases students' understanding of the core concepts being taught and also opens further research opportunities.

The following subsections describe ideas on how to teach students to think parallel and provide a more detailed curriculum development plan.

4.1 Thinking In Parallel

Broadly, writing a parallel program involves three major steps [33, 37]: decomposing the problem into units of parallel work, mapping these units of work to the underlying computing devices, and synchronizing (or coordinating) the parallel tasks. Decomposing the program into tasks (or threads) requires either decomposing the application at the architecture level (into coarse blocks, e.g., GUI and back-end or producer and consumer) or thinking algorithmically in parallel. Both require deep abstract reasoning. We plan to address the inherent difficulties in this abstract reasoning by employing visual tools [28, 57] to aid our students in understanding the subtleties of parallelism.

Understanding Shared Memory Parallel Execution. After decomposition, mapping and coordinating parallel threads require a solid understanding of the machine model and semantics of synchronization operations. Since the most prevalent model is shared memory, it will be the initial focus of our efforts. A key component in this model is memory interleaving, which determines the communication that actually occurs between threads. Two factors that complicates one's understanding of memory interleaving are the intricacies of the memory consistency model (the allowed reordering of memory operations) and nondeterminism. We plan to address these factors by developing a tool that lets students visualize the flow of data between memory instructions (or coarser portions of a program execution). In fact, we have already developed a prototype of this tool using the Prefuse visualization toolkit [7]. Synchronization between threads in a shared memory machine occurs via operations with shared memory [38]. Therefore, understanding memory consistency [10] and memory interleaving will provide a strong foundation for understanding synchronization operations.

Teaching undergraduate students the basic concepts of parallelism and shared memory machine models gave me a first hand understanding of the difficulties students have in understanding parallel execution. For example, during one class, I showed two pieces of code with multiple accesses to shared memory and asked the students to enumerate all possible outcomes. Several outcomes were missed because students tended to think in terms of coarse-grain interleavings and missed the cases involving finer-grain ones.

Non-Local Reasoning. When inserting synchronization in parallel code, the reasoning the programmer needs to perform is non-local [14, 60]. A programmer writing a piece of synchronization code must constantly keep track of other parts of the code that might be interacting with it, ensuring that all related critical sections are compatible. Consequently, to avoid bugs, programmers must simultaneously consider multiple parts of the code to ensure the proper invariants are being held in all possible interactions between threads. We will investigate tools that help the programmer with non-local reasoning by simultaneously displaying related parts of the code on the screen.

Debugging Tools. Since writing parallel code is inherently more error prone and harder to optimize than single-threaded code, it is important to teach students what debugging tools can do and how to use them. Specifically, we plan to use tools for deterministic debugging (using our Sw-DMP prototype), data-race detection and performance profiling (potentially using Intel's Threading Analysis Tools).

4.2 Curriculum Development

Undergraduate Classes. During my first quarter at the University of Washington, I taught Computer Organization and Assembly Language (CSE378), a class based on the Patterson and Hennessy book [46]. I reserved the last week of classes to discuss multicores and the basic principles of concurrency at the

architecture level — basic cache coherency and synchronization. At the end of the class, many students came to me interested in doing research on multicores, and I took 3 undergraduate students to work with me; one recently graduated and his honors thesis was based on multicore research.

Seeing the enthusiasm of the students when learning about parallelism encouraged me to begin thinking of ways to expand the undergraduate curriculum to include parallel programming. A top-level decision in this curriculum extension is whether parallelism concepts should be discussed in multiple current classes or contained in a few stand-alone classes. I would encourage a mix of both. The two Introduction to Programming classes at UW-CSE use Java (CSE142 and CSE143 [2, 3]), which provides an opportunity to introduce some parallel programming concepts with Java threads. The natural progression is to then introduce the concept of data sharing between threads in the data structures class (CSE326 [5]). Our current Introduction to Operating Systems (CSE451 [6]) and Concepts and Tools for Software Development (CSE303 [4]) classes currently include concepts on synchronization and inter-process interaction. Finally, I plan to collaborate with Prof. Larry Snyder to create a new class, Thinking Parallel, on parallel programming in modern multicore architectures. The class will have a pragmatic focus and will include Java Threads, C/C++ with Pthreads, OpenMP and Intel Thread Building Blocks. We will also address the use of tools for debugging and performance tuning.

Graduate Classes. During the Winter quarter of 2008, I taught a graduate-level Computer Architecture class that focused on parallel computer systems. The class included a project related to multicores and a strong exploratory research component. The positive feedback from the class reinforced my belief that doing research is a powerful way to learn complex concepts. During the Spring quarter of 2008, I taught a Programmability of Multicores class that focused on recent research. This class was also a success, as students from multiple areas enrolled, and the class had very insightful discussions that generated multiple research project ideas. While I plan to offer these two courses again, I will also create a new class, Parallel Programming on Modern Multicore Architectures. In this new class I plan to experiment with new tools to aid with parallel reasoning and new forms of concurrency bug detection. In addition, the funds from this grant will be used to fund graduate students and permit them to attend relevant conferences, give talks and gain exposure in the research community.

