Replay for Concurrent Non-Deterministic Shared Memory Applications

Mark Russinovich and Bryce Cogswell

CIS-TR-95-18
December 1995

Department of Computer and Information Science
University of Oregon

Replay For Concurrent Non-Deterministic Shared Memory Applications

Mark Russinovich and Bryce Cogswell
Department of Computer Science
University of Oregon
Eugene, OR 97403
{mer, cogswell}@cs.uoregon.edu

Replay of shared memory program execution is desirable in many domains including cyclic debugging, fault tolerance and performance monitoring. Past approaches to repeatable execution have focused on the problem of re-executing the shared memory access patterns in parallel programs. With the proliferation of operating system supported threads and shared memory for uniprocessor programs, there is a clear need for efficient replay of concurrent applications. The solutions for parallel systems can be performance prohibitive when applied to the uniprocessor case. We present an algorithm, called the repeatable scheduling algorithm, combining scheduling and instruction counts to provide an invariant for efficient, language independent replay of concurrent shared memory applications. The approach is shown to have trace overheads that are independent of the amount of sharing that takes place. An implementation for cyclic debugging on Mach 3.0 is evaluated. The algorithm implemented is compared with optimal event-based tracing and shown to do better with respect to the number of events monitored or number of events logged, in some cases by several orders of magnitude.

Keywords: Non-determinism, shared memory, repeatable execution, instruction counter

1.0 Introduction

Repeatable execution is a crucial component of cyclic debugging, performance monitoring and fault tolerance based on journaling. For example, in cyclic debugging bugs are located by repeatedly running an application and then replaying executions that exhibit erroneous behavior in order to isolate the cause of the problem. In fault tolerance, programs running on a system that has failed can be recovered by starting them from a saved earlier state and bringing them up to the state that existed at the time of the failure by recreating inputs as well as other external events.

The most difficult types of programs to replay are those that contain non-determinism. Non-determinism is present if an application given the same inputs can obtain different states and possibly produce different outputs across multiple executions. Sources of non-determinism include asynchronous interrupts, concurrent or parallel access to shared data, and dependence on external conditions such as time. In many cases increased performance can be obtained by reducing the amount of synchronization among processes sharing data, which leads to the introduction of non-deterministic access patterns. At other times a form of non-determinism, called a *race,* is inadvertently created through the incorrect implementation or omission of synchronization.

A great deal of research has focused on ways to replay program execution, with the majority focusing on replaying the non-determinism created through shared memory access in parallel programs. Recently, however, uniprocessor

operating system support for shared memory and light-weight threads has become commonplace [4][7][16][15], leading to a need for efficient mechanisms that enable the replay of concurrent shared memory programs. Existing solutions for replaying shared memory non-determinism are all based on the principle that a shared memory access is an *event*, and that an application execution can be recreated by recording and replaying the order of the events executed by the application. While this approach does enable the design of concurrent systems with repeatable execution, the overheads incurred are often prohibitive. For example, a concurrent program running on a 20 MIPS processor where 2% of instructions executed are interleaved shared memory accesses, and where each trace record is 10 bytes in size, will generate trace information at the rate of 2MB/s of execution. Tracing, and even monitoring, each shared memory access can also degrade performance significantly. Further, since the amount of tracing done depends on the application's sharing characteristics, the overheads obtained can make the approaches practical for only a restricted class of programs.

Another disadvantage of event-based approaches is that the system must be aware of every memory access that is potentially shared. In most cases, this means one of three things: that the shared accesses are well defined at the language or operating system level; that the trace process is further augmented to detect sharing dynamically; or that all memory accesses must be considered shared accesses. Each case has drawbacks that can narrow the range of applications to which the approach can be effectively applied.

In this paper we present an algorithm called *repeatable scheduling* for recording and replaying the execution of shared memory applications that has several major advantages over existing approaches. The technique is based on the fact that repeatable execution can be obtained by recreating the scheduled behavior of an application at instruction level granularity. This is accomplished through the use of an instruction counter [1][11] and a mechanism that makes the application's schedule visible to the replay system.

