Virtues and Obstacles of Hardware-assisted Multi-processor Execution Replay

Cristiano Pereira, Gilles Pokam, Klaus Danne, Ramesh Devarajan, and Ali-Reza Adl-Tabatabai

Intel Corporation

{cristiano.l.pereira, gilles.a.pokam, klaus.danne, ramesh.devarajan, ali-reza.adl-tabatabai}@intel.com

ABSTRACT

The trend towards multi-core processors has spurred a renewed interest in developing parallel programs. Parallel applications exhibit non-deterministic behavior across runs, primarily due to the changing inter-leaving of shared-memory accesses from run to run. The non-deterministic nature of these programs lead to many problems, which will be described in this paper. In order to get a handle on the non-determinism, techniques to record and replay these programs have been proposed. These allow capturing the non-determinism and repeat the execution of a program, hence enabling better understanding and eliminating problems due to non-determinism. In this paper, we argue that an efficient record and replay (R&R) system must be assisted by hardware support. However, adding support for R&R in hardware requires strong justification, in terms of added value to the processor. Up until now, debugging has been the primary motivation used by previous work. We argue that stronger usage models are required to justify the cost of adding hardware support for R&R. Hence the first contribution of this paper are additional usage models (the virtues of R&R) and how we think they will add value to future micro-processors. We think the current lack of focus on wider and more applicable usage models is an obstacle for its adoption. The other obstacle for implementation of hardware-assisted R&R is that some complexities of such endeavor have not been addressed and studied properly. Current research approaches have shortcomings that prevent them from becoming an implementable feature. In this paper, we also identify those shortcomings and indicate research directions to bridge the gap between the research and a product implementation.

1. INTRODUCTION

The wide spread availability of shared-memory multi-core processors compels application developers to write software that explores the parallelism. This is done by dividing the application in multiple threads of execution, which communicate and synchronize through shared-memory. However, developing parallel applications is a complex and error-prone activity. This is because it is impractical to predict and test all possible inter-leavings of instructions that access shared-memory, thus making it hard to ensure the correctness of the program. In addition, contrary to many sequential programs, the execution of a parallel program is non-deterministic. That is, the execution of a program with the same input is not repeatable from run to run; different inter-leavings lead to different control flow and different behaviors even with the same input.

The non-determinism and the difficulty to reason about memory instruction inter-leavings are the source of many problems in parallel programming [10]. Prior research work has shown that test coverage of multi-threaded programs is a hard task [16], given the large number of possible inter-leavings. As a consequence, production code is subject to untested inter-leavings in the field, which leads to concurrency bugs. These bugs are hard to understand and to fix. In order to address this, developer tools to dynamically find data-races [22, 23], atomicity violation [11], and other types of concurrency bugs (e.g. deadlocks [27]) have been created, both in academia and in the industry. In addition, when a concurrency bug manifests in the field [18], it is often hard-to-reproduce it and debug the root cause of the problem. Therefore, parallel programming significantly complicates the tasks of testing and debugging programs.

Furthermore, the non-deterministic nature of parallel applications causes a number of other problems. It makes it difficult to support high-availability system solutions, because it increases the complexity to keep a replica synchronized with a primary execution [13, 25, 26]. Non-determinism also creates challenging problems for post-silicon validation, which relies on techniques similar as PSMI [24]. These techniques enable reproducing a bug, observed in the silicon prototype, for debugging of a newly produced silicon. It is becoming increasingly challenging to repeat such a buggy execution in a functional simulator, in order to identify problems with the prototype. Finally, non-determinism opens up opportunities for vulnerability attacks and makes it difficult to detect such attacks [6].

One prominent technique to cope with the problems mentioned above is the ability to record and replay the execution of a multi-threaded program in a shared memory multi-processor environment. Record and replay (R&R) records a number of events during the execution of a program that enables repeating the execution as in the original run. This enables many nice properties that help mitigate the issues previously described. One obvious application of R&R that has been the focus of academic research is deterministic replay for debugging of parallel programs, which enables repeatability of bugs and other useful debugging enhancements such as time-travel debugging.

