Using Pin as a Memory Reference Generator for Multiprocessor Simulation *

Collin McCurdy[1,2] and Charles Fischer[2]

[1]Oak Ridge National Laboratory
[2]Department of Computer Sciences, University of Wisconsin–Madison

Abstract

In this paper we describe how we have used Pin to generate a multithreaded reference stream for simulation of a multiprocessor on a uniprocessor. We have taken special care to model as accurately as possible the effects of cache coherence protocol state, and lock and barrier synchronization on the performance of multithreaded applications running on multiprocessor hardware.

We first describe a simplified version of the algorithm, which uses semaphores to synchronize instrumented application threads and the simulator. We then describe modifications to that algorithm to model the microarchitectural features of the Itanium2 that affect the timing of memory reference issue. An experimental evaluation determines that, while our methods enable accurate simulation, the use of semaphores has negative impact on the performance of the simulator.

1. Introduction

Dynamic instrumentation tools, such as Intel's Pin [3], enable users to dynamically modify binary applications on the fly through the static definition of procedure calls that are inserted dynamically into the instruction stream based on instruction type.

The idea of using a tool like Pin for architectural simulation is enticing for two reasons:

1. Ease of implementation – since the tool provides access to every instruction in the instruction stream, and methods for distinguishing their important features, much of the burden of decoding and interpreting instructions is removed from the simulator writer.

2. Performance – since instructions, after an initial instrumentation by Pin, run on the *hardware* they were meant to run on, rather than through an interpreter, there is the potential for increased simulator performance versus a more standard implementation which decodes every dynamic instruction.

These were among the considerations that led us to Pin in our search for a mechanism to generate memory references for our existing back-end memory system simulator, a modified version of the Slicc/Ruby simulator created by the Multifacet group at the University of Wisconsin [5].

In this paper we describe how we have used Pin to generate a multithreaded reference stream for simulation of a multiprocessor on a uniprocessor. Since our goal was to use the simulator to evaluate the performance of cache-coherence protocols on existing multiprocessor hardware (the HP Superdome), a primary concern was to model as accurately as possible the effects of coherence protocol state, and lock/barrier synchronization on the performance of multithreaded applications running on multiprocessor hardware.

The algorithm we describe ensures that references are simulated in the same global order that they would have occurred in a multiprocessor by halting a thread when it blocks on a memory reference and only releasing it after time has been simulated up to and including the time at which the reference completes. In this way we ensure that the simulator never simulates a reference by one thread to an address that another thread should have referenced first.

The rest of the paper proceeds as follows: we begin in Section 2 with a brief discussion of Pin and how it works. Then in Section 3 we introduce our algorithm by describing the implementation of a prototype version of our full-scale simulator which blocks on every reference. Since the Itanium2 [4] processor on which the Superdome architecture is based does not block on most memory references, the actual algorithm used by our simulator is more complicated. Section 4 describes in detail how we model those elements of the Itanium2 microarchitecture essential for the accurate simulation of memory references. Finally, in Section 5 we present an evaluation of both the accuracy of the simulator and its performance.

*Prepared by OAK RIDGE NATIONAL LABORATORY P.O. Box 2008 Oak Ridge, Tennessee 37831-6285 managed by UT-Battelle, LLC for the U.S. DEPARTMENT OF ENERGY under contract DE-AC05-00OR22725

2. Pin

In this section we give a brief overview of Pin and the application interface it presents to users.

A user creates a Pin tool – by writing C or C++ code with calls to the Pin library – compiles and links it with the Pin library, and then runs an application binary through the Pin tool by passing it (and it's arguments) on the command line.

The primary interface to the Pin library is the *PIN_AddInstrumentInstructionFunction()* which takes as an argument a user-defined function which will be called every time the Pin tool instruments an instruction from the target application. (Another library function *PIN_AddInstrumentInstructionSequence()* may be used instead to instrument at the granularity of a basic block.)

Within that user-defined function, known in Pin parlance as an "instrumentation" function, use can be made of inspection routines to determine the instruction type, and then based on type, insert calls to "analysis" routines. Analysis routines can take as parameters runtime values such as register contents.

The idea is that instrumentation routines are called only a few times since they are associated with the static executable, and thus can afford to be broadly targeted (i.e., to every instruction), while analysis routines are associated with the instructions of interest, and are only called when those instructions are executed.