The first advantage of our approach over the beset event-based techniques is that trace generation rates can be reduced by several orders of magnitude. Because trace information is created only at scheduling points the amount of data generated and the frequency of such points is generally dependent only on the time-slice of the scheduler. Typical time-slices are on the order of 1-2ms, so for the 20 MIPS computer example mentioned earlier, and in general any computer, trace generation rates are fixed and on the order of 1-2KB/s. The second advantage of the approach is that it is language independent. No information about shared memory accesses or memory layout is necessary, so the program does not have to be instrumented at the access level and no run-time overhead is incurred for shared accesses. Finally, programs are instrumented with a software instruction counter, a technique that requires minimal analysis of program structure, making it simpler to implement than many existing event-based analysis techniques.

The remainder of this paper is organized as follows: In Section 2.0 we present related work in the area of repeatable execution. Section 3.0 discusses our approach, known as Repeatable Scheduling, and in Section 4.0 a replay system aimed at cyclic debugging that has been implemented using our approach is described. Section 5.0 provides a quantitative comparison of repeatable scheduling with the most efficient event-based algorithm. We summarize and conclude in Section 6.0.

2.0 Related Work

All past approaches to replaying shared memory applications have focused on recording how processes interact. The most straight-forward way of recording an execution is to save the value of every shared read operation [14]. The obvious draw-back of this approach is that if sharing is common, trace log size will rapidly explode. In order to minimize the amount of data saved, most approaches save the order of shared accesses, rather than the content.

Approaches that save order can be divided into two groups: those that assume coarse-grained sharing through synchronization or objects, and those that monitor every shared memory access. An object oriented treatment of shared memory is seen in [3], [5], [8], and [9]. While this scheme can lead to substantial reductions in trace size and overhead, it requires that the objects be implemented correctly so that they do not introduce race conditions. In some cases the object based systems can be made to treat each shared memory access individually as shared objects, but this can lead to high overheads since they are not designed for fine-grained operation.

Recording the order of data synchronization is the strategy seen in [17]. A drawback of this technique is that it is not applicable in the debugging domain since applications must be free of races for successful replay to be guaranteed.

An approach that is designed for the monitoring of every shared memory access is [12]. The method optimizes the amount of ordering information saved by dynamically performing a transitive reduction on the shared memory access patterns so that the recorded ordering is optimal. This is shown to reduce trace overheads by one to two orders of magnitude over approaches such as [8]. One potential issue is the amount of run-time space overhead incurred by the algorithm since a dependency vector must be maintained for each shared memory location. This can easily raise the memory requirements of an application by an order of magnitude if there are large amounts of shared data.

There are several shortcomings common to all of these approaches. Object based systems require support either at the language or system level. Applications written without using the designated object model cannot be re-executed under these systems. In addition, the fact that object encapsulation is assumed to be race-free can make these systems inapplicable in the debugging domain. The approaches that monitor every shared memory access require that the shared memory accesses be visible to the system and distinguishable from accesses to private data. In some systems shared data is explicitly defined through language or system constructs, but often data sharing is not obvious and can be defined at run-time. For example, many current operating systems support multithreaded execution. Applications written for this model of computation have threads of control that share their entire address space, requiring monitoring of the complete address space even though only a few locations may actually be shared. Static analysis of these applications alone is not sufficient to differentiate shared from non-shared data since pointer dereferences resolved at run-time can determine the difference. None of the existing solutions for replay can be effectively applied for practical replay of such applications.

3.0 Repeatable Scheduling

While most existing solutions to shared memory replay are designed for parallel systems, our work concentrates on uniprocessor concurrent execution. In a concurrent system running on a uniprocessor only one process or thread of control will be executing at any given point in time. Non-determinism arises in applications that share memory because the interleaving of accesses to the shared data by different processes or threads may be different across exe-

cutions of the application given identical inputs. The key to efficient repeatable execution of such applications is the realization that the interleaving is controlled by the system scheduler. To precisely repeat the execution of a non-deterministic shared-memory application the coarse-grained interleaving characteristics can be recreated by repeating the same schedule, ensuring that context switches occur at exactly the same instructions during replay as during the original execution.

This section first presents the assumptions made throughout this work, and then describes the basic approach to repeatable scheduling. Throughout this section and the rest of the paper we refer to executions of an application that are meant to exactly repeat some other execution as *repeat executions* or *replay executions*. The execution that is being used as the basis for the repetition is known as the *original execution*.

