S-Kendo: Deterministic execution for arbitrary programs

Gwendolyn Voskuilen
Purdue University
geinfeld@purdue.edu

Balajee Vamanan
Purdue University
bvamanan@purdue.edu

ABSTRACT
The emergence of Chip Multi-Processors (CMPs) and the continued scaling of the number of cores per chip necessitates that applications be parallelized to distribute work efficiently to different cores. However the deployment of shared-memory based multi-threaded software is slow. As a result application performance has not matched the performance offered by today's CMPs. While it is true that substantial effort is required to develop parallel counterparts to existing sequential algorithms, many applications with well known parallel algorithms have yet to be implemented as multi-threaded applications[1]. We argue that the effort involved in debug-ability and test-ability of parallel programs is the main reason for their slow adoption. The lack of determinism in the order of memory accesses from different threads of a parallel program complicates debug-ability and test-ability to a large extent. In this paper, we address this problem by proposing a software-only solution to restore deterministic execution for arbitrary shared-memory based multi-threaded programs. Our idea is to use a static race detector in combination with a set of library functions that guarantee determinism when all shared locations are protected by locks. Our experimental results show that we can achieve deterministic execution without hardware support for programs with memory races.

Categories and Subject Descriptors D.1.3 [Programming Techniques]: Concurrent Programming – Parallel Programming; D.2.5 [Software Engineering]: Testing and Debugging – Debugging Aids; D.4.1 [Operating Systems]: Process Management – Synchronization

General Terms Performance, Design, Reliability, Synchronization

Keywords Deterministic execution, MultiCore, Memory race, Multithreading

1. INTRODUCTION

Technology scaling provides exponential increase in transistor density with every subsequent generation. In the superscalar era, computer architects utilized these extra transistors to increase clock frequency (deeper pipeline) and to increase instruction level parallelism (wider instruction window). As a result, sequential applications scaled in performance to match the hardware without much programmer effort. However this trend has not continued because of two major reasons:

Power Wall: The dynamic power dissipated by a transistor is directly proportional to the square of clock frequency. Hence it is not possible to continue increasing clock frequency indefinitely.

ILP Wall: Larger instruction windows lead to improved application performance only if there are enough independent instructions that can be issued out of order. Unfortunately studies have shown that there is not enough ILP present in most applications[2].

The hardware community worked around the power wall by building Chip Multi-Processors (CMPs). However to harness the performance provided by multi-core machines, existing sequential programs need to be written as parallel programs, using multiple threads on different cores and allowing inter-thread communication via shared-memory. Thus, unlike sequential programs where performance is largely decoupled from the programming model, CMPs tightly couple the programming model with performance.

A key issue in multi-core processors is that, the order of accesses to a memory location from multiple threads depends on the actual timing of low level hardware events (like bus grant). A correct multi-threaded program requires proper synchronization primitives to order memory accesses across threads to produce the desired output. Hence, correctly synchronized (not just serialized) multi-threaded programs produce deterministic output while incorrectly synchronized programs produce non-deterministic output. Traditionally programmers relied on repeatability of program results to incrementally debug the program. Because of the lack of determinism in incorrectly synchronized multi-threaded programs, incremental debugging is time-consuming and often, nearly impossible. Additionally, because thread interleavings can vary and only some interleavings lead to bugs, test coverage is more difficult to assess. This leads to an increase in testing time and cost, and a simultaneous decrease in test reliability[3].

The debugging, reliability and testing challenges can be addressed by ensuring that accesses to shared-memory
locations from different threads happen in a deterministic order. Note that often not all possible memory interleavings affect program output. Ensuring a deterministic order for all shared-memory locations by serializing accesses to every shared locations can severely degrade performance[4]. To overcome this performance penalty, substantial hardware support is required for tracking, intelligent conflict detection and resolution[5]. In a race free program, all shared data are protected by locks. In such a program, one has only to guarantee that the lock acquisitions and lock releases happen in a deterministic order. Marek Olszewski et.al[3] present Kendo, a scheme to implement deterministic multi-threading in software for such programs without incurring a large performance penalty. However it does not offer any deterministic guarantees to programs with memory races. We observe that a static race detector can identify memory locations that are possibly accessed by different threads without being protected by locks[6]. We propose a software only solution to guarantee deterministic execution for all programs (including programs with memory races) using a combination of a static race detector and thread library functions to deterministically order lock acquisitions and lock releases by different threads. We make the following contributions in this paper.