Essentially a "just-in-time" compiler, the Pin implementation works as follows:

1. Pin intercepts the execution of the first instruction of the application.

2. Pin then generates code for the straight-line code sequence starting at this instruction, inserting any required calls to analysis routines and ensuring that it *regains* control when a branch exits the sequence.

3. Pin then transfers control to the generated sequence.

4. After regaining control, Pin generates more code for the branch target and continues execution.

5. Translated and instrumented code is saved in a code cache for future execution of the same sequence of instructions to improve performance.

3. Simulator Prototype

As discussed in the introduction, ensuring accurate modeling of synchronization and cache coherency protocol state requires that special care be taken to ensure that threads execute instructions in an order that is consistent with a multiprocessor execution. This requirement imposes a need for synchronization between the threads and the simulator. We use semaphores to implement that synchronization.

3.1. Semaphores

Introduced by Dijkstra [1] as a mechanism to implement critical sections, semaphores are essentially counters that control access to resources. The interface to semaphores consists of two components: *wait* and *signal*, also known as *down* and *up* respectively. A typical implementation of *wait* causes the initiating thread to sleep until it is *signal*'d by another thread.

Semaphores provide a simple mechanism to synchronize the producer/consumer relationship between the application threads and the simulator:

1. Consumer *wait*s for producer

2. Producer produces, then *signal*s the consumer and *wait*s for consumer.

Unfortunately, in the absence of a lightweight thread library, the efficient implementation of semaphores is still something of an open problem: because *wait* and *signal* require atomicity and an efficient sleep mechanism, they are typically handled in the operating system kernel, implying a costly context switch on every call.

3.2. Prototype Algorithm

Figure 1 demonstrates our algorithm stripped down to its bare essentials.

Application threads run through the Pin tool created from the code on the left-hand side of Figure 1: the instrumentation routine *Instruction()* associates the *reference()* analysis routine with every memory reference: i.e., all *ld* and *st* instructions will call *reference()* before they execute. Meanwhile, all other instructions are instrumented to call the *incr_cycle()* analysis routine.

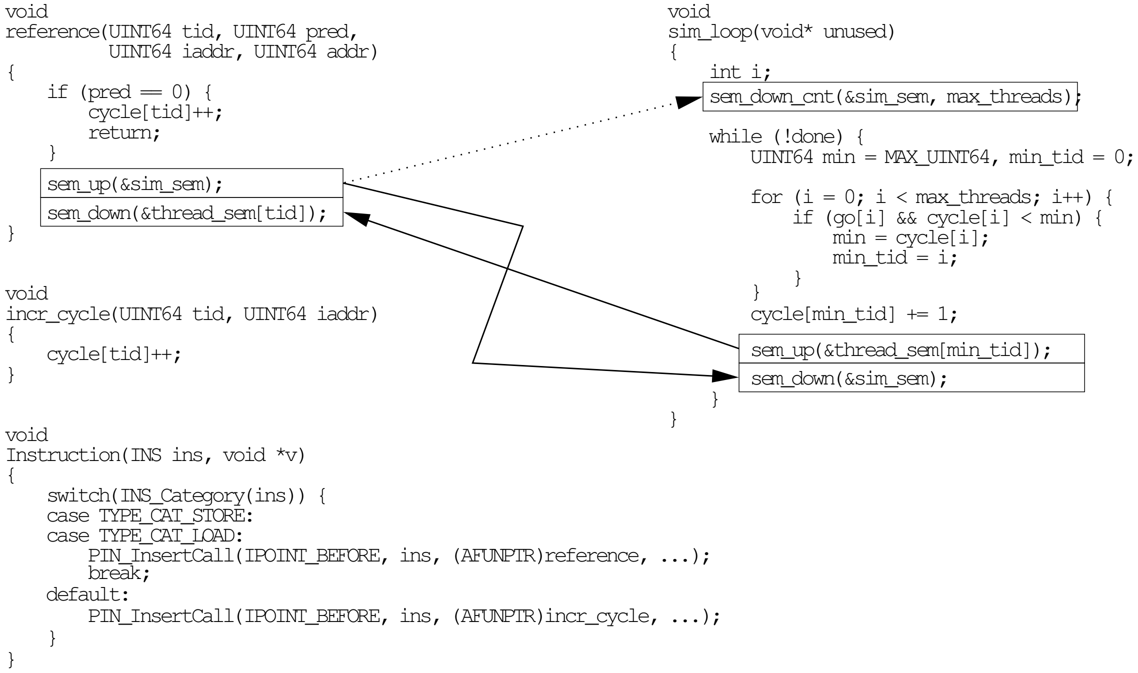
Not shown in the figure, the Pin tool makes use of another Pin library routine to start a simulator thread before starting the application threads. The prototype "simulator" simply executes the code on the right-hand side of Figure 1.