3.1 System Model

This work assumes that applications consist of one or more processes or threads. There are no restrictions on the amount of address space shared among threads or processes of the application and this can vary between specific threads or processes of the application. Further, memory sharing can be local between two or more processes or threads in the application, or global across all the processes and threads in the applications. There are no assumptions regarding the visibility or ordering of shared memory accesses. Since synchronization of shared memory accesses is not assumed, non-determinism based on shared memory access ordering or the arrival of asynchronous events can be present in the application.

Applications run under the control of a *replay system*, which is in charge of both recording and playing back an application's behavior. It is assumed that the replay system can be notified of context-switches within the application before a newly scheduled process or thread is allowed to execute. It is also assumed that there is a way for the replay system to control the scheduling of processes and threads. Basic support for this assumption is provided on most systems with sleep and wakeup system calls.

Finally, we address issues involving replaying shared memory access only. Issues regarding the recreation of an external environment required for replay, such as input, the state of other applications or hardware for example, are outside the scope of this work.

3.2 How the Repeatable Scheduling Algorithm Works

The idea behind the repeatable scheduling algorithm is partly based on past work in repeatable execution that focuses on replaying asynchronous events [5][11]. In the previous work, the model is that of a single process where non-determinism is present in the form of asynchronous interrupts. Such interrupts can be delivered to a process at any instruction, and correct replay of a process that has received asynchronous events requires that such events be replayed at the same instructions in each repeated execution. This can be accomplished by using an instruction counter, as in [11], to record how many instructions have executed between events and then controlling how many instructions are executed in a replay before an event must be recreated.

An instruction counter implemented in software is a memory location or dedicated register that is initialized to its maximum value at the start of original execution. The counter is incremented each time the application makes a backward control transfer. Thus, for any instruction in the execution sequence there is a unique counter value and instruc-

tion pointer that can be used to identify the precise location in the sequence of asynchronous events. When an asynchronous event occurs the replay system saves the instruction count and the instruction pointer of the thread or process being interrupted. During replay execution, the counter is initialized to the negative of the value stored in the log, and when the counter increments to zero a routine is called which places a breakpoint on the instruction associated with that count. When the process or thread hits the breakpoint the asynchronous event is replayed. The next pair of counter and pointer is then read to determine when to replay the next event.

In our algorithm we allow for non-determinism to be introduced through shared memory accesses. On a uniprocessor the ordering of shared memory accesses is ultimately determined by how the processes or threads of an application are scheduled. We therefore have extended the notion of an asynchronous event in [11] to include process or thread preemptions performed by the scheduler. In the original execution an instruction count is saved at each preemption point in the application, recording the exact place in the execution where the pre-emption occurs, and the identifier of the process or thread that it preempted. In a replay execution, the saved preemption information is read and used to force preemptions at the same places in the execution.

Consider the example shown in Figure 1(A). An application that is to be replayed consists of three threads which are scheduled as shown. At each of the pictured preemption points the instruction count, instruction pointer and the identifier of the thread being preempted are saved to a log. In the replay, shown in Figure 1(B), the replay system starts the re-execution by putting threads 2 and 3 to sleep, but letting thread 1 run until it has executed the same number of instructions as it did in the original execution before the first preemption point. When that preemption point is reached, the replay system is notified and puts thread 1 to sleep. The replay scheduler then makes the same scheduling decision as in the original run so it again wakes up thread 2. A similar procedure is followed at the second preemption point.

The two requirements that make a replay system possible are that the application's schedule is visible to the replay system and that an instruction counter is provided to measure the progress of the application.

4.0 Application to Debugging

The debugging of concurrent and parallel programs can be especially difficult because of non-determinism in shared memory access, both intentional and inadvertent. We have implemented a replay system for concurrent Mach 3.0 [16] applications that require repeatable execution for debugging purposes. Mach provides both explicit shared memory with virtual memory control primitives and implicit shared memory through the use of multithreading, making existing event-based replay techniques unsuitable.

