Owl: Next Generation System Monitoring

Martin Schulz
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
schulzm@csl.cornell.edu

Brian S. White, Sally A. McKee
Computer Systems Lab
Cornell University
{bwhite,sam}@csl.cornell.edu

Hsien-Hsin S. Lee
School of Electrical and Computer Engineering
Georgia Institute of Technology
sean.lee@ece.gatech.edu

Jürgen Jeitner
Department of Informatics
Technische Universität München
jeitner@in.tum.de

ABSTRACT

As microarchitectural and system complexity grows, comprehending system behavior becomes increasingly difficult, and often requires obtaining and sifting through voluminous event traces or coordinating results from multiple, non-localized sources. Owl is a proposed framework that overcomes limitations faced by traditional performance counters and monitoring facilities in dealing with such complexity by pervasively deploying programmable monitoring elements throughout a system. The design exploits reconfigurable or programmable logic to realize hardware monitors located at event sources, such as memory buses. These monitors run and writeback results autonomously with respect to the CPU, mitigating the system impact of interrupt-driven monitoring or the need to communicate irrelevant events to higher levels of the system. The monitors are designed to snoop any kind of system transaction, e.g., within the core, on a bus, across the wire, or within I/O devices.

Categories and Subject Descriptors: C.4 [Performance of Systems]: Measurement Techniques

General Terms: Performance, Measurement

Keywords: Autonomous Performance Monitoring, Reconfiguration, Performance Analysis

1. INTRODUCTION

As microarchitectural and system complexity grows, comprehending system behavior becomes increasingly difficult. Users nonetheless anticipate future *autonomic* systems that adapt dynamically to provide greater performance, avoid or repair transient faults, intercept adversarial attacks, and reduce system management costs. A substantial gap remains between what system designers can now provide—in terms of introspective data itself and the means of processing, analyzing, and visualizing it—and what users expect and will soon demand.

A simple example of application memory performance monitoring highlights the disparity between the current state of the art, where probes report only symptoms of poor performance, and the goal of a self-aware system that discovers the cause. Rather than placing the burden on a user to query hardware registers or instrument binaries, a continuously monitored [1] system might employ a unobtrusive background monitor that detects spikes in some metric, e.g., cache miss rate. The monitor then instantiates additional monitors on the L1-L2 and memory buses to capture individual L1 and L2 misses, which can be correlated. For example, thrashing between the caches for specific data indicates conflict misses. This more precise miss characterization helps in selecting an appropriate optimization, such as cache-conscious data placement [4].

Monitoring is most often applied for performance considerations, but future systems will also demand it for security, fault detection, and maintenance. Consider memory fault isolation, where loads, stores, and jumps are guarded to ensure validity of the memory access or jump target. This may be done by manual instrumentation [30] or by a logically similar hardware method that "macro expands" instructions into guarded equivalents [6]. An active monitor can provide this functionality with no overhead. When an access check fails, the monitor signals the operating system, which might either terminate the process or decide to enlist more monitors to sandbox the potentially malicious code.

Current monitoring facilities are ill-equipped to handle the above scenarios. Traditional hardware monitors are restricted to a fixed set of events and cannot perform sophisticated, online analysis. Their deferral to software for processing comes at the expense of costly and frequent interrupts or of loss of accuracy due to sampling. Further, they rely on a proper handling of software events, which is clearly not adequate for monitoring software and system faults. Software solutions [13, 17, 20, 35] rely on sampling, offline traces, or heavyweight instrumentation resulting in high system overhead or loss in accuracy.

In order to provide the introspection necessary to understand and efficiently use future architectures, we need

CF'05, May 4–6, 2005, Ischia, Italy.

new *hardware* solutions for system monitoring. In recent years, several academic projects have focused on such hardware (e.g., solutions by Prvulovic and Torrellas [22] or Xu et al. [33]). However, each of these approaches provides specialized monitoring capabilities for specific problems or questions. No *general* framework has yet been proposed.

In this paper, we propose Owl, a generic and programmable monitoring framework. It consists of reconfigurable or programmable logic elements deployed throughout the system. The user can program these monitors to acquire performance data without system interference, perform application-specific data analysis, and write results directly into main memory without interrupting the processor. The latter is necessary to minimize system perturbation. Owl's programmability is flexible enough to encompass existing performance counters and extant or proposed monitoring techniques, as well as new, previously infeasible monitoring applications. The latter include monitoring addresses on memory buses, collecting a complete history of individual cache replacement decisions, snooping I/O bus activity, or checking assertions on all memory accesses.