4.3 Broader Audience and Outreach

As mentioned earlier, the core audience of this proposal's educational goals also includes professionals from the local technology industry. Furthermore, we plan to entice high-school teachers and students into learning about concurrency and parallel programming. Some earlier experiences from the computer science community have been quite successful. These experiences ranged from using games as a technique to teach concurrency concepts [26], to teaching high-school students about parallel programming [51] and monitoring the evolution of students' understanding of synchronization [27]. Interestingly, one of the special challenges in teaching concurrency is precisely the nondeterministic nature of parallel systems [11], which is what the research component of this proposal aims to solve.

I am enthusiastic about teaching concurrency concepts to high-school students, especially since such efforts provide opportunities to understand how "fresher" minds absorb these concepts and approach parallel programming; this might lead to new ideas on more convenient models of expressing concurrency. I will visit high-schools to give special lectures and use the UW CoE Engineering Open House [9] to get students excited about learning more.

In addition to teaching and research, promoting diversity and tolerance in society is a central role of universities. The University of Washington's Minority Science and Engineering Program (MSEP) helps to bring in historically underrepresented minorities to the engineering and science fields. I will integrate interested undergraduate students in this program into components of this proposal. I am committed to continuing to attract and retain women and minorities to computer science both at the undergraduate and graduate levels. I have advised one (of 3) female undergraduate student, and I am in the process of involving another one in my research group. I also plan to use the Engineering Advising and Diversity Center as a resource.

My personal interests in geographic diversity relates to my home country of Brazil. While Brazil's economy has been improving steadily, most of its population is very poor by American standards. However, Brazil has very good engineering schools with many students eager to have impact in the world. Being born

and raised in Brazil, I have several personal contacts with faculty members in the best Brazilian engineering schools (USP, Unicamp, among others). I intend to foster higher CS education in Brazil by giving talks on the opportunities of doing cutting edge research and also by attracting the best graduate students from Brazil to our programs at UW-CSE. My goal is to make these students world class leaders in technology, capable of benefiting the US economy, the CS research community and, in the long-term, Brazil's society as well.

Finally, I maintain many contacts in industrial research labs, especially IBM Research, where I worked before going to graduate school and where I spent 3 summers during my time at UIUC as a Ph.D. student. I also have contacts at industrial research labs in the Seattle area, such as Microsoft Research and the newly formed AMD Research and Advanced Development Labs. I plan to continue strengthening my ties with industrial research, as industry is both a consumer and motivator of good research.

5 Work Plan and Long-term Career Goals

This section outlines the research and education milestones and their integration for the five-year duration of this proposal. It also includes my long-term career goals.

Year 1. Define the general DMP system architecture in detail; this includes understanding the balance between hardware and software support and designing the ISA. Devise optimizations to decrease the performance impact of determinism. Begin promoting more undergraduate interest in multicore systems and parallel programming. Offer a graduate class with projects geared towards understanding and improving programmability of multiprocessors. Explore opportunities to include concurrency topics in undergraduate classes currently offered at UW-CSE.

Year 2. Develop a detailed simulator of a DMP system and potentially implement some of its features in FPGAs using the RAMP [8] infrastructure. Create the low-level software stack used to form quanta, and control the deterministic token for multiple processes; this includes both binary instrumentation and compiler passes. Start exploring the operating system implications and opportunities in DMP systems. Understand and mitigate the impact of I/O in thread interleaving of commercial code, such as servers and browsers. Plan the Thinking Parallel undergraduate class with Prof. Larry Snyder and offer it during the Spring quarter.

Year 3. Develop debugging tools for DMP systems. Start exploring the deployment of deterministic execution and support for better crash reporting. Evaluate the graduate and undergraduate offerings. Offer the graduate class on pragmatic parallel programming with Prof. Larry Snyder. Refine the undergraduate class based on students' feedback and the quality of the class projects.

Year 4. Investigate the use of the DMP infrastructure for software testing; this may require adjustments to the ISA to conveying the ordering patterns used for testing. Make simulator and software-only version of DMP available to the community, allowing users to both debug parallel code and do research on DMP systems. Perform user study on the effects of determinism in parallel software development. Organize workshop on multiprocessor programmability, possibly in conjunction with ASPLOS.

Year 5. Develop support for dynamic concurrency bug avoidance using the DMP infrastructure, potentially integrating it with an open-source VM environment such as Xen. By now we should have a better understanding of the implications of determinism in parallel software development and deployment. Graduate first Ph.D students.

Long-term Goals. I intend to devote much of my career to investigating architectures that make it easier to write efficient and reliable software. My research interests are in the hardware/software interface, conceptual mechanisms and their design trade-offs. I believe that an integral part of these goals is to inspire, educate and mentor undergraduate and graduate students. This proposal described one of the major steps towards my long-term goals.