4.1 Mach 3.0 Replay System Design

To make our implementation as general and portable as possible we have made it part of a new "debug" version of the Mach system library. The steps necessary to make an application ready for repeatable execution are shown in Figure 2. The application is first passed through a preprocessor that instruments the assembly language output of the compiler with a software instruction counter. The application is then linked with the debug library (which is also instrumented) in lieu of the standard library. After the program is run, it can then be forced to repeat non-deterministic

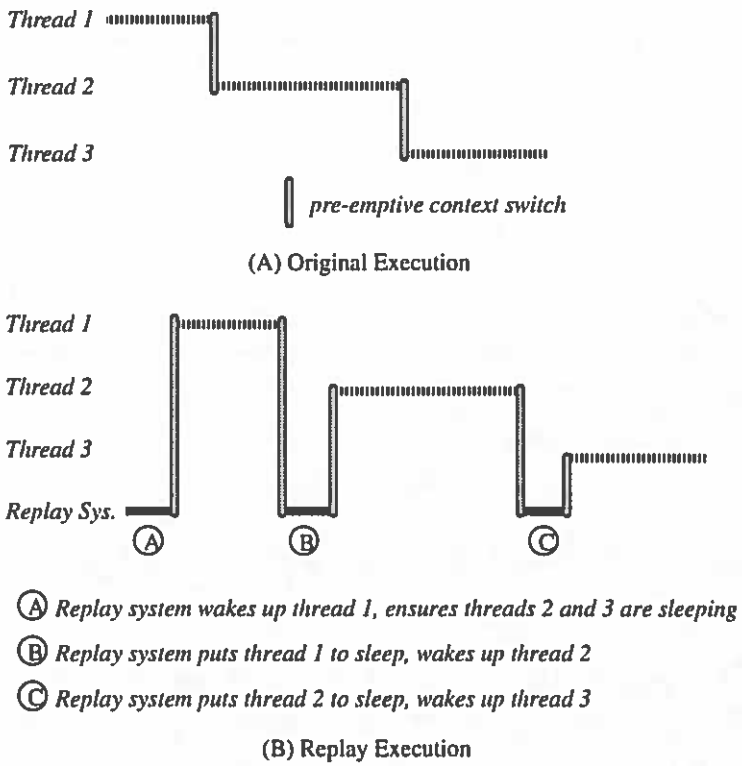

Figure 1. Example of repeatable scheduling

behavior by specifying a command line flag with the log file of the original run. By running the application under an existing debugger, such as GNU's *gdb* for Mach 3.0, bugs can be isolated and identified.

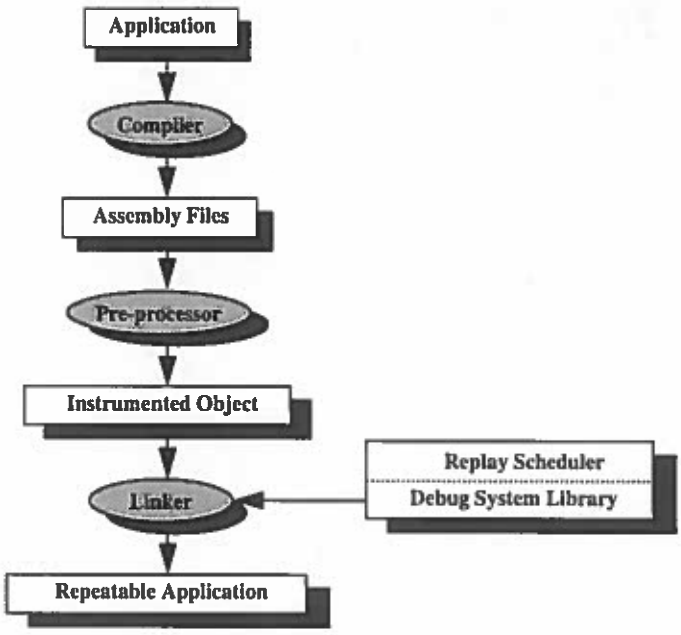

Figure 2. Steps in preparing an application for repeatable execution

4.2 The Preprocessing Stage: Implementing The Instruction Counter

Applications being readied for repeatable execution are first compiled with the GNU *gcc* compiler, which allows one to specify that a particular register be left unused in the assembly output. The assembly language output of such a compilation is fed to the replay preprocessor that performs the software instruction counter instrumentation, using the dedicated register as the instruction count. Backward branches and calls are located, with code such as the following pseudocode fragment added at each occurrence:

```
increment count register;
if(count register == 0) call overflow();
```

The preprocessor also replaces the application's `main()` function with a wrapper in the debug library that is used to initialize the replay system. This function also allows for command line flags to be passed to the replay system and is further described below. The preprocessor implemented as an *Awk* script [1] is about 100 lines long.

