System Introspection with Hardware Watchmachines

Nicholas Hunt, Brandon Lucia, and Luis Ceze
{*nhunt,blucia0a,luisceze*}*@cs.washington.edu*
University of Washington, Department of Computer Science and Engineering

1 Introduction

Software complexity has led to a need for better tools for understanding system state. Such mechanisms provide *introspection*, which has a wide variety of uses. The success of introspection in Java, tools like Valgrind and gprof, and hardware performance counters demonstrates its value.

Unfortunately, current introspection mechanisms usually impose a high performance overhead or are difficult for programmers to use. For example, a 1000x slowdown using Valgrind for application debugging is not uncommon, and the lack of precise instruction counters has complicated research in deterministic multiprocessing [1, 7, 2]. However, with careful hardware support, introspection can be made both efficient and precise. One such example is the success of debug registers in x86 processors, which allow the system to efficiently monitor memory accesses, and trap to software precisely when a read or write occurs to a monitored address.

In spite of their limitations, we have used existing x86 debug registers to implement several low-overhead program analysis tools[1] for multithreaded code. However, basic event monitoring only scratches the surface: future research across the system stack *demands* better support for introspection.

In this work we are advocating for the addition of new, more sophisticated hardware mechanisms for introspection. The main obstacle to the adoption of such mechanisms is the increase in cost of designing, implementing and verifying additional hardware that doesn't yield obvious or immediate performance benefits. Our main claim is that the benefits provided by system support for introspection justify their treatment as first-order design goals. Our view is shared by others in our community [4]. In line with our claim, this paper outlines one possible direction for future research that illustrates the power of hardware support for introspection.

[1] *More information can be found at http://cs.washington.edu/homes/blucia0a/introspect.html*

2 Hardware Watchmachines

Motivated by the introspection capabilities of hardware watchpoints in modern processors, we propose *Hardware Watchmachines* (HWMs). HWMs are a novel hardware mechanism for monitoring sequences of operations, such as the execution of certain instructions or references to certain data by instructions or coherence messages. The sequence of operations is represented in an HWM as a finite state machine (FSM), where the states encode the progression through the sequence and transitions fire when a specified operation occurs. The system traps to software whenever an FSM reaches an accept state.

Performance Profiling HWMs can be used to profile program performance. Analysis techniques (*e.g.*, static data-flow analysis, or dynamic profiling in a JIT) can be used to identify potentially interesting sequences of operations or data. The HWMs could be configured to encode these sequences as FSMs. Software can maintain a count representing the number of times each FSM traps on an accept state. This analysis will identify hot code and data *sequences*, rather than single code points or objects, without the overhead of software-based techniques. Furthermore, the low-overhead of hardware support may enable profiling of deployed applications.

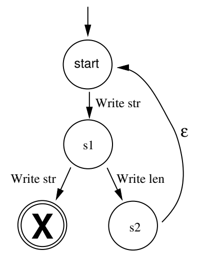
Debugging Prior work has characterized normal program behavior using FSMs [5]. We can use HWMs to detect deviations from normal behavior by constructing a new FSM identical to the original FSM identified as normal behavior, but with a new accept state that represents a deviation from the expected sequence. Transitions to this new state occur whenever an operation is triggered that doesn't correspond to a transition in the original FSM. In the HWM trap handler, debug information (the FSM, for example) can be reported. Section 3 illustrates using HWMs for debugging in more detail.

Avoiding Errors In addition to detecting bugs as described above, we can use HWMs to *avoid* bugs as well (*cf.* prior work [6, 8, 9]). We can create FSMs characterizing the buggy sequence of events, adding accept states before the behavior manifests. When the system traps for

```
f(...) {
  str = "foo"
  len = 3
}

g(...) {
  str = ""
  // BUG: Forgot to
  //  update len
}
```

(a)



(b)

Figure 1: `g()` violates a program invariant requiring all updates to `str` be followed by an update to `len`. Figure 1(b) is a FSM characterizing this invariant.

these accept states, it can take an avoidance action such as a delay or rollback to a checkpoint of correct state.

Testing HWMs can be used to improve software testing. During testing, a system can collect FSMs of observed behavior. HWMs can then be configured with these FSMs, and as testing progresses, the system can avoid the previously observed behavior by changing values used in control flow, or adding delays to change a parallel program's thread schedule. Steering execution away from previously observed behavior is likely to increase test coverage.

