Operating System Support for Replay of 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 briefly 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. We focus on the design of a general operating system exported framework that makes possible the use of hardware instruction counters and scheduling notifications for efficient implementation of repeatable scheduling.

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 and 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.

In this paper we present an algorithm called *repeatable scheduling* [8] 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][4][7] and a mechanism that makes the application's schedule visible to the replay system.

The remainder of this paper is organized as follows: Section 2.0 discusses our approach, known as Repeatable Scheduling and Section 3.0 describes the general operating system support we are adding for efficient application replay. Finally, Section 4.0 summarizes with some conclusions.

2.0 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 executions 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.

2.1 System Model

The idea behind the repeatable scheduling algorithm is partly based on past work in repeatable execution that focuses on replaying asynchronous events [3][7]. 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 [7], to record the number of instructions that have executed between events and then controlling how many instructions are executed in a replay before an event must be recreated.

.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 [7] to include 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.

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.

2.2 Scheduling

The replay system keeps track of the threads that exist throughout an application's execution. When threads are preempted, the replay system records the thread's identifier and instruction count value before waking up the next thread.

To replay an execution the replay system reads preemption records from a log and instead of preempting based on time-slices, use the instruction counter to determine when to force preemptions. Because the scheduler is deterministic (an assumption discussed later), 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 synchronization (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 and blocking on I/O (the timing of which can vary from run to run).

3.0 Operating System Support

A repeatable scheduling system can be implemented using either a hardware [5][4] or software [7] based instruction counter, and the necessary scheduling support can reside in either user space or within the operating system. We have implemented a software instruction counter based approach in user space under the Mach 3.0 operating system is described in [8], and the performance is shown to compare well to native execution time (overhead of 10-15%). However, a number of improvements to this system are made available by making use of a hardware instruction counter supported by operating system extensions.

Hardware instruction counters have been in general use since about 1987 when HP introduced its HP-Precision architecture, and they have recently been implemented in the Pentium and Pentium Pro processors. This trend indicates wide-spread acknowledgment of the potential uses of hardware instruction counters.

The advantage of using a hardware instruction counter is improved performance and, more importantly, application transparency. In a software-based approach the code must be built into applications at compile-time. Moving to a hardware-based counter allows replay to be accomplished without recompiling.

Providing operating system hooks supporting such a hardware instruction counter vastly simplifies the replay system design, and lends greater flexibility to the scheduling policies that are possible. This is due to the fact that the OS is able to save and restore the instruction counter values at context-switches and it is directly involved in performing preemptions and scheduling decisions.

The remainder of this section will describe the general requirements for a hardware based operating system supported replay system, and then describe an implementation in progress using the Mach operating system running on the Pentium processor.

3.1 Requirements

The requirements for implementing an OS supported hardware based replay system are minimal, and can be divided into two categories: support for reading and writing the instruction counter and scheduling support. These two areas will be discussed in turn.

The purpose of the instruction counter is to measure the number of instructions that are executed by a thread during each of its timeslices, which allows an identical execution profile to be recreated during replay. In support of this, the OS must associate an instruction counter value with each thread and provide read/write access to it during context switches among threads that are to be replayed. This is easily accomplished by having the OS save and restore the instruction counter value along with the remainder of the register set associated with each thread, and then send a signal or message to the replay system with access rights to the thread's context during context switches.

OS scheduling support is more complicated, because one of the predicates of the replay system is that scheduling is deterministic. Our definition of deterministic scheduling is that the thread selected to run during each preemption or blocking operation depends only on the sequence of previously scheduled threads and their associated instruction counts. Beyond this requirement it is necessary to allow the replay system to record the scheduling decisions made during the initial run, and then enforce that scheduling behavior during replay. OS support can be provided by sending a message to the replay system during context switches indicating the next thread to be scheduled, and allowing the replay system to modify the decision if desired.

3.2 Mach 3.0 Implementation

To support efficient repeatable scheduling, as well as other types of replay based on hardware instruction counters, we have designed a few simple, yet powerful system calls that give applications the necessary power for replay with minimal intrusion and overhead. The system calls are divided into two general and distinct categories: those dealing with hardware instruction counters, and those dealing with application thread scheduling.