4.3 The Debug Library: Scheduling the Application

Once the application has been instrumented, it must be linked with instrumented versions of the system libraries such as `libc.a`. Normally, Mach programs are linked with the `libmach.a` library, but when repeatable execution is desired the application is instead linked with the replay version of the Mach library called `libreplay.a`. This version of the Mach library is instrumented and monitors or recreates the scheduling of applications. The decision to implement the scheduler as a library limits the replay to single applications, but is the most portable approach to making an application's schedule visible to the replay system.

When an application begins executing it consists of one Mach thread running in one task (a task is essentially an address space - one thread running in a task is equivalent to a Unix process). If the command line flags indicate that the run is an original execution the replay system prepares itself for logging, but remains dormant until the application becomes multi-threaded through the creation of another thread. At that time the replay system spawns a scheduling thread that runs at a higher priority than the application threads and begins controlling the scheduling of the application.

The replay system scheduler keeps track of the threads that exist throughout the application's execution. At any point in time the scheduler allows only one thread to be active and therefore runnable by the Mach scheduler. When threads are preempted, the scheduler records the thread's identifier, instruction count value and instruction pointer before waking up the next thread.

To replay an execution the user must specify a replay log on the application's command line. The replay scheduler reads pre-emption records from the log and instead of preempting based on time-slices, it uses the instruction counter to determine when to force preemptions. Because the scheduler is deterministic, it performs the same scheduling decisions as it did in the original run of the application, meaning that no information need be logged during context-switches that occur due to blocking (since the succeeding thread will be the same during both the original run and replay). This ensures that the only scheduling decisions that must be logged are those due to time-slice preemptions.

4.4 Mach 3.0 Replay System Performance

The replay system is implemented on Mach 3.0/UX (UNIX 4.3 BSD) running on a 90MHz Pentium processor with 16MB of memory. Three synthetic workloads are used to demonstrate the best and worst-case performance of the replay system. All three are multithreaded single task applications written using the Mach C-Threads library.

The first application is a simple counting program where multiple threads increment a shared counter until it reaches a specified value. There is no synchronization in the increment loop so the order the threads perform the increment is non-deterministic. Therefore the scheduler's time-slicing policy will determine the order and frequency threads take turns at incrementing the counter. This application demonstrates the overheads incurred by the instruction counter in small loops and the overhead of monitoring and logging pre-emptions.

The second application is a variant of the first with added synchronization. The shared increment operation takes place in rounds, where each thread takes a turn at incrementing the counter and then waits until all the other threads have incremented the counter once before incrementing it again. Increments within each round occur in a non-deterministic order, and because blocking context-switches will occur on virtually every increment of the counter, this demonstrates the replay system's overhead for performing scheduling decisions at non-pre-emptive context-switches.

The final application is a standard multi-threaded matrix multiply algorithm. A specified number of threads are created and assigned blocks of the matrices to multiply independently. This application contains no synchronization and will exhibit the same context-switching behavior as the first application. However, the number of shared variables accessed depends on the sizes of the matrices being multiplied. This application demonstrates that for repeatable scheduling, overheads are independent of the amount of data sharing performed by an application.

The SPLASH-2 (Stanford Parallel Applications for Shared-Memory) benchmark suite is used to quantify the expected behavior in real applications [18]. All SPLASH executions use 8 threads. Results for both the synthetic workloads and the SPLASH benchmarks are given in Table 1. The Native column contains execution time using the standard Mach and C-Threads libraries under the control of the Mach system scheduler. The Instrumented column shows execution time when applications are instrumented and run using the repeatable scheduling scheduler, and the Replay column gives time for subsequent replayed executions. The final column is the size of the trace logs generated for the instrumented executions.

The table demonstrates that performance degradation due to instrumentation and scheduling is around 10-15%. In two instances, *synchronous count* and *Cholesky*, the performance with instrumentation is superior to the native system because the imposed scheduler uses a different, occasionally superior, scheduling algorithm than the native scheduler. In those cases two measurements are shown for native execution: time for a run with a single thread and for a run with the same number of threads as was used for the instrumented execution. Trace size generation rates are fairly constant at about 1KB per second across all the applications measured. This result is significant in light of the fact that it is independent of the amount of shared accesses in the application. For example, *LU-contiguous* executes 93 million shared reads and 44 million shared write, all of which require atomic monitoring by event-based replay algorithms. Under repeatable scheduling only 14KB of trace is generated with an execution overhead of about 15 percent.