3 Debugging with HWMs

As a concrete example, this section demonstrates the use of HWMs to detect buggy behavior in an application. Figure 1(a) shows a segment of code from a program that is supposed to maintain the invariant that all updates to the `str` variable are followed by a corresponding update to `len`. Function `f()` correctly updates both, whereas function `g()` updates only `str`. Failure to update `len` in `g()` could lead to a crash later in the program since `len` is no longer an accurate reflection of the string's length.

In Figure 1(b), we have constructed a FSM that encodes this program invariant. Prior work [5] has demonstrated techniques for identifying such program invariants automatically. Alternatively programmers could explicitly specify such an FSM in their code.

The FSM starts in the state labeled `start` and upon observing a write to `str`, transitions to state `s1`. When the program behaves correctly and updates `len` next, the FSM transitions to `s2` and ultimately back to the start state. However, if a second write to `str` is observed without an interleaving update to `len`, the FSM transitions to the accept state labeled **X**, indicating the invariant was violated. Encoding the FSM in a HWM allows the developer to execute diagnostic code in a trap handler whenever the program violates the invariant.

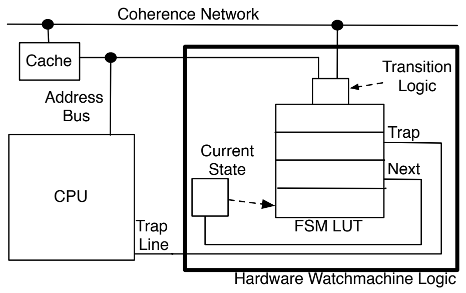


Figure 2: A schematic showing the hardware support for implementing HWMs. Incoming coherence traffic, and addresses accessed by the CPU are inputs to the HWM hardware extensions.

4 Hardware Support

We can implement HWMs with a small amount of hardware support. Figure 2 shows a block diagram of the hardware support we envision. There are no major changes to the CPU implementation. The FSM logic required to implement HWMs can be implemented with a register to track the FSM's current state, and lookup tables (LUTs) to determine the next state for each action. The FSM transition logic is connected to the coherence network and to the CPU's address bus, so when an access or coherence message occurs, the HWM can transition if necessary. Figure 2 shows the logic for one HWM; we envision an array of tens or hundreds of HWMs in a system.

We can limit the amount of unnecessary checking HWMs must do by encoding the set of all addresses involved in its FSM in a hardware bloom filter [3]. The FSM's incoming connections can be deactivated if their values do not appear in the bloom filter.

Hardware limits the size of HWMs – the LUTs are of fixed size, and so some state machines may not fit in the HWM hardware. We can use the bloom filter address hash to help virtualize HWMs. All addresses involved in an excessively large state machine can be encoded in the hash. If an address is found in the hash, and not in the LUT, the HWM could trap to a software handler to determine the next FSM state. For performance, common case transitions should reside in the LUTs, and rarer cases can be handled by software at higher cost.

5 Conclusion

In this work, we have proposed a new hardware introspection mechanism called Hardware Watchmachines, and several applications of this mechanism illustrating its potential. Our goal is to bring attention to the value of hardware introspection mechanisms and the opportunities they present for future research, despite the cost of their design, implementation, and verification.

References

[1] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 53–64, New York, NY, USA, 2010. ACM.

[2] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dos. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[3] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *Proceedings of the 33rd annual international symposium on Computer Architecture*, ISCA '06, pages 227–238, Washington, DC, USA, 2006. IEEE Computer Society.

[4] C. Click. Iwannabit! In *MSPC*, pages 20–25, New York, NY, USA, 2008. ACM.

[5] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *SOSP*, pages 57–72, New York, NY, USA, 2001. ACM.

[6] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-aid: Detecting and surviving atomicity violations. In *ISCA*, ISCA '08, pages 277–288, Washington, DC, USA, 2008. IEEE Computer Society.

[7] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 97–108, New York, NY, USA, 2009. ACM.

[8] J. Wu, H. Cui, and J. Yang. Bypassing races in live applications with execution filters. In *OSDI*, 2010.

[9] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *ISCA*, pages 325–336, New York, NY, USA, 2009. ACM.