In this paper, our first major contribution is a detailed discussion of how R&R can add value to the micro-architecture beyond the debugging usage model. We will describe the usage scenarios that could greatly benefit from such technology and how we envision its use. We believe that debugging is not a strong value proposition to justify, by itself, the cost and the risk of adding hardware support for R&R, and hence other stronger value propositions are required.

The other major contribution of this paper is an analysis of how practical it is to implement R&R in today's micro-architectures. We will argue that the academic proposals [28, 18, 7, 14, 20, 9] currently existing are not practical and lack solutions to many important problems. We will discuss what are the major problems that separate the academic proposals from a real implementation on today's micro-processors. We will also argue that software-only solutions [21] cannot be efficient because the architecture neither exposes nor provides an efficient way to record key events that are needed to reproduce a multi-threaded execution on a shared memory multi-processor.

2. VALUE PROPOSITION OF DETERMINISTIC REPLAY

Hardware support for R&R was proposed two decades ago by Bacon *et al* [2]. Bacon motivated the need for R&R for debugging by arguing that a bug can only be fixed if it can be reproduced when running the program with the same input. Since parallel program runs are not reproducible due to shared-memory inter-leavings, Bacon proposed recording the memory inter-leavings for replay. Since Bacon, many other proposals for R&R were introduced [28, 18, 7, 14, 20, 9]. Most of the proposals also present debugging as the major motivation for R&R.

The recorder of an R&R system needs to log two types of events: 1) recording the input that is not deterministic, that is, inputs that cannot be reproduced by simply executing the program again, such as memory mapped I/O and DMAs [21]; 2) recording the shared-memory interleaving across multiple processors.

Recording the shared-memory dependencies is typically the most complex because it involves observing cache coherence events (e.g. invalidations, data request) in the memory subsystem and interconnect. In today's and past processors, it is impossible to observe such events without hardware modifications [17]. *Because the hardware modifications required involve changes in the memory subsystem and cache hierarchy – components which are typically hard to validate and change –, the justification for its implementation has to be very compelling.* Although we think R&R is a very important component of debugging parallel programs, we do not think debugging alone is a strong enough value proposition to justify the cost of the implementation. The market that would benefit from such a feature for debugging is very limited (only parallel programmers directly benefit from it). Since the feature is likely to increase the cost and the power consumption of the processor, broader usage models are necessary. In this section, we expand beyond the debugging usage model and show how we envision the technology will be useful.

2.1 High Availability Systems

High-availability (HA) of systems can be achieved at multiple levels. In one extreme, systems with hot-swappable components is one form of HA, which allows a system to be available despite reconfiguration or exchange of some components. On the other extreme, *replication and fail-over* provides a full replica of the primary system, which replicates both hardware and software, thus coping with disruptions by moving the workload from the primary to the replica. This ensures continuous availability or minimal disruptions. The level of HA required depends on the financial consequences of a down-time; if the cost (lost lives for emergency-response organizations or money for businesses; HRG - Harvard Research Group [8] estimates an average of $10,000 per hour of down-time in the year 2000) outweighs the price of HA, then it makes sense. While there are other levels of HA, we will not discuss them in this paper. In this paper, we focus on the replication and fail-over scenario, which we think is critical for many enterprises. Applications that have built-in fault-tolerance and high-availability features, such as many databases (e.g. Oracle database), do not require additional HA solutions, because the application itself can ensure continued service, without any disruption. Other server applications, however, may not have built-in solutions. These include web servers and mail servers, for example. If those applications are critical and service disruption means big fiscal consequences for the company operating them, HA solutions which are transparent to the application are fundamental.