As its key design principle, Owl differentiates between user-defined analysis modules, which perform monitoring and analysis such as data aggregation and compression, and monitor capsules, which provide a standard interface between the module and the hardware. The programmability and autonomy of the modules support event processing close to the source, domain-specific monitoring, and the ability to react or adapt to observed events or application phases. Modules are implemented in reconfigurable or programmable logic, such as programmable microengines or FPGAs, within capsules. This flexibility enables less invasive hardware implementations of existing software techniques and, ultimately, more sophisticated monitoring than previously possible.

The pervasive deployment of capsules throughout the system provides a user with alternate views of the same event (e.g., following a page miss through caches to disk) or simultaneous views of correlated events (e.g., declining IPC and branch prediction accuracy). The capsules' standardization allow the same analysis algorithm to examine events throughout the system, despite potential dissimilarity among monitored devices. Flexibility in deploying capsules results from the simple assumption that they are only able to observe system events in the form of transactions. Thus, they may be attached to any transaction interface, including memory buses, I/O buses, or network interfaces.

In Section 2 we describe existing hardware approaches for system monitoring. In Section 3 we present requirements for a next-generation monitoring facility and describe the design of Owl, a framework leveraging autonomous, programmable monitors to achieve those requirements. In Section 4 we describe several sample applications for novel memory monitoring capabilities, including efficient memory access logging, memory access characterization, and dynamic pattern recognition. In Section 5 we present results discussing the small system perturbation caused by using Owl, and we show the hardware complexity of a sample Owl module.

2. RELATED WORK

Most current architectures include at least rudimentary hardware assists for system monitoring, usually in the form of counter registers. The UltraSPARC IIi [27] is an archetype of more advanced systems: it exports two performance counters that can monitor any of 20 predefined events. Counters may be programmed to raise an interrupt upon overflow.

Counter-based techniques suffer common shortcomings [25]: too few counters, sampling delay, and lack of address profiling. Modern systems try to address these deficiencies. For instance, the Pentium 4 [15, 26] comes with 18 performance counters. In addition, it tags μops when they trigger certain performance events. These events are not counted until the μops are retired, ensuring that spurious events from speculative instructions do not pollute samples. This microinstruction-targeted approach also overcomes sampling delay.

However, these mechanisms can only be employed to collect aggregate statistics using sampling. It is not possible to react to single events or to collect additional data, e.g., load target addresses of memory accesses. This prohibits a direct correlation of observed events with the data structures causing the events. The Itanium Processor Family [14] and other newer systems overcome this deficiency and allow the detection of such events, e.g., memory accesses or branch mispredictions. The access mechanisms provide microarchitectural event data, but these data are delivered to the consuming software through an exception for each event. The process using the information experiences frequent interrupts, and system perturbation occurs at many levels. Overheads costs of these mechanisms, particularly with respect to time, limit the extent to which software can reasonably exploit them.

Many academic projects have focused on novel performance monitors for interconnection systems. Martonosi et al. [19] propose using the inherent coherence/communication system in the Stanford FLASH multiprocessor as a performance monitoring tool. *FlashPoint* is embedded in the software coherence protocol to monitor and report memory behavior in cache miss counts and latencies, inducing a 10% execution time overhead. In the SHRIMP project [18], performance monitoring boards allow each node to collect histograms of incoming packets from the network interface. A threshold-based interrupt is used to signal the application software and operating system to take proper action in response to a specified event. The SMiLE project [16] includes a SCI-based hardware monitor to detect memory layout problems in NUMA clusters; the monitored information helps to guide data layout and transparent data migration.

3. Owl: PROGRAMMABLE, SYSTEM-WIDE MONITORING

The research community's shift from using simple aggregate metrics characterizing entire applications, e.g., cache miss rate, to more advanced statistics such as reuse distance [34], periodicity [13, 24], and correlating multi-layer analysis [31], signals that traditional hardware monitors are becoming insufficient. Further, new goals such as security and system health maintenance require monitoring different classes of events and eliciting varied responses, including the recruitment of other monitors to focus on specific symptoms. Discovering trends in system behavior requires monitors to be long running, and therefore unobtrusive, as well as capable of correlating events. The monitors must produce results judiciously to avoid overwhelming the system with inordi-

nate amounts of data. These considerations point to an autonomous monitoring facility that is flexible enough to handle varied and evolving system behavior, leading to the following requirements:

- **Coordinated, cooperative system-wide monitoring:**
  Observation of a single data source provides a myopic view, which may be sufficient to establish the symptom but not the cause of a problem. For example, if a monitor detects declining IPC, it might spawn additional monitors to examine branch prediction accuracy and cache miss rates. Indication that the latter is a problem might lead to additional coordinated monitoring.