1. Implement a deterministic threading library on top of POSIX Threads API, which can be used to deterministically execute any race free program.
2. Modified an existing static race detector to use our deterministic thread library functions instead of POSIX thread library functions.
3. We merge (1) and (2) and provide a software flow to run any program deterministically.
4. We evaluate our scheme with a mix of scientific and micro benchmarks, as well as a real world application.

The paper is organized as follows. We discuss related work in section 2. In Section 3 we lay the foundation for the paper by describing determinism. Section 4 discusses [3] in which we our paper is based on. Static race detection is explained in section 5. We provide details of our implementation in section 6 and our evaluation in section 7 before concluding the paper.

2. RELATED WORK

Many researchers in the community have looked at the problem of non deterministic interleaving of shared-memory accesses from different threads. Record and replay systems[7] [8] do not guarantee any deterministic order of shared-memory accesses from different threads during a "record" run. Instead, they record the observed interleaving of these accesses. One has to re-run the program using a software replayer to reproduce the program output. The replayer uses the recorded information to reproduce the observed order of shared-memory accesses. The replayer incurs significant performance penalty, and hence can only be used during debugging. This severely restricts the applicability of record-and-replay systems in the case of software testing (testing time increases greatly) and software reliability (multiple replicas of the same program are run to identify the faulty run).

[4] achieves deterministic execution, albeit by serializing all the threads in a deterministic order whenever they access a shared-memory location. Threads alternatively switch between parallel and serial modes of execution. During a parallel mode of execution, those threads that want to access shared data have to wait for other threads to finish so they can enter serial mode. Hence a long section of parallel mode causes starvation of these threads, and degrades load balance. On the other hand, shorter sections of serial and parallel sections force unnecessary serialization. To identify shared data from private data, accesses have to look up a table. This lookup incurs substantial performance overhead, since lookups are frequent and the table could be big.

The Kendo paper [3] upon which we build our idea, is based on the key observation that in race free programs, one can guarantee deterministic execution by only guaranteeing the order of lock acquisitions and lock releases across threads. However they do not address the problem of deterministic execution of programs with memory races, which is very common. Since we build our ideas on top of Kendo framework, we provide a relatively detailed overview of it in Section 4.

3. DETERMINISTIC EXECUTION

3.1 TYPES OF DETERMINISM

Thread communication through shared-memory can be interleaved in a deterministic manner in order to restore the determinism guarantees provided by sequential programs. Marek Olszewski et.al[3] define this property as deterministic multithreading, and classify it into the following two categories

Strong determinism: ensures a deterministic order of all memory accesses to shared data for a given program input.
Weak determinism: ensures a deterministic order of all lock acquisitions for a given program input.

Hence the determinism guarantees provided by weak determinism are the same as those provided by strong determinism if a program is race free. Marek Olszewski et.al[3] argues that providing strong determinism requires substan-
tial hardware support. Instead they guarantee deterministic execution for race free programs using a set of thread library functions (a software only solution).

We take a different approach from [3]. We argue that one can provide strong determinism for any program using a combination of static race detection and a collection of thread library functions, without needing to use expensive hardware.

Please refer to section 6 for details of our proposed solution that provides deterministic execution for any program.

4. THE KENDO FRAMEWORK

In this section, we give a brief overview of the kendo framework[3]. Kendo ensures deterministic execution for race free programs. The paper makes the key assumption that in a race free program; only lock acquisitions and lock releases need to be made deterministic.

4.1 Logical time

Nondeterminism in multi-threaded programs arises because memory accesses from different threads are ordered based on the exact timing of low level events such as bus grant, DMA etc. To achieve deterministic execution, we need to eliminate such dependence on low level events. A Deterministic logical time is created for each thread from a monotonically increasing deterministic logical clock, which is updated independent of other threads and other nondeterministic events. The kendo framework[3] uses logical time as the fundamental entity based on which lock acquisitions and lock releases are made.