There are companies that provide transparent HA solutions in the market (e.g. Stratus [25], Marathon [13], and VMWare [26]). Stratus, for example, uses proprietary hardware to keep two running CPUs in lockstep, where the instructions in each CPU are processed simultaneously, in addition to special hardware to handle redundancy for all other I/O components. If one CPU fails, the other takes over. These solutions are expensive, hard to maintain and are becoming increasingly complex, especially with the proliferation of multi-core architectures. The sources of non-determinism with multi-core architectures are hard to capture, in particular maintaining the same order of shared-memory accesses.

The other, more prominent solution for high-availability due to its reduced cost and flexibility, is virtualization. This is the approach taken by Marathon and VMWare. Academics have also been looking into VM-based HA solutions [5]. In this context, two virtual machines are kept in sync with one another, both running the same operating system and application. The VMs are kept in sync by either frequently exchanging checkpoints or by exchanging logs containing the non-deterministic events processed by one machine, which are then used to keep the replica in sync. The replica always runs a bit behind of the primary, but close enough to be able to continue the execution. The primary and the replica exchange data through an *availability-link*. Cully *et al* [5] made a qualitative comparison between the two approaches. Frequent check-pointing the state of the VM was shown to have performance overhead and bandwidth requirements that can affect performance of low-latency applications. We speculate that Marathon [13] offers a similar solution, though it is not clear. Cully *et al* acknowledge R&R as an alternative, but point out that a software only solution has overhead that is too prohibitive because shared-memory communication order has to be tracked and propagated from one machine to the other.

We conjecture that providing hardware support for tracking shared-memory communication solves this problem and enables the use of R&R as a solution for high-availability. VMWare [26] has shown that R&R can be done very efficiently in software for single-processor machines. With hardware support for R&R, tracking shared-memory communication can be done with low overhead, hence enabling the technology on modern processors. R&R can then be used for HA, thus enabling another usage model, which opens opportunities for new markets, therefore adding value to the microprocessor.

2.2 Post-silicon validation

Post-silicon validation heavily relies on transfer of failures from silicon to either RTL model or tester environment for debug [24]. Processor state, event and pin trace information collected using logic analyzers are used to reconstruct and replay the complete processor execution. However, the development of current generation of multi-core processor and platform topologies has reached a stage wherein the various bus structures are not economically viable points of instrumentation for debug. In certain classes of validation scenarios, the number of visibility points have increased from observation of a single bus (like a FSB) to numerous inter-processor or processor-memory links. In other validation scenarios, due to tighter integration of the various building blocks to a single system on chip, there is very limited observation points for debug. This is a compound problem which results in:

1. Increased cost for logic analyzer probes for observation;
2. Associated complex platform instrumentation;
3. No direct co-relation of bus observed data with the internal CPU activity due to inter-processor or memory protocols;
4. Necessity to handle higher bus or link bandwidths which leads to data collection errors.
5. Complex ordering of transactions obtained across multiple channels.

Hardware-assisted R&R coupled with hooks for probe-less data extraction (without a logic analyzer) provides an alternative to tackle the problem areas highlighted above for validation. To facilitate logic debug, we will need R&R to capture clock accurate information on asynchronous events, shared memory interleaving transactions and any source of non-determinism like DMA updates to memory. Bus functional models can then be created to run along with RTL to inject the memory/event information observed in order to detect and debug logic issues. Architecturally valid models can also be used to debug software, concurrency issues which manifest in validation world due to incorrect inference of architecture specification, porting issues when moving test content from one family of processors to another, or environment related issues which have not been accounted for. In these scenarios, R&R can be used in a first triage as a backbone to isolate the "false alarms" due to software test content issues. After initial triage, R&R can also be used for a more deep-dive logic-based debug, if necessary. In essence, time to debug and root-cause a sighting in post-silicon validation environment can be decreased significantly with hardware-assisted R&R.

As modern microprocessors and platform topologies become more complex, with increased focus on reducing the validation schedule, although hardware-assisted R&R comes up with the overhead of extra silicon real-estate, design and development time, this overhead will be compensated with reduced validation costs and quicker triage of issues, resulting in faster time to market.