The algorithm then proceeds as follows:

- Immediately after starting, the simulator *wait*s on a "multiple count" semaphore: the semaphore must be *signal*'d multiple times – in this case once by each thread – before the simulator can continue.

- All application threads are then started as normal by Pin. When an application thread hits its first memory instruction it *signal*s the semaphore that the the simulator is waiting on (modeled by the dotted arrow in the figure) and then *wait*s on its own semaphore.

- After all the threads have *signal*'d and *wait*'d, the simulator thread wakes up and "simulates". The prototype simulator does nothing, but the real simulator can now safely simulate time up till the next completion of a reference.



```
void
reference(UINT64 tid, UINT64 pred,
          UINT64 iaddr, UINT64 addr)
{
    if (pred == 0) {
        cycle[tid]++;
        return;
    }
    sem_up(&sim_sem);
    sem_down(&thread_sem[tid]);
}

void
incr_cycle(UINT64 tid, UINT64 iaddr)
{
    cycle[tid]++;
}

void
Instruction(INS ins, void *v)
{
    switch(INS_Category(ins)) {
    case TYPE_CAT_STORE:
    case TYPE_CAT_LOAD:
        PIN_InsertCall(IPOINT_BEFORE, ins, (AFUNPTR)reference, ...);
        break;
    default:
        PIN_InsertCall(IPOINT_BEFORE, ins, (AFUNPTR)incr_cycle, ...);
    }
}
```

```
void
sim_loop(void* unused)
{
    int i;
    sem_down_cnt(&sim_sem, max_threads);

    while (!done) {
        UINT64 min = MAX_UINT64, min_tid = 0;

        for (i = 0; i < max_threads; i++) {
            if (go[i] && cycle[i] < min) {
                min = cycle[i];
                min_tid = i;
            }
        }
        cycle[min_tid] += 1;
        sem_up(&thread_sem[min_tid]);
        sem_down(&sim_sem);
    }
}
```

Figure 1. Prototype code.

- The prototype simulator then releases the thread with the oldest reference which executes until it reaches a memory instruction, wakes up the simulator, and so on.

4. Full-Scale Simulator

As noted in the Introduction, the Itanium2 does not block on many memory operations, but rather on instructions that depend on the results of those operations. In this section we describe how we modified the prototype to more accurately model the Itanium2 processor. Though our primary interest is not accurate microarchitectural simulation, certain microarchitectural features must be taken into account to ensure the accurate timing of reference generation.

4.1. Handling Multiple Outstanding Loads

Since reads do not necessarily block, the placement of semaphores to synchronize with the simulator is no longer obvious: blocking is not associated with the *ld* instruction that initiated the request, but rather with another, arbitrary, instruction that uses the register into which the value is read from memory.

Our approach to this problem is to use indirection: semaphores are associated with a table of outstanding requests, and each register use is preceded by a lookup into the table to determine whether the register is associated with an entry in the table; if it is then the thread blocks on the associated semaphore. Our Pin tool's analysis routines maintain two structures to help implement the task:

1. MemQueue, with an entry for each outstanding request, keeps track of outstanding requests: each request is given a slot in the MemQueue and each slot in the MemQueue has an associated semaphore.
2. RegReady, with an entry for each CPU register, keeps track of when that register is ready to be read. Instructions that write registers use the structure to indicate the cycle at which the register will be ready for reading. For example, arithmetic operations that write a register would update the RegReady entry with the current cycle plus 1 for integer operations or 4 for floating point operations. *Ld* instructions, on the other hand, insert a pointer to a reserved MemQueue slot into the RegReady entry. The thread then continues processing instructions.

If an issuing instruction depends on a register whose RegReady entry contains a pointer to an entry in the MemQueue, then the thread returns control to the simulator and blocks on the semaphore associated with the MemQueue entry. When time has been simulated up till the reference completes, the simulator replaces the pointer in the RegReady entry with the current cycle value and returns control to the thread.