- **Hardware assist for data aggregation:**
  Data probes have the potential to collect copious data, especially when applied broadly to highly used resources, such as monitoring all L1 accesses. To manage such data, the monitoring system must be capable of performing at least preliminary analysis online in hardware. This can reduce data by using compression, aggregation, or statistical analysis. Data aggregation need not entail a loss of information, but rather can remove obfuscating data to bring a trend to light.

- **Domain-specific monitoring through programmability:**
  One type of monitor cannot address all possible scenarios: some kind of programmability or reconfiguration capabilities are thus required. These enable the system to retarget monitors to perform new tasks or to gear to a specific device, application, or application phase. For example, a programmer may wish to monitor accesses to a specific data structure to determine if it exhibits poor locality so that it may be targeted by future optimizations, such as data regrouping [11]. Such fine-grained monitoring would be greatly aided by compiler support [12].

- **Data delivery without process interruption:**
  To minimize system perturbation, monitors must deliver their output asynchronously and deposit it into main memory. This contrasts with most currently implemented schemes, which either require a process interruption and event handler invocation for each observable event or are limited to sampling. The amount of monitoring data injected into the system, and hence the amount of system perturbation, depends on the semantics of the analysis module. Designing module-embedded analytical techniques therefore requires an estimation of system perturbation. We study data injection rates to provide general overhead results intended to guide future designs and to allow early decisions regarding their feasibility.

3.1 Owl Architecture

The Owl monitoring framework is designed to fulfill these requirements. As its key architectural principle, it splits the monitoring functionality into two parts: programmable capsules, which are attached to the actual data probes or sensors; and analysis modules, which are loaded into the capsules to perform data aggregation and preprocessing. The

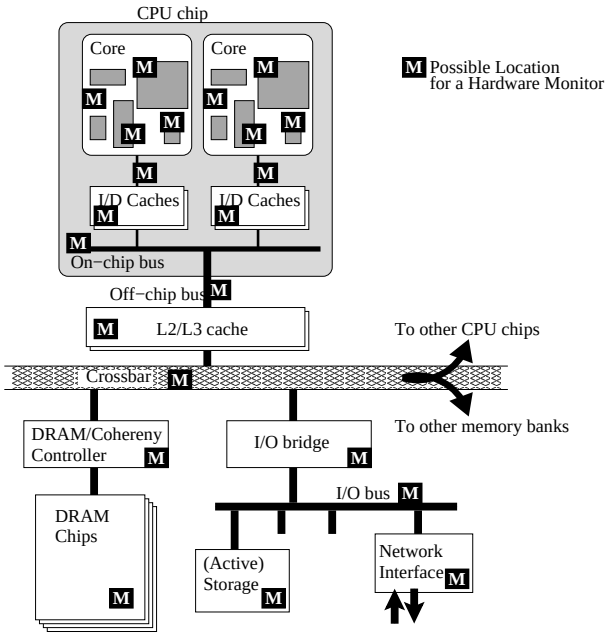

Figure 1: Possible monitor capsule locations for system-wide monitoring

capsules may be included in any component providing interesting data and hence can be located throughout the system, as illustrated in Figure 1.

Each capsule is connected to one or more data probes or sensors embedded within the monitored component (see also Figure 2). These probes provide the respective data in event form, e.g., memory bus transactions, coherence events, disk accesses, accesses to the reorder buffer for out-of-order microprocessors, updates to the branch predictor, or rollback events in case of misspeculation. The event, including any relevant parameters such as addresses, values, or event types, is received by the *Architecture Hardware Interface* and translated to a standardized interface connecting the capsule and the analysis modules, the *Monitor Hardware Interface*. A module loaded into the programmable part of a capsule uses this interface to receive monitored data. This use of a system-wide, standardized interface allows the execution of any analysis module in any capsule, independent of its location and concrete data source, and thereby enables the reuse of analysis and aggregation techniques.

Analysis modules are loaded and instantiated from a system-wide library of modules. This library can be dynamically extended and can contain application-specific modules. Library management and concrete module selection is left to the system components or tools using Owl for their monitoring purposes. Once loaded, a module may further be customized through memory-mapped configuration registers in each capsule. When activated, the capsule directs the probed data to the module, where it is preprocessed, analyzed, aggregated, compressed, or even sampled, which may be the desired functionality, rather than a limitation imposed by the monitor framework. The results of this analysis step are forwarded to the capsule for storage.