Logical time for a thread is derived based on the number of store instructions committed by each thread. Since the number of stores committed by each thread does not depend on any non-deterministic information, logical time is repeatable from run to run. The thread library maintains the logical time of all threads in shared-memory and so is accessible to all threads.

4.2 Deterministic Thread Library

Threads do not look at their logical time while they are executing out of their critical sections. Whenever a thread wants to acquire a lock, it can acquire the lock only if it has the lowest logical time among all other threads. Thread IDs are used to break the tie between threads with the same logical time.

Thread IDs are issued in a deterministic order based on the logical time i.e. thread creation order is deterministic. Since threads acquire locks in a deterministic order, they also release locks in the same deterministic order i.e. lock releases are trivially deterministic. Apart from these basic thread library functions, any other function like malloc() that can nondeterministically modify the logical time need to be handled. In this case, a simple solution is to disable the logical time before calling such functions. Also functions like rand() need to depend on the logical time of the thread, instead of the actual physical time to make them independent of external events.

We provide a detailed section on implementation of our thread library (which is based on Kendo) in section 6.3.

5. STATIC RACE DETECTION

The Kendo libraries ensure determinism when all shared accesses are protected by locks. To ensure determinism for arbitrary programs, we integrate the deterministic library with the Locksmith[6], a static race detection tool. The Locksmith analysis detects shared data locations and ensures that whenever shared location x is accessed, a corresponding lock m is held. The analysis is context and flow sensitive to reduce false positives.

The first step of the Locksmith algorithm is to build a control-flow constraint graph. From this graph, an analysis is done to determine the locks held at each program point. To remove false positives (i.e. only consider locks for data locations that are shared), Locksmith employs sharing analysis. A location is determined to be shared if it may possibly be shared with its parent or any active thread. Additionally, the analysis discards locations dereferenced before the first thread fork and variables that do not escape threads, as well as locations that are only read among threads (i.e. read-read sharing is not considered a race). The result is a pruned list of locations which may be shared. Finally, Locksmith combines the shared location list with the locks-held results. The intersection of the set of locks held at each reference to a shared location yields either the lock that guards that location or NULL if no lock covers all such locations. In the latter case, Locksmith flags the variable as possibly being involved in a data race.

Locksmith is implemented using CIL[6] as a front end, and Banshee[ref] to facilitate the constraint-graph encoding and some of the analysis on the graph. For S-Kendo we extended Locksmith to recognize the deterministic library functions. This extension was straightforward and involved replacing Kendo's definitions of pthread functions with the corresponding deterministic threading definitions.

6. S-KENDO DESIGN

We propose to use a combination of static race detection and a set of deterministic thread library functions that
guarantee weak determinism for race free programs, to provide strong determinism for any program.

6.1 Tool Flow

We implement a development flow that leads to deterministic execution. The flow is depicted in Figure 1. The static race detector flags possible shared accesses. We use the output of static race detector to protect critical sections with locks. Once this is done, the program no longer contains memory races. The POSIX thread library functions are overloaded by our deterministic thread library functions, so the source code of the program need not change. Once we have a race free program, it can be compiled with our thread library directly. The compiled code can be run deterministically. Unlike coredet[4], our system does not classify accesses from threads as being private or shared at run-time. Instead static race detection is used to classify accesses conservatively. While static race detection is conservative, we believe that classifying accesses at run-time is more expensive and hurts the common case.

6.2 Logical Clock Implementation

Logical clocks are implemented using the performance counters that are available in most modern microprocessors. Specifically we chose the "number of stores committed" as our logical clock, since in X86 architectures this counter excludes interrupts and IO events which are non-deterministic. Since polling on performance counters would be wasteful, we enable interrupts on performance counter overflow. The overflow threshold can be adjusted to trade-off precision and performance. Frequent interrupts degrade performance when there is less communication between threads. On the other hand, frequent interrupts help threads to quickly acquire locks without having to wait for a long.

It is in the interrupt handlers that the logical time of a thread gets updated. The logical time is kept in shared-memory so that all threads can access it. Performance counters can be stopped, and started as required. The thread library functions use such operations to avoid recording non deterministic events such as those caused by IO.

We use PAPI4.0[9] to access these performance counters and to setup interrupt service routines. Hence our library functions call PAPI APIs to access performance counters and to setup interrupt routines.