The hardware instruction counter routines essentially export hardware instruction counter control and visibility to user space, allowing a threads to control and monitor the progress of other threads. Each thread has its own unique instruction counter value that measures the progress of the thread. Uniqueness and repeatability are achieved by having the operating system record counts for only instructions executed by the threads in user-space (i.e. instruction counting for a particular thread would stop on any transition of control from the thread to the operating system and start on any transition in the reverse direction).

Listed below are the user-level call prototypes and descriptions for the replay functions, which are actually library routines that sit on a message-based system call interface, that are added to provide an application-level abstraction to a hardware instruction counter.

```
void mach_thread_register_counter( thread,
                                   rollover_callback )
thread_t   thread;
thread_t   (*rollover_callback)();
/* callback definition */
void       rollover_callback( thread_t thread );
```

This call registers a thread with the operating system for instruction counting. It can only be made on threads currently in a suspended state and typically is used just after a thread is created, but before it is allowed to execute. The callback function is a routine that is called in the registering thread's context whenever the instruction counter of the target thread rolls over. The callback procedure can be used to monitor or control the execution of the application.

```
void mach_thread_set_instruction_counter( thread,
                                          countval)
thread_t   thread;
int        countval;
```

This call simply sets the specified thread's instruction count to the indicated value. The counter's value determines the number of instructions executed before a thread that has registered a rollover callback will be notified.

```
int mach_thread_get_instruction_counter( thread )
thread_t   thread;
```

Calls to this function return the current value of a thread's instruction counter.

```
void mach_thread_register_scheduling( thread,
                                      sched_callback )
thread_t   thread;
thread_t   sched_callback;
/* callback definition */
thread_t   sched_callback( thread_t prev,
                           thread_t next );
```

In order to support repeatable scheduling, the replay manager must be aware of, and have control over, the context switches of threads it is running. The register scheduling function informs the scheduler that the calling thread wishes to receive notifications whenever the thread specified as an argument is pre-empted or blocks due to external conditions such as hardware I/O or message queues that are registered. Further, all threads in the system that are registered with this system call are, among themselves, scheduled deterministically at all non-I/O non-preemptive context switches (i.e. blocking). Any threads that are spawned by threads that have been registered inherit the registered characteristics of their parent. In other words, pre-emptions of those threads, as well as the first scheduling of those threads, also generate notifications to the same callback routine and they are added to the group of deterministically scheduled threads.

The callback routine is passed identifiers of both the thread that is being pre-empted and the thread that the scheduler would schedule next under its deterministic policy. The callback routine can change the selection of the next thread by returning as the result an alternate thread (from the group of registered threads) to be scheduled.

3.3 Using the Interface for Efficient Repeatable Scheduling

Use of the calls to implement repeatable scheduling based on a hardware instruction counter is relatively straight forward. A single thread is used to implement the replay manager. It launches the first thread by creating it, registering it with the scheduler and setting its instruction counter to the maximum value. After the thread begins execution on the replay scheduler is notified of any scheduling pre-emptions that occur in it or any of its descendant threads.

The pre-emption notification callback of the replay manager records the instruction counter of the pre-empted thread, obtained by reading its value, and the thread that the operating system has indicated will be run next.

In replay executions, the initial thread is started by setting its instruction counter to the value it had at its first pre-emption. Any child threads are also set in the same way. When a currently executing thread's counter rolls over, the replay manager wakes up, reads a record from the log it recorded during the original execution, and determines which thread is supposed to be scheduled next. At the same time, it updates any counter values necessary to ensure preemptions at appropriate counts.

4.0 Summary

Building upon our experiences with software-based instruction counting for repeatable scheduling, we have described an improved implementation in which the infrastructure of the system is built into the operating system, and takes advantage of hardware based instruction counting. This new approach requires relatively few new OS services, and yields a simple, high performance, application transparent system for repeatable scheduling.

5.0 Acknowledgments

We would like that thank Mootaz Elnozahy and Zary Segall for their helpful discussions and insights regarding applications for hardware instruction counters.

6.0 References

[1] 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.

[2] 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.

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

[4] E. Elnozahy, "An Efficient Technique for Tracking Nondeterministic Execution and its Applications," Carnegie Mellon University technical report CMU-CS-95-157

[5] 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.

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

[7] 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.

[8] M. Russinovich and B. Cogswell, "Replay For Concurrent Non-Deterministic Shared Memory Applications", U. of Oregon technical report CIS-TR-95-18.

[9] 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.