2.3 Debugging and Testing

R&R was originally motivated by the cumbersome task of debugging parallel programs on multi-processor architectures. It is notoriously difficult to reproduce concurrency bugs due to non-determinism. Even if the bug is reproducible, it can be very hard to root cause the source of the bug and fix it. Reproducing the bug in-house is a difficult task faced by many parallel computing programmers. Debugging tools and traditional addition of statements to print out variables and values often hide the occurrence of the problem, making the task time-consuming and frustrating. An even more daunting task is to reproduce the bug which manifests itself only in production environments due to specific environment configurations. R&R assisted by hardware proposes a solution to all debugging problems above, in a multi-processor environment, with very low overhead. This enables the mechanism to be "always on" during development of applications, in contrast to previous software-only proposals. Whenever a test fails, the programmer can use the logs to repeat the execution exactly, as many times as necessary, and then root cause the problem. Replay can be integrated with single-step debuggers such as GNU GDB or Visual Studio and also combined with tools for dynamic detection of concurrency bugs (e.g. [23, 22, 11]). This significantly improves the experience of debugging parallel code since the analysis is conducted on a deterministic execution that guarantees the bug is present. If the replay logs are independent of the operating system, as in BugNet [18], a bug can be captured at the customer site and debugged at the developer site. This allows debugging customer problems in house, without having to repeat the environment in which the bug was exposed. In addition, it allows for dynamic analyses to be executed during replay, hence decoupling the analysis from the execution and minimizing perturbations to the original program run [4]. Replay can also be used with tools that aim at profiling and performance analysis of the execution (e.g. PinPlay [19], based on Intel's Pin framework [12]).

In addition to the integration with debuggers and dynamic analysis tools for concurrency issues, R&R can also be used to improve testing of parallel programs. In particular, systematic testing of parallel programs has surged with new momentum in this new era of multi-processors. Testing of concurrent code has been shown to cover a limited number of thread inter-leavings, hence limiting the efficacy and predictability (due to non-determinism) of testing. Tools for systematic testing such as Microsoft CHESS [16] aim at overcoming this limitation by intelligently perturbing the execution to generate buggy inter-leavings. Once a buggy inter-leaving is detected, CHESS has methods to reproduce it, provided the execution is on a single-processor machine. One obvious contribution of R&R is the ability to capture the buggy execution on a multi-processor machine, allowing to replay-debug it as mentioned previously. Another usage model is the ability to replay a non-buggy execution and during replay intelligently choose where to perturb the execution with the goal to expose more concurrency bugs. The ability to replay and analyze the code enables profiling the runs to better choose where to perturb.

2.4 Intrusion Detection

Maintaining computers systems secure is not an easy task. The wide availability of anti-virus software and the stricter security concerns of today's enterprises are not enough to keep systems from being broken into. As a result, the ability to analyze attacks after the system intrusion occurs is important because it enables understanding the vulnerability of the system and consequently allows the proper fix of potential damage created by the attacker. Recording non-deterministic events during execution allows a system administrator to replay the execution in order to understand the chain of events that led to the intrusion. Dunlap *et al* introduced the concept of deterministic replay as a potential solution for the intrusion detection problem [6]. As in previous examples, recording the execution in multi-processors environment without hardware support incurs prohibitive overhead. Therefore, in order to reproduce the execution accurately and efficiently, R&R assisted by hardware is required.

3. PRACTICAL ISSUES

In this section, we discuss the practicality of adding hardware support for R&R into a microprocessor. There exist two hardware approaches for recording the input non-determinism. One relies on logging a subset of load instructions values that cannot be reproduced during replay. BugNet [18] is the state of the art. To identify which load values to log, either the caches are modified with a bit to indicate when to log or the cache lines have to be logged for every load-miss. Another approach proposes OS specific changes to record non-deterministic events such as memory mapped I/O and interrupts. Capo [15] is a recent approach and the state of the art.