6.3 Thread Library Functions

In this section we provide the key algorithms behind our thread library functions (which is similar to the kendo thread library). For a detailed pseudo code of these functions, refer to Figure 2.

wait_for_turn()

This function is used to deterministically serialize threads. Threads call this function before accessing shared data - i.e. lock acquisition. Threads also call this function before creating new children threads. It is a blocking function, so the caller has to wait till the logical time of the caller is the minimum of the logical times of all the threads.

lock()

This function is the key to deterministic execution. Threads have to acquire locks in a deterministic order. If we let more than one thread contend for a lock, it is impossible then to guarantee any order of lock acquisition. Hence threads wait-for-turn() before even they try to acquire the underlying POSIX thread library lock. Note that since wait-for-turn() and the lock() calls can take arbitrary number of stores, we disable the logical clock before calling wait-for-turn(). We consider each lock() call as doing one store instruction, so we increment the logical
clock by one. It does not matter how much we increment so long as it is a constant across different runs and threads. The logical clock needs to be enabled before the function can return.

The above mentioned locking algorithm does not handle nested locks properly and can cause deadlock. This can be avoided by imposing the following invariant[3]: only one thread may hold a given lock at a given deterministic logical time. If a lock is free in both real and deterministic logical time, then the thread acquires the lock and exits the function.

unlock()

Since threads acquire the locks in a deterministic order, the order of lock releases is trivially deterministic. However, the unlock() function needs to tag the logical time in the lock data structure to avoid the deadlock we mentioned in the previous section. Since we need to get the current logical clock for tagging, we have to freeze the logical clocks during the process i.e. logical clocks are paused before the function and resumed after the underlying unlock() call.

thread-create()

Whenever there is a tie between threads waiting to acquire a lock, thread IDs are used to break the tie. However, this imposes an additional constraint - threads IDs have to be generated in a deterministic order. The parent thread has to wait-for-turn() before calling the underlying POSIX library thread-create() function. Once the parent thread gets its turn, it assigns a thread ID to the newly created thread and sets up its logical time to be one greater than that of the parent. Finally it calls the underlying POSIX library to actually create the thread before returning.

Other functions

Standard library functions like malloc(), calloc(), memcpy() can do non deterministic number of store instructions. Hence before calling these functions, logical clock has to be paused. After these functions complete, logical clock is resumed. The existing random number generator relies of physical time(seed). This could lead to non deterministic execution because we do not have control over physical time. We modified these functions to use the thread local logical time as seed.

7. EVALUATION

7.1 Methodology

We use PAPI to access performance counters of Intel X86 processors. PAPI is available as a library and provides us with APIs to access, trigger and stop performance counters supported by the processor. It also allows the programmer to set up interrupts on performance counter events like overflow, and register an interrupt handler for the same. PAPI requires performance counter interface support in Linux kernel. This support is included by default in the latest versions of Linux kernel starting from 2.6.31.
We used a 2.11 GHz Intel Core 2 Duo CPU running “Ubuntu GNU/Linux” with kernel version 2.6.31. We used GNU/C Library 2.10 which includes the NTPL/POSIX thread library. We developed our library for deterministic multi-threading on top POSIX thread library functions – our library functions make calls to functions in POSIX thread library. Our library includes PAPI library so we can use the APIs from PAPI library in our library. We use Locksmith, version 0.4.

The set of benchmarks that we used for our evaluation is shown in the Table 1. Aget is a multi-threaded download accelerator. It supports HTTP downloads and can be run from the console. Aget fetches HTTP URLs in a manner similar to wget, but segments the retrieval into multiple parts to increase download speed. It can be many times as fast as wget in some circumstances. LU is a kernel from SPLASH benchmark suite of scientific applications. The LU kernel factors a dense matrix into the product of a lower triangular and an upper triangular matrix. The dense matrix A is divided into an array of blocks ( ) to exploit temporal locality on sub matrix elements. Elements within a block are allocated contiguously to improve spatial locality benefits, and blocks are allocated locally to processors that own them. Microbench is a micro-benchmark we created to simulate the kind of fine grain synchronization observed in database benchmarks. Comparatively it is smaller in size to the other benchmarks, but features more nested locks and synchronization primitives than other benchmarks.