Due to Owl's programmability, each module contains application-specific data aggregation and filtering tech-

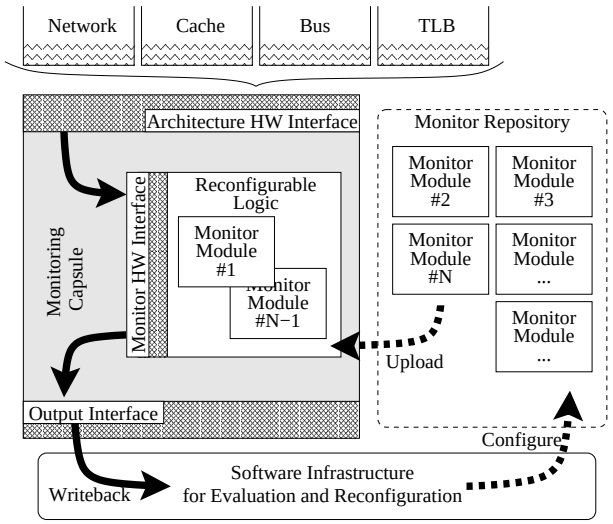

Figure 2: Architecture of a monitoring capsule

niques that are uploaded directly to the location of the data probes. This enables the modules to acquire and process every individual event without system perturbation. Only the results of the data aggregation are written to main memory, and the total size of these results is usually significantly smaller than the total size of all observed events. Further, the storage of the monitor results is initiated autonomously by the monitor module: it hands any data packet containing monitor data to the capsule through a standardized interface. The capsule then generates a memory packet, injects it into the memory system, and stores in a contiguous area of main memory organized as a ring buffer. Consumers of monitor data can then read the data from this memory region and process it asynchronously. The ring buffer itself is reserved by the operating system, and its size depends on both location of the capsule and the expected event rate. In order to avoid buffer overruns, each capsule has the ability to signal consumers using interrupts when a ring buffer is about to become full.

3.2 Providing Programmability

The programmable nature of the modules is the key to their flexibility and filtering capabilities. Several means of achieving programmability exist, e.g., dynamic selection of predefined components, use of microprogrammable processing cores, or use of reconfigurable logic in the form of FPGAs. We consider the first option too restrictive, since only a predefined set of monitoring modules would exist. The latter two options provide similar capabilities, and we will explore both avenues. Here we focus on FPGA-based solutions, since they provide a low-level and direct interface to the hardware. Furthermore, any concurrency expressible in hardware designs is directly exposed to the analysis modules and thereby permits an easy and efficient use of pipelining. This can be used to ensure a high event handling rate despite complex analysis operations for each event.

Reconfigurable hardware is increasingly used in modern architectures to complement general-purpose processors. An industry trend towards hybrid-reconfigurable systems indicates the potential and viability for architectures like Owl. For example, SRC Computer platforms are architected with Direct Execution Logic (DEL), comprised of dynamically reconfigurable functional elements (Multi-Adaptive Processors) intended to maximize parallelism of a given application code. OctigaBay Systems (now part of Cray, Inc.) uses one Xilinx Virtex-II Pro FPGA per node as an application-specific accelerator to perform vector operations. In the embedded market, several chip manufacturers provide single-chip solutions combining processor cores, such as PowerPC or ARM, with FPGAs enabling easy and efficient customization of processors [2]. In general, we see a trend towards faster and more efficient FPGAs, and recent studies have shown that some FPGAs can compete with the clock frequencies of most modern microprocessors [28].

3.3 Design Considerations for Modules

Monitoring efficiency depends on a module's ability to intelligently preprocess and "semantically" reduce data traffic, i.e., to extract the essential information before injecting the result into the memory system. Further, hardware complexity limitations restrict the nature of the analysis techniques that are loaded into reconfigurable logic. The design of each module must reflect this balance between traffic reduction and hardware complexity.

In general, modules are not intended as full-featured analytical engines, but as semantic preprocessors and partial data analyzers. As a result, the data stream will be reduced by filtering interesting events and aggregating data as needed by the consumer. This contrasts with traditional compression techniques or static aggregation steps, which do not make use of semantic information. As a consequence, they either push the processing completely into software or potentially lose interesting events. Recognizing the power and flexibility inherent in preprocessing data close to the source within the computational and memory constraints of the capsule are, as yet, a subtle art. Future work must systematically address the division of labor between an analysis module and high-level software analysis tools.

3.4 Programming Monitoring Modules

Programming FPGAs using low-level hardware description languages is a complex task often only accessible to the advanced hardware designer. In order to make Owl accessible to a wider range of users, we must take further steps. While not part of the initial design, we present a few directions we are pursuing in ongoing work.

For the common user, we will provide a comprehensive module library containing common analysis modules. Users can extend this library at any time with new modules developed by any programmer. The result is similar to kernel modules, which can be loaded into the kernel at runtime to extend the system's capabilities without the user having to know details about the kernel itself or implementation details of such modules.