Various software-only proposals of R&R were introduced in the past [21]. Solutions that capture the data-race non-determinism typically incur large run-time overhead. The main reason is that events which reveal shared-memory dependencies (i.e. cache coherence) are not exposed to the software layer. As a result, the software has to either monitor read and write instructions to memory and emulate coherence or track dependencies at page-level granularity. Both schemes are slow and would limit the applicability of R&R. Nagarajan *et al* [17] proposed exposing cache coherence to the software through interrupts. Even so, the projected runtime overhead is still unacceptable for some usage models. Hence we think the only way to achieve reasonable performance is through hardware-assisted recording of memory races.

For recording the shared-memory inter-leavings, most of the past academic proposals [28, 29, 18, 7, 14, 20, 9] rely on observing coherence requests to identify when a shared-memory dependency occurs. Once detected, each technique provides a clever way to decide when to log a dependency, striving to imply as many dependencies as possible to minimize log sizes and number of logged dependencies. The combination of input non-determinism and shared-memory inter-leavings is sufficient for deterministically replay a parallel program on multi-core architectures.

While the previous proposals are a significant step towards the implementation of R&R in hardware, we think there are other complex challenges that need to be overcome to bridge the gap between these proposals and a real implementation. In this section, we will discuss these challenges and point out research directions to overcome them.

3.1 Hardware Complexity and Intrusiveness

A feature proposal that requires changes in the hardware needs to be carefully evaluated before deciding whether the feature will be added or not. Minimal hardware complexity and performance overhead are necessary requirements because the cost of development and validation of a microprocessor is high. In addition to strong justification and usage models to add value, which we discussed in the previous section, an implementation that has minimal complexity and intrusiveness is required. The current proposals made significant advancements to simplify R&R in hardware. Proposals such as [7, 20] do not require modification to the cache. Instead, they use bloom-filters [3] to track dependencies across processors. All these proposals rely on piggybacking timestamps on cache coherence messages. This contributes to design complexities and potential performance overhead. The scheme proposed in [20] significantly minimizes the additional traffic overhead due to piggybacking on cache coherence messages. We think a viable implementation should strive to minimize coherence traffic overhead to be acceptable. Nonetheless, a careful and convincing evaluation of the performance impact and perturbation of workload execution has not been properly done yet.

While the performance of logging input data for single-core processors using VMMs has been studied in the past [26], the performance of logging memory-races using hardware has been neglected. The hardware records the memory race logs and keep them in a buffer, inside the processor. However, those logs have to inevitably be written out to memory, so that the logs can be stored in disk. The performance and perturbation impact of writing logs to memory has not been studied sufficiently. The performance impact depends on the logging frequency (which depends on the applications being recorded) and on the policy to write the logs out. The addition memory operations to write the logs will contend with the memory operations of the program itself, hence perturbing the original execution.

3.2 Memory Consistency Models

All previous proposals, with the exception of [29] and [14], assume a multi-core processor with sequential consistency (SC) [1] memory model. Looking at most micro-architectures available today, we cannot find a major processor which implements sequential consistency as its memory model. Hence there is a clear disconnect between the proposals and the reality of current processors. RTR [29] provides one step towards handling Total Store Order (TSO). The approach suggests that one can detect a sequential consistency violation and then log the value of the violating load for correct replay. While the strategy is sound, the mechanism to implement sequential consistency violation is not described. In addition, logging the values of sequentially consistent violating loads can potentially add up to the complexity. DeLorean [14], though not assuming traditional sequential consistency, assumes a transaction memory based execution substrate, which is not currently available in mainstream processors.