We judiciously select a mix of scientific and commercial applications. Scientific applications usually do not require complex fine grain synchronization primitives like nested locks and exhibit long parallel sections. Commercial applications on the other hand, contain large parallel sections and a small set of synchronization points. Using a mix of applications with differing traits would give us a more complete picture of the problem, and enable us to make meaningful trade-offs.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Type</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aget</td>
<td>Unix utility</td>
<td>3000</td>
</tr>
<tr>
<td>Lu</td>
<td>SPLASH kernel</td>
<td>1000</td>
</tr>
<tr>
<td>Microbench</td>
<td>database</td>
<td>300</td>
</tr>
</tbody>
</table>

Table 1: Benchmarks

7.2 Results

We ran our benchmarks with locksmith to identify potential memory races. Once identified, we protect the racy memory accesses with locks. Please note that this does not make the program execution deterministic (or correct), but only makes the program race free. We take this race free program, and compile it using our library instead of the standard POSIX thread library. This yields an executable which guarantees deterministic execution.

For each benchmark, we compare the performance of the original non deterministic program with that of our deterministic program. The original program is input to locksmith, a static race detector. Locksmith flags all the data in the program that can potentially cause memory races. We then protect all the shared data with locks. After this step, the program is race free. Note that static race detection is conservative, so there may be cases where locks may not be necessary. To quantify the effect of conservative locking, we also create another version of the program with only bare minimum locks needed. Remember that such perfect locking is time consuming and so long as conservative locking does not hurt performance significantly, it is the preferred approach taken by most programmers. As a next step, we compile both the programs – the one with perfect locks and the one with conservative locks using our deterministic thread library. We measure the run time of both these versions with the original program. For our set of benchmarks, the following figure captures the performance degradation because of deterministic execution, and conservative locking. It is quite clear that conservative locking performs as well as perfect locking.

![Figure 3: Performance of S-Kendo](image-url)

We also measure the scalability of our approach with increasing number of cores. The following figure presents
our scalability study. It is evident that serializing lock acquisitions hurts performance with increasing number of threads. Rewriting some of the library functions to use distributed algorithms could improve scalability to a large extent.

Deterministic execution comes at a price. We had to give up the flexibility of acquiring and releasing locks without a predetermined order. This could make threads wait for longer than is otherwise needed. But if this overhead turns out not to be too large, then we argue that it is worth to tradeoff a little performance to ease parallel software development. We also argue that a lot of overhead of the thread algorithms can be reduced by looking at distributed versions of these algorithms.

8. CONCLUSION

In this paper, we presented S-Kendo, a practical software solution to guaranteeing deterministic execution for any program. We developed S-kendo to be seamless to use with existing programs, and we did so by overloading the existing POSIX thread library functions with our deterministic threading functions. Hence porting existing multi-threaded applications to S-kendo is straight forward, and does not require complex changes to the program. By using a static race detector with S-kendo, one can achieve deterministic execution for any program. Previous approaches to achieving deterministic execution suffered from some or all of the following problems

- Expensive hardware mechanism to track and resolve conflicts
- Expensive table lookups in software which cause huge performance degradation

- Restricted to race free programs.

In this paper, we address all the above limitations. S-Kendo does not need expensive hardware. Though it incurs a performance degradation of around 4X and 16X for the aget and lu benchmarks respectively, conservative locking does not increase this degradation. We feel that we can improve the performance of our thread library functions further. Besides, S-Kendo is not restricted to race free programs.

9. FUTURE WORK

We plan to extend our approach to improve the performance of applications that use S-kendo. Specifically, we want to reduce the waiting time of threads before they can acquire a lock. In its present form, a thread compares its logical time with the logical time of all the other thread in the system. While this is conservative, there are many instances in a program where only a small subset of threads is contending for a particular lock. In other words, locks are not always associated with all the threads of a multi-threaded program. In such cases, it is intuitive to make a thread wait only for all those threads which contend for the same lock.

We also plan to tightly integrate our S-Kendo library with the static race detector. This would enable us to directly execute the output of the static race detector to achieve deterministic execution.

REFERENCES