Application		Native Execution (s)	Instrumented Execution		Replay Execution		Trace Size
			Time (s)	Ovrhd	Time (s)	Ovrhd	
Synthetic Workloads	count	8.47	9.16	8.1%	9.44	11.5%	11KB
	sync. count	0.24/6.42	3.32	na	3.58	na	3KB
	matrix multiply	27.59	30.05	8.9%	31.35	13.6%	26KB
Splash Kernels	Radix	4.94	5.22	5.7%	5.37	8.7%	6KB
	LU - contiguous	10.58	12.33	16.5%	12.20	15.3%	14KB
	LU - non-contiguous	16.08	17.52	9.0%	17.79	10.6%	20KB
	FFT	1.65	1.82	10.3%	1.85	12.1%	1KB
	Cholesky	1.66/16.48	2.63	na	2.66	na	2KB

TABLE 1. Performance of Mach 3.0 Replay System

5.0 Comparison with Optimal Event-Based Replay

We present in this section a comparison of the repeatable scheduling algorithm to the optimal parallel trace algorithm of [12]. Though the optimal trace algorithm supports replay on true parallel machines as well as sequential, to this date it is also the optimal approach for replay of concurrent systems. The optimal trace algorithm, while requiring more complex run-time monitoring of shared accesses than other event-based replay methods, dynamically detects race conditions in order to generate the minimal amount of trace information. Race-condition detection is performed by maintaining dependency vectors for each shared memory location that indicate the order of thread read and write accesses to the location. The comparison is made not on raw performance, but rather on the criteria of number of trace records generated, and the number of events monitored during execution.

Both algorithms require approximately the same amount of information in a trace record so comparing the number of records will indicate relative trace log size. The repeatable scheduling algorithm requires that scheduling take place at system calls that can block, so we treat such calls as monitored events. In the optimal trace algorithm, all shared memory accesses are monitored to determine minimal sequencing information. To make the comparison as favorable as possible for the optimal trace algorithm, we assume that monitoring is performed only on the actual data being shared by the algorithm implemented in each test program rather than the entire address space of the threads as is actually the case. Further, we do not model the optimal trace algorithm's run-time space overhead that results from the fact that dependency vectors must be maintained for each shared memory location. The dependency vector overhead can raise the memory requirements of an application like matrix multiply by an order of magnitude. The same test programs presented in the previous section are again used here. The results of the comparison are shown in Table 2.

For the count programs the number of threads is set at 10 and the iterations set at one million. Execution requires 1 second on a system with 100ms time-slices, so there are 10 preemptive context switches during the run. Since there is no synchronization in this version of the count program both algorithms will generate trace information only at preemption points, at which time the thread accessing the counter changes. However, each access to the counter must be monitored by the optimal trace algorithm, which in practice will lead to substantial performance degradation.

Application	Optimal Trace		Repeatable Scheduling	
	Events Monitored	Events Logged	Events Monitored	Events Logged
count	10^6	10	10	10
sync. count	10^6	10^6	10^6	10
matrix multiply	$12.8x10^6$	$6x10^3$	$6x10^3$	$6x10^3$

TABLE 2. Comparison of Optimal Trace and Repeatable Scheduling

The comparison for the synchronized count program is shown next. Because non-preemptive context switches occur after every counter increment, the repeatable scheduling algorithm will monitor the same number of events as the optimal trace algorithm (note that this does not include the lock, which is another shared variable that must also be monitored by the optimal trace algorithm); however, the repeatable scheduling algorithm will only generate trace information at pre-emptive context-switches, while the optimal trace algorithm must save a record after every increment.

Finally, a comparison of the matrix multiple program is shown at the bottom of the table. In the execution modelled, two 400 by 400 arrays are multiplied together. This takes about a minute on a 90MHz Pentium, resulting in about 6000 pre-emptive context switches. Since there is no synchronization, the repeatable scheduling algorithm does not monitor any events, but the optimal trace algorithm must monitor every shared variable access, resulting in a high number of monitored events.

6.0 Summary

We have presented an algorithm called *repeatable scheduling* for repeatable execution of concurrent non-deterministic shared-memory applications. The algorithm has several major advantages over existing replay algorithms. First, unlike existing event-based techniques, there is no need to differentiate shared memory accesses from non-shared accesses. The trace log generation rate is fixed, small, and is independent of the amount of sharing or synchronization present in the application being replayed. The number of monitored events is also independent of the number of shared memory accesses. Finally, the algorithm is simple to implement and has minimal impact on application memory requirements.