The prior proposals rely on using committed instruction counts to establish dependencies across cores. On a SC system, this is not a problem because when a store instruction commits, it makes its value globally observable to other cores. Hence the commit count matches the order in which memory operations are executed and made available to other cores. Handling a non-sequential consistent memory model is hard for two reasons: 1) first it breaks the assumption that committed instruction counts are enough to order memory instructions, and as a consequence, it requires more complex schemes to identify when instruction counts need to be augmented with additional information to allow proper replay; 2) it requires modification to the core in order to properly identify when sequential consistency violations happen and hence can potentially increase the complexity of logging shared-memory inter-leavings.

3.3 Instruction Atomicity

With instruction atomicity violation, we refer to the fact that some instructions in the processor do not execute atomically, i.e. their side-effects can be exposed to other processors before the instruction has completed execution. For example, on a cache access, a load or a store operation can result in several other individual memory operations being executed. This is typically the case when a memory operation on one processor updates data to a shared address that splits over two cache lines. In that case, another processor might read corrupted partially updated data. An R&R system that tracks dependencies between instructions cannot define a correct dependency between the two instructions. This is especially true for Intel processor families, where macro instructions are visible to the programmer but micro instructions are executed by the processor, hence creating the problem.

In such case, the R&R system must be able to record dependencies on finer granularity, e.g. at micro-architecture execution level. For the same reason, a software-based replayer cannot simply execute one instruction before the other, but instead must emulate the same behavior that occurred during the instruction atomicity violation. Therefore it is clear that R&R needs to detect the occurrence of instruction atomicity violation and properly log how it should be replayed.

3.4 Replay Speed

The replay speed refers to how fast the execution of a program can be replayed as non-deterministic events are injected and threads are controlled to enforce the proper order of shared-memory accesses. Since the most focused usage model for R&R has been debugging, there is always an implicit assumption that the replay speed in which a program is executed does not matter. While this is generally true for debugging in some contexts, we do not think replay speed can be neglected. For instance, if we are applying dynamic analysis to find bugs during replay, we do not want to compound the slowdown of the analysis with the slowdown of replay. For the testing scenario where we want to explore different thread inter-leavings, a fast replay would benefit and maximize the number of candidate inter-leavings to try out. This adds convenience and improves the types of techniques that can be applied. More importantly, for other usage models such as high-availability, replay speed is not only desirable, but it is essential to enable the technology. A replica needs to execute at a speed that is fast enough to catch up with the primary, so that downtime is minimized. The issue of fast replay speed is not trivial and warrants careful investigation. For example, for VM-based replay, synchronizing cores can be an expensive operation if the execution has to fall-back into the VMM every time a synchronization is required. To mitigate this, speculative techniques or even hardware support could provide a solution.

4. CONCLUSIONS

R&R is a technique that allows recording non-determinism during the execution of a parallel program on a multi-core processor. The run of the program is then reproduced by injecting these non-deterministic events during replay. Hardware assisted R&R has the potential for capturing the non-determinism with very little overhead. However, hardware modifications require strong justification to offset the risks of implementing it. In this paper, we argued that multiple usage models for such technology are required to justify its implementation. Therefore, we described multiple usages for hardware-assisted R&R that would make a stronger case for its deployment. In particular, we described the following usage models: 1) Implementation of high-availability systems, using replication and fail-over; 2) Optimization and cost reduction of post-silicon validation; 3) How it can improve debugging (we do think debugging is a useful usage model, but we do not think debugging by itself is a strong enough justification) and testing of parallel programs, by allowing capturing hard to reproduce and understand concurrency bugs, and by having the potential to expose concurrency bugs from a parallel program run; 4) And finally by allowing intrusion detection analysis, during replay, to better understand how a vulnerability was exploited.

Various proposals for the implementation of hardware-assisted R&R were presented in the past. However, we still believe many technical obstacles still need to be overcome so that one can exploit all the potential virtues of the technology. We think the complexity and intrusiveness of the implementation need more careful analysis. We also pointed out that an implementable solution for systems that assume relaxed memory models has not been proposed, hence preventing the technology to become mainstream. Another source of complexity is a problem we refer to as instruction atomicity violation, which exposes the side-effects of not-yet-completed instructions execution to other processors. This adds complications to both the recorder and the replayer. This problem has not been addressed at all. Finally, we discussed how replay speed is fundamental to enable some of the usage models proposed, in particular high-availability.