Virtual Registers. The discussion above has so far left out an element of the Itanium architecture which significantly complicates our efforts: Itanium instructions refer to virtual registers rather than physical registers. Virtual registers serve two purposes: as a mechanism for implementing a register stack for procedure calls, and as a mechanism for generating more efficient software pipelining code.

At procedure calls, Itanium compilers allocate a set of fresh local registers for use in the procedure and to pass parameters. The new procedure always thinks that its local registers begin at r32. Under the covers hardware maps the virtual registers to new physical registers, essentially sliding a window of virtual register names forward over the physical register file. Upon return from the procedure, hardware pops the stack, sliding the window back so that the logical registers map back to the physical registers they mapped to before the call.

Virtual registers are also used to implement a rotating register mechanism, which enables compilers to generate extremely compact software pipelining code, as in the following example:

```
loop:
        (p17)   st4  [r11]=r33,4
                ...
        (p16)   adds  r32=100,r36
                ...
                br.ctop.sptk.few      loop
```

The value written to r32 by the *adds* instruction in one iteration of the loop is read by the *st4* instruction on the next iteration, though the store lexically precedes the add. Predicate registers also rotate, facilitating pipeline startup and winddown.

Fortunately, both of these cases are marked in the instruction stream by the use of specialized branch instructions: *call* and *ret* mark procedure calls, while *ctop* and *wtop* indicate rotation points.

In our implementation, the RegReady structure is actually associated with an “infinite” array of physical registers, and we keep track of where virtual registers point in the physical registers after pushes, pops, and rotations, via pointers. We can then use a FindReg() routine to map a virtual register to physical register in order to set or read the ready time for the physical register. Since we assume an infinite number of physical registers we do not model the spilling of register contents to memory necessary when hardware runs out of physical registers.

Stores. We have chosen not to model the complexity of a store buffer. Instead we simply use a “write” semaphore to handle the synchronization between threads and the simulator on *st* instructions: on a *st* a thread gives control to the simulator and blocks on the write semaphore. The simulator returns control to the thread after it has time simulated time up until the cycle that the write completes. Acquire and release memory operations are handled in a similar fashion.

4.2. Other Microarchitectural Considerations

Instruction issue and counting cycles. The Itanium architecture has very strict requirements on the number of, and order in which, instructions may issue each cycle. The compiler groups instructions into “bundles” which obey these constraints. One of the runtime values that an analysis routine can receive as a parameter from the instrumentation routine is the value of a stop bit which indicates whether the current instruction is the last instruction of a bundle. With that information our analysis routines can ensure that at most two bundles are issued before incrementing a cycle counter.

	4	16	64
barnes			
Bodies	16K	64K	256K
Insns/Proc (Million)	251	287	313
Refs/Proc (Million)	28	33	36
fmm			
Bodies	16K	64K	256K
Insns/Proc (Million)	822	901	871
Refs/Proc (Million)	36	39	39
moldyn			
Bodies	2K	8K	32K
Insns/Proc (Million)	50	54	59
Refs/Proc (Million)	5	5	6

Table 1. Benchmark statistics.

The cycle counter is also incremented 1) when an issuing instruction depends on a register whose RegReady value is greater than the current cycle, and 2) when control returns to a thread after a *st* instruction completes.

Features not modeled. In not completely modeling the microarchitecture we are of course making some assumptions about our workload:

- We do not model the instruction stream coming from memory, so we are assuming that: 1) L1 instruction cache misses are rare and do not affect performance, and 2) instructions do not conflict with data in the L2 and L3 caches.
- We do not model branch prediction hardware so we are assuming that most branches are predicted correctly.
- Finally, we do not model finite functional units so we are assuming that in our workload, structural hazards are rare and do not affect performance.

5. Accuracy and Performance Evaluation

In this section we present results from experiments with our simulator. The experiments were performed using portions of three applications relevant to our research into the performance of irregular scientific applications on parallel hardware: *barnes*, *fmm* and *moldyn*. All three perform N-body simulations, the difference is in the way the algorithms avoid the $O(n^2)$ complexity of the all-pairs algorithm: *moldyn* uses a “cut-off” heuristic, while both *barnes* and *fmm* make use of a tree data structure to reduce the complexity to $O(nlogn)$ and $O(n)$ respectively.

Table 1 presents some vital statistics from the runs: the number of bodies simulated, the number of instructions simulated on each processor and the number of references in that instruction stream.