In addition, we will work on high-level abstractions to design or compose monitoring modules. As a first step, we are developing a toolset that enables users to combine monitor submodules (e.g., histogram generators, counters, compression algorithms, or pattern detectors) into new modules. Once composed, the toolset generates and compiles the new hardware design as specified by the user and adds it to the module library. As a next step, we will investigate high-level programming approaches, such as C to HDL compilers,

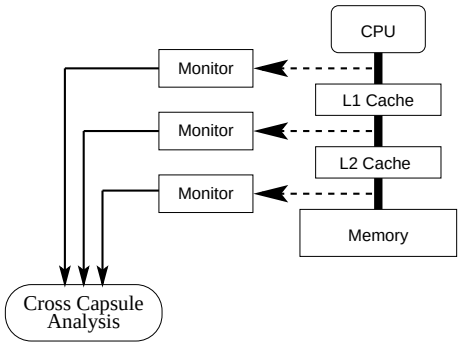

Figure 3: Multi-level Monitoring in the Memory Hierarchy: Location of Monitoring Capsules

to make Owl's full flexibility available to the user. Several projects and products already address such compilation approaches, and we plan to build on top of them. So far, these approaches have had limited success when applied in a general scenario due to inefficiencies when compiling arbitrary algorithms into hardware. In Owl, however, we are faced with a less complex problem, since all modules will naturally follow a given pattern—they target online event processing. We expect such compiler solutions to be more efficient when used to generate Owl modules.

4. USAGE SCENARIOS FOR Owl

Owl provides a versatile and highly flexible monitoring infrastructure that can be used for a variety of scenarios. It can be used to implement existing hardware counters as well as most hardware monitors currently proposed in the literature. Owl is therefore downward compatible, providing a true superset of extant monitoring functionality. Further, Owl allows users to transfer hybrid hardware/software solutions into a single hardware module, eliminating the often costly software component. For instance, sampling and profiling based on performance counters can be implemented inside the capsule connected to the associated data probe.

In addition, Owl can be used to implement previously infeasible or overly expensive analysis techniques. In the following we describe three such examples that aggregate and pre-analyze memory traffic: the creation of memory access traces for logging and debugging, the generation of memory access and cache miss histograms for memory performance tuning, and data structure access monitoring and pattern recognition. These represent three typical classes of algorithms, namely classic filtering and compression; partial, custom aggregation for the creation of histograms; and pattern detection and extraction.

For these examples, we assume Owl capsules are distributed in the memory system. More specifically, we assume a three-level memory hierarchy and attach a capsule between each level, as illustrated in Figure 3. We focus on monitoring the memory system initially, since it remains one of the most significant performance bottlenecks in modern architectures. The overall Owl architecture, however, can be deployed system-wide, and modules similar to those described below can be used in other system components.

4.1 Memory Access Logging

Software-based memory trace facilities [10, 21, 29] are heavily used for workload characterization and simulation. In most cases, however, they are implemented based on architectural simulation and hence are limited by high simulation overhead and by the fact that they fail to capture complete system-level behavior.

These limitations can be overcome using a monitoring module designed for trace generation. To avoid excessive memory traffic induced by full logging, monitoring modules can perform loss-less compression, ranging from well established schemes, such as run-length encoding, to more sophisticated approaches including semantic trace compression [7] or load-value predictor-based compression [3]. All these can be implemented in hardware to reduce traffic significantly.

The modules can also be used to perform aggressive filtering, logging only accesses corresponding to particular code or address regions or occurring within a given time window. In the extreme case, a module only maintains a short time access log in the form of a ring buffer. Triggered by certain events, e.g., illegal accesses, this log is written to memory and delivered to a corresponding tool for further analysis. This mechanism provides new means for debugging at all system levels, e.g., through transparent assertion testing or value inspection. It can also be applied for security tests, e.g., detection of buffer overruns. In the latter case, the monitoring module is provided by the operating system as a system component to transparently and non-intrusively control any running application.

4.2 Memory Access Histograms

The ability to monitor addresses of memory accesses allows the creation of memory access histograms, as depicted in Figure 4 (left) for a sort algorithm (RADIX of SPLASH-2 [32]). These histograms show the number of accesses per address or cache block to each level of the memory hierarchy over a period of time and thereby enable observations on individual data structures. In the example, the different access behavior for the two main arrays (cache block 200-8500 and 8500-16500), as well as to the global configuration data (cache blocks 16500-17300), are clearly distinguishable. Furthermore, the availability of histograms for each level of the memory hierarchy enables the computation of cache miss rates on a per address basis, as shown in Figure 4 (right). The example shows that both arrays are faced with high L1 cache miss rates of around 30% interleaved with regular spikes to particular blocks with almost 100% miss rate. Through reverse mappings of addresses to data structures, this information is used to identify data structures with poor cache behavior and hence focus performance analysis and optimizations.