5. REFERENCES

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66 – 76, 1996.

[2] D. F. Bacon and S. C. Goldstein. Hardware-assisted replay of multiprocessor programs. In *Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging*, pages 194–206. ACM Press, 1991.

[3] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7), July 1970.

[4] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[5] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: high availability via asynchronous virtual machine replication. In *NSDI'08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 161–174, Berkeley, CA, USA, 2008. USENIX Association.

[6] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*, pages 211–224, New York, NY, USA, 2002. ACM.

[7] D. Hower and M. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *Proceedings of the International Symposium on Computer Architecture*, 2008.

[8] HRG. Hrg insight: The total cost of downtime. `http://www.hrgresearch.com`.

[9] D. Lee, M. Said, S. Narayanasamy, Z. Yang, and C. Pereira. Offline symbolic analysis for multi-processor execution replay. In *Micro-42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 564–575, New York, NY, USA, 2009. ACM.

[10] E. A. Lee. The problem with threads. *Computer*, 39:33–42, 2006.

[11] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: detecting atomicity violations via access interleaving invariants. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 37–48, New York, NY, USA, 2006. ACM.

[12] 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 *Proceedings of the Conference on Programming Language Design and Implementation*, 2005.

[13] Marathon. Marathon technologies website. `http://www.marathontechnologies.com`.

[14] P. Montesinos, L. Ceze, and J. Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Proceedings of the International Symposium on Computer Architecture*, 2008.

[15] P. Montesinos, M. Hicks, S. King, and J. Torrellas. Capo: A software-hardware interface for practical deterministic multiprocessor replay. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[16] M. Musuvathi. Systematic concurrency testing using chess. In *PADTAD '08: Proceedings of the 6th workshop on Parallel and distributed systems*, pages 1–1, New York, NY, USA, 2008. ACM.

[17] V. Nagarajan and R. Gupta. Ecmon: exposing cache events for monitoring. In *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*, pages 349–360, New York, NY, USA, 2009. ACM.

[18] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *Proceedings of the International Symposium on Computer Architecture*, 2005.

[19] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. Pinplay: A framework for deterministic replay and reproducible analysis of parallel programs. In *International Symposium on Code Generation and Optimization (CGO)*, pages 2–11, April 2010.

[20] G. Pokam, C. Pereira, K. Danne, R. Kassa, and A.-R. Adl-Tabatabai. Architecting a chunk-based memory race recorder in modern cmps. In *Proceedings of the International Symposium on Microarchitecture*, 2009.

[21] G. Pokam, C. Pereira, K. Danne, L. Yang, S. King, and J. Torrellas. Hardware and software approaches for deteministic multi-processor replay of concurrent programs. *Intel Technology Journal*, 13(4):20–41, Fall 2009.

[22] P. Sack, B. E. Bliss, Z. Ma, P. Petersen, and J. Torrellas. Accurate and efficient filtering for the intel thread checker race detector. In *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 34–41, New York, NY, USA, 2006. ACM.

[23] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[24] I. Silas, I. Frumkin, E. Hazan, E. Mor, and G. Zobin. System-level validation of the intel pentium m processor. *Intel Technology Journal*, 07(2):20–41, May 2003.

[25] Stratus. Stratus technologies website. `http://www.stratus.com`.

[26] VMWare. vsphere availability guide. `http://www.vmware.com`.

[27] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *OSDI '08 Proceedings. 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 281–294, December 2008.

[28] M. Xu, R. Bodik, and M. Hill. A flight data recorder for enabling full-system multiprocessor deterministic replay. In *Proceedings of the International Symposium on Computer Architecture*, 2003.

[29] M. Xu, R. Bodik, and M. D. Hill. A regulated transitive reduction (rtr) for longer memory race recording. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.