We have implemented the algorithm in the context of a debugging replay system for multi-threaded Mach 3.0 applications and shown that the repeatable scheduling technique is an efficient way to debug non-deterministic concurrent programs. In addition, comparisons of repeatable scheduling with the optimal trace event-based replay algorithm indicate that repeatable scheduling does significantly better in either number of events monitored or the number of events logged for asynchronous as well as synchronous applications.

Future work includes performance enhancements to the repeatable scheduling implementation as well as its application in the fault-tolerance domain. Exploration into the use of repeatable scheduling for debugging parallel applications on a uniprocessor is also being considered. Because varying the time-slices in the scheduling algorithm provides the ability to control the amount of non-determinism present in an execution, repeatable scheduling may provide an efficient way to debug parallel programs with control over the non-determinism versus performance and trace size trade-off.

7.0 References

[1] A. Aho, B. Kernighan and P. Weinberger, "The AWK Programming Language," Addison-Wesley, Reading, MA, 1988.

[2] T. A. Cargill and B. N. Locanthi, "Cheap Hardware Support for Software Debugging and Profiling," in *Proc. Symp. on Architectural Support for Prog. Lang. and Operating Syst.*, Palo Alto, CA, Oct. 1987, pp. 82-83.

[3] R. H. Carver and K. C. Tai, "Reproducible Testing of Concurrent Programs Based on Shared Variables," in *Proc. 6th Int. Conf. on Distributed Computing Systems*, Boston, MA., May 1986, pp. 428-432.

[4] H. Custer, "Inside Windows NT," Microsoft Press, Redmond, WA, 1993.

[5] P. Dodd and C. Ravishankar, "Monitoring and Debugging Distributed Real-Time Programs," *Software Practice and Experience*, Vol. 22(10), Oct. 1992, pp. 863-877.

[6] M. Johnson, "Some Requirements for Architectural Support of Software Debugging," in *Proc. of the Symp. on Architectural Support for Prog. Lang. and Operating Syst.*, Palo Alto, CA, Mar. 1982, pp. 140-148.

[7] A. King, "Inside Windows 95," Microsoft Press, Redmond, WA, 1994.

[8] T. J. LeBlanc and J. M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay." *IEEE Trans. on Computers*, Apr. 1987, pp. 471-482.

[9] C. Lin and R. LeBlanc, "Event-Based Debugging of Object/Action Programs," in *Proc. of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, 1988, pp. 23-34.

[10] C. E. McDowell and D. P. Helbold, "Debugging Concurrent Programs," *ACM Computing Surveys*, Dec. 1989, pp. 593-622.

[11] J. M. Mellor-Crummey and T. J. LeBlanc, "A Software Instruction Counter," in *Proc. Symp. on Architectural Support for Prog. Lang. and Operating Syst.*, Palo Alto, CA, Apr. 1989, pp. 78-86.

[12] R. Netzer, "Optimal Tracing and Replay for Debugging Shared-Memory Parallel Programs," in *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, May 1993, pp. 1-11.

[13] R. Netzer and B. Miller, "On the Complexity of Event Ordering for Shared-Memory Parallel Program Executions," in *Proc. Int. Conf. on Parallel Processing*, 1990, pp. 93-97.

[14] D. Pan and M. Linton, "Supporting Reverse Execution of Parallel Programs," in *Proc. SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, May 1988, pp. 124-129.

[15] M. L. Powell, *et. al.*, "SunOS Multithreaded Architecture," Sun Microsystems White Paper, Sun Microsystems, Cupertino, CA, June 1995.

[16] R. Rashid, et. al., "Mach: A Foundation for Open Systems," in *Proc. 2nd Workshop on Workstations and Operating Syst.*, Sept. 1989, pp. 27-29.

[17] K. C. Tai, R. H. Carver, and E. E. Obaid, "Debugging Concurrent Ada Programs by Deterministic Execution," *IEEE Trans. on Software Engineering*, Jan. 1991, pp. 45-63.

[18] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Proc. of the 22nd International Symposium on Computer Architecture*, June 1995, pp. 24-36.