All measurements for simulated runs were obtained from dual-processor Itanium2 systems on the Teragrid at NCSA, running a version 2.4 Linux kernel. Data for hardware

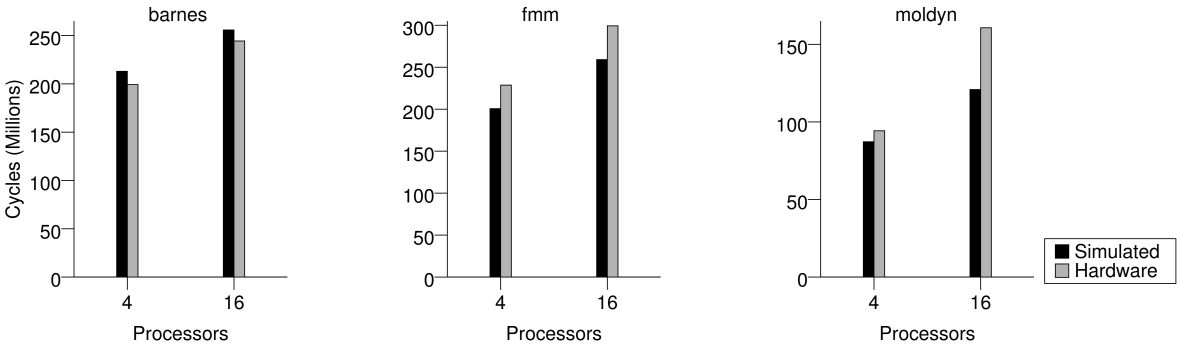


Figure 2. Simulator Accuracy.

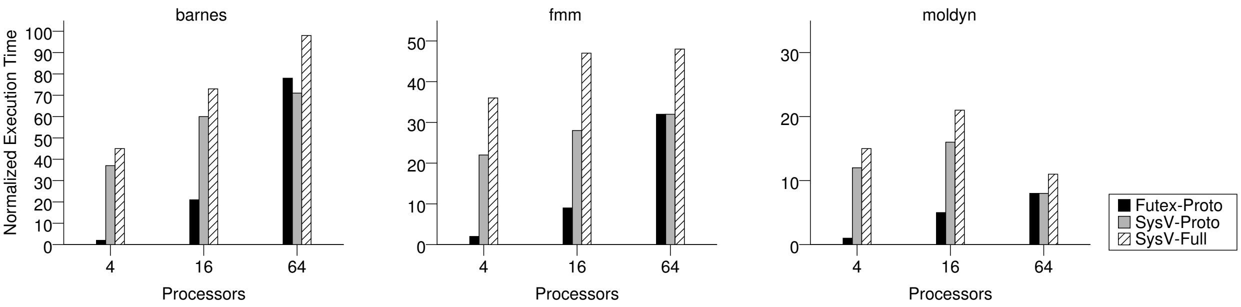


Figure 3. Simulator Performance.

comparisons was gathered from runs on the 64 processor Itanium2-based HP Superdome at the University of Kentucky.

5.1. Accuracy

The first set of results, presented in Figure 2, demonstrate the accuracy of our simulator, *including* the back-end memory system simulator, relative to the hardware that it is attempting to model. Our goal here is not to provide in-depth analysis of the accuracy of the simulator, but simply to demonstrate, through the implementation of the ideas presented in Sections 3 and 4, that our approach is feasible, and that reasonable accuracy is possible.

In support of that claim, Figure 2 shows that in almost all cases the simulated cycle count is within 10% of the actual hardware count as measured with Itanium2 CPU counters. We believe that the one case that's worse, *moldyn* at 16p, is due to the fact that our simulator does not currently simulate system calls: the benchmark features a relatively large number of print statements and there is likely contention for resources as the processor count increases that we are not modeling.

5.2. Performance

Figure 3 presents results describing the performance of the simulator.

Simulators are not created equally: even if they simulate the same hardware and they are likely doing so at different levels of accuracy. This makes performance comparisons with other simulators difficult. Furthermore, our focus in this work is really only the front-end reference generator as opposed to the entire simulator; in particular, we are interested in determining the cost of using semaphores to perform synchronization.