4.3 Dynamic Pattern Recognition and Reduction

Periodic program behavior presents a rich opportunity for program characterization and optimization [13, 24]. Loops involving non-affine iterative steps or indirect accesses are a barrier to static periodicity detection, and long-range correlations may be obscured by procedure boundaries. Online approaches are attractive in combating these limitations and can be implemented within the proposed framework in the form of analysis modules. They are capable of detecting periodic behavior and delivering these observations for con-

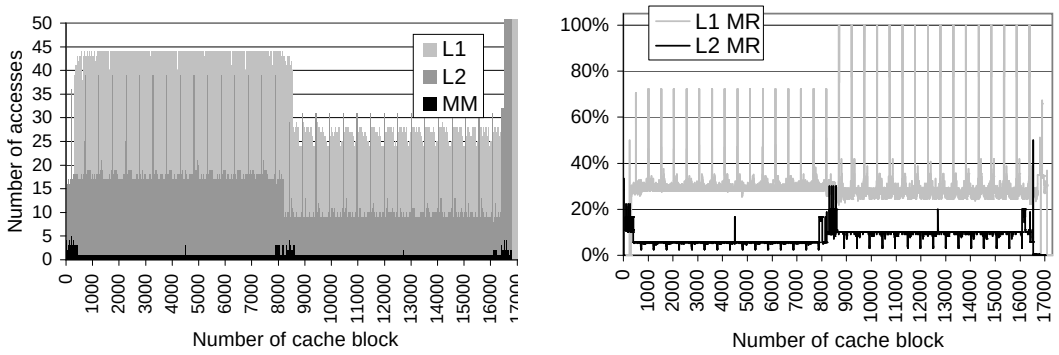

Figure 4: Memory access histogram (left) and cache miss rate diagram (right) for sort algorithm (radix sort)

sequent system optimization, such as prefetching. At the same time, the repeating character of detected patterns can be used for "semantic" data reduction, since each pattern, once recognized, can be represented by a single data point. Such aggregation is significant, as it reduces the amount of data that need be communicated by the monitor, visualized by the user, stored on disk or memory, or analyzed by a tool or algorithm.

A pattern detection module first recognizes arbitrary, repeating memory patterns and then performs a semantic data reduction, representing each complex pattern as a single point. We apply this technique to an off-line Alpha 21264 trace of `ammp`, an n-body molecular dynamics code from the SPEC 2000 suite. More specifically, our study focuses on the `mm_fv_update_nonbon` function, which dominates computation time for many inputs, including the SPEC reference input set used here.

A monitoring module first extracts repeating subsequences from a stream of load target addresses by applying a standard longest common subsequence algorithm. Assuming generous 32-bit table entries to record distances between load addresses, a module implementing such a pattern detection algorithm would require a pattern match table with 200×100 entries (approximately 80KB in raw form). The elimination of dead table entries, the application of partial evictions, and the use of width-adaptive registers will decrease this space requirement.

The results of pattern detection and aggregation of `ammp` are shown in Figure 5. The top pane of the figure shows a trace of addresses from the execution of `mm_fv_update_nonbon`. Applied to this data, the monitoring module captures a number of recurring patterns. The most interesting one is shown highlighted in the middle pane, which is a blown up section of the top band in the original trace. These patterns iterate over a larger address space, interrupted by gaps. These observations can be related to dynamic control flow such as if statements (i.e., the gap in addresses) and loops (i.e., the repetition of the pattern).

Having recognized the pattern, little additional information is gained by the inclusion of every one of its constituent accesses in a trace. Rather, the aggregation of the entire pattern to a single point reveals the underlying periodicity (as shown in the bottom pane of Figure 5) and at the same time significantly reduces the amount of data transfered from the monitor to the ring buffer. In this example, the 72 individual accesses comprising the pattern are reduced to a single

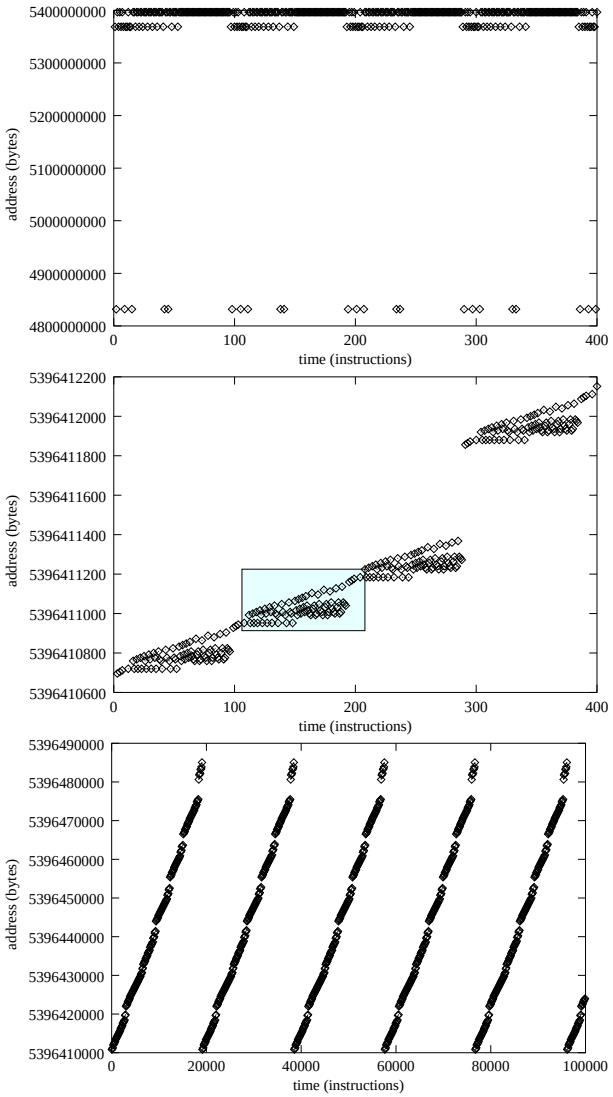

Figure 5: `ammp` substructure: trace of `mm_fv_update_nonbon` (top), trace restricted to address range of top band with repeating access pattern highlighted (middle), extended trace with access pattern aggregated to a single point (bottom)

representative access to the base address of the pattern, resulting in an injection rate of 1/72.

The behavior so exposed indicates that a second round of pattern matching should be applied, which would distill the entire loop body to only a few points. Applying this multi-resolution approach recursively allows users to expose higher-level program structure and periodicity without the usual attendant increase in data.

5. EXPERIMENTS AND RESULTS

In order to be successful, a reconfigurable monitoring infrastructure, like Owl, must facilitate the implementation of monitoring modules with low hardware complexity and guarantee low system perturbation. In the following, we present experimental results showing that Owl satisfies these conditions.

5.1 Hardware Complexity

To study the complexity of monitoring models, we implemented a VHDL version of the histogram module described above. Memory histogram generation conceptually requires an individual counter for each memory location. We use an associative counter array to avoid maintaining a large number of counters. The module contains a small set of counters that are dynamically associated with observed addresses. When the number of observed addresses exceeds the number of available counters, the module frees the least recently used counter, writes its contents to main memory as a partial result, and reuses the freed counter for new events. This drastically reduces the required frequency of writebacks to memory, and ensures low system perturbation. The concrete number of counters used within the module depends on the capsule location and the observed traffic patterns. Across a range of applications, our experiments show that 32 to 128 counters are sufficient to achieve an injection rate of less than 1/8 [23].

Using a Xilinx XC4085XLA with 40K gates, the analysis logic in its current version consumes about 66% of the available resources. Considering that modern FPGA chips contain up to 8M gates, this monitor can be realized with a modest chip real estate budget. Furthermore, aggressively pipelining the design of the analysis module enables the monitor to execute at the maximum frequency of the FPGA.

5.2 System Perturbation

To characterize perturbation of monitored applications we implemented the Owl framework within the memory system of SimpleScalar (SimAlpha version 4.0) [9] and varied injection rates to simulate different monitoring modules. We implement capsules at each level of the memory hierarchy using two different techniques for injecting the recorded data into the memory stream: the first simply injects the traffic into the memory system at the location of the capsule as regular memory packets, while the second ensures that monitor packets bypass all on-chip caches and uses a separate off-chip memory controller to store the monitor data. The former minimizes the architectural impact when introducing Owl, and hence is a suitable solution for commodity systems, while the second results in less perturbation, but comes at a significantly higher implementation cost, and hence is only an option in specialized high performance systems.

Our simulator is configured as closely as possible to the validated model of a Compaq DS-10L Alpha Server, as described in previous studies [8, 9]. The memory system is a 64KB, two-way associative L1 cache with 64B lines and three-cycle latency followed by a 2MB direct-mapped L2 cache with a 13-cycle latency. The benchmarks are selected from the SPEC 2000 benchmarks to achieve a broad coverage of behaviors and they use the SPEC reference input sets. In order to achieve reasonable simulation times we rely on SimPoint [24]. More specifically, we use SimPoint in the original version with multiple simpoints per code [5] to guarantee the highest possible accuracy. The complete configuration of the benchmarks along with the input sets for the cycle-accurate simulation are shown in Table 1. In case of multiple possible input sets, we choose the one with the lowest error rate under SimPoint execution.

Benchmark	Type	Input	#Instr.
164.gzip	SPECint	ref/program	256B
171.swim	SPECfp	ref	1563B
175.vpr	SPECint	ref/route	240B
176.gcc	SPECint	ref/expr	15.2B
177.mesa	SPECfp	ref	492.2B
179.art	SPECfp	ref/470	198.9B
186.crafty	SPECint	ref	264B
188.ammp	SPECfp	ref	1924B
254.gap	SPECint	ref	473B
256.bzip2	SPECint	ref/program	58B

Table 1: SPEC benchmark parameters

baseline: no monitoring activated
1/1 injection rate; naive injection (naive)
1/8 injection rate; naive injection (naive)
1/64 injection rate; naive injection (naive)
1/1 injection rate; separate off-chip memory (sepmem)
1/8 injection rate; separate off-chip memory (sepmem)
1/64 injection rate; separate off-chip memory (sepmem)

Table 2: Test cases

The discussion of modules in Section 4 reflects the dependence of traffic injection rates on the analysis technique employed by the module: 1/1 for full address logging, 1/8 as an upper bound for the memory access histogram generation, and 1/64 for pattern matching. The configuration space is shown in Table 2 and the results in Figure 6.

Most codes are capable of hiding injection rates of 1/8 and less without system perturbation, while some of the more memory intensive codes require rates as low as 1/64. However, these codes can be executed at higher injection rates if the user can tolerate higher system perturbation. Similar to the selected results above, introducing a separate monitoring memory could eliminate most perturbation for higher injection rates.

This overhead is still orders of magnitude lower than any software scheme. In addition, as described in Section 4, most applications of logging schemes do not require full logs over the application runtime, but rather concentrate on specified subregions. This will reduce the amount of traffic the system has to deal with, enabling full access logging in these cases.

6. CONCLUSIONS AND FUTURE WORK

Researchers have proposed myriad analysis techniques, and practitioners have made efficient use of these for sys-

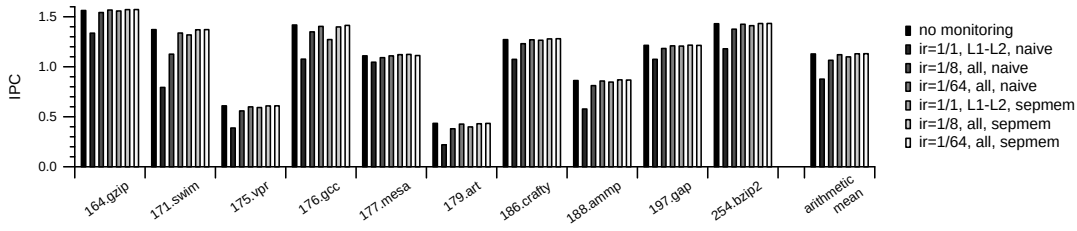

Figure 6: IPC across several SPEC benchmarks and with varying configurations

tem adaptation and optimization. Nonetheless, the capabilities of current systems fall far short of what is needed and expected for reliable, efficient, complex computing systems. Most existing approaches suffer from the inflexibility and limitations of the monitoring hardware or rely on purely software implementations with extreme overhead costs. All of these techniques would benefit from the existence of a unifying, general-purpose monitoring architecture that allows a better balance of functionality between hardware and software, and even allows the user to configure each technique to the scenario at hand, to be activated on demand.

In this paper, we propose such a flexible, general infrastructure for reconfigurable monitoring, and we illustrate how it can be used for understanding memory behavior to optimize access patterns and data placement. Further, we show that such monitoring can be implemented with very low overhead, causing little or no system perturbation, and thus minimal observable performance penalties.

The framework itself is novel: it separates system probes from data analysis functionality, allowing the latter to be dynamically controlled by the user in the form of analysis modules. For instance, in the memory monitoring examples, these modules perform online preprocessing of the observed memory addresses, and deliver results to the consumer for post-processing—without necessitating any process interruptions. Our results show that a monitoring system with autonomous data delivery has a relatively small impact on system performance and that with lower injection rates the overhead becomes negligible.

The feasibility study performed in the course of this work demonstrates the viability of the general approach. As the framework was designed as a general monitoring facility, we believe its success in the specific context of memory analysis will extend to more pervasive system-wide monitoring—and towards better understanding of system behavior in general.

7. ACKNOWLEDGMENTS

This work was supported by the National Science Foundation under