To that end, in our results we first separate out the time spent in the back-end memory system simulator by contrasting the run times of the prototype Pin tool presented in Section 3 with the times of the full-scale simulator including the back-end. Then, to focus in on the cost of semaphores, we normalize all results to the performance of the prototype Pin tool with semaphores *disabled* (i.e., the implementations of *sem_up()* and *sem_down()* simply return control to the caller rather than performing operations on semaphores).

In addition, the figure presents results for two versions of the prototype[1], the first using standard System V semaphores, and the second using futex ("fast userspace mutexes") based semaphores [2]. Counter increments and decrements of futex-based semaphores are implemented at the user-level with the aid of atomic fetch-and-add instructions, so a system call is only required when a thread needs

[1] We were unable to use pthread semaphores due to compatibility issues with Pin. However, it is our understanding that pthread threads are actually implemented as *processes* in Linux, so we would not expect a substantial performance difference relative to the two implementations of semaphores we tested.

to be put to sleep or awakened.

Figure 3 clearly indicates that a large percentage of total simulation time is spent in the front-end, generating references: the full-scale simulation, including the back-end memory simulation, only takes at most 50% longer than the System V semaphore prototype. In other words, reference generation takes about two-thirds of the entire simulation time of the full-scale simulator.

While the performance of the prototype futex implementation appears promising for low thread counts, as the number of threads increases performance degrades to the level of the System V semaphores. This would appear to point to poor scalability of futexes. However, we note that futexes are native to the 2.6 linux kernel, and the version we are using has been back-patched into the 2.4 kernel present on the machines at NCSA. It is thus possible that we are not seeing the full performance potential of futexes.

Interestingly, while *barnes* takes almost 100X longer than the non-semaphore version at 64 processors, *fmm* only takes about 50X. We attribute this to the higher percentage of memory references – and subsequent increased semaphore usage – in *barnes*. Oddly, *moldyn* takes relatively less time at high processor counts compared to the non-semaphore version: this is likely due to the I/O mentioned earlier. I/O is relatively much more expensive for the non-semaphore version since it accounts for a much larger percentage of the total runtime; the effect is enhanced by the fact that our Pin tools do not instrument system calls.

6. Conclusions

We have described our implementation of a multithreaded front-end reference generator for an existing back-end memory system simulator using the dynamic instrumentation tool Pin.

In Sections 3 and 4, we demonstrated how our use of semaphores to synchronize instrumented application threads with the memory system simulator, in conjunction with careful modeling of a few necessary components of the Itanium2 microarchitecture, enables the accurate timing of memory reference generation, both within a thread and between threads.

Then in Section 5 we showed that while cycle-accurate multithreaded simulation is possible using our approach, current semaphore implementations make it an expensive alternative, perhaps prohibitively so. Our experiments indicate that reference generation takes two-thirds of the total time for simulation with our full-scale simulator, including the memory system back-end.

Perhaps with more effort we could find a better implementation of semaphores, or a more clever alternative. However, our experience of frustratingly long waits for simulations of a relatively small number of processors on smallish input sets has led us to conclude that, even if we were able to completely eliminate semaphore overhead, this approach is not viable for the kind of research we want to do. Simulating thousands of processors for billions of instructions is simply not practical on a uniprocessor.

We are currently looking at several alternative approaches. The simplest is to locate application problem areas through runs on hardware, with the aid of CPU and network performance counters, and then simulate large numbers of processors on just that area. The hope is that this approach would considerably reduce the number of instructions that need to be simulated. A more long-term goal is to parallelize the simulator. Synchronization of threads with the memory system simulator might be more efficient, however it is unclear at this point how well the back-end simulator will parallelize.

7. Acknowledgments

The authors would like to thank R. Scott Studham of Oak Ridge National Laboratory for his support while this work was written up. We would also like to thank NCSA and the University of Kentucky for access to their computational resources.

References

[1] E. Dijkstra. Cooperating Sequential Processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, 1968.

[2] H. Franke, R. Russell, and M. Kirkwood. Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux. In *Proceedings of the Ottawa Linux Symposium*, 2002.

[3] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Programming Language Design and Implementation*, Chicago, IL, June 2005.

[4] C. McNairy and D. Soltis. Itanium2 Processor Microarchitecture. *IEEE Micro*, Mar-April 2003.

[5] D. Sorin, M. Plakal, A. Condon, M. Hill, M. Martin, and D. Wood. Specifying and Verifying a Broadcast and a Multicast Snooping Cache Coherence Protocol. *IEEE Transactions on Parallel and Distributed Systems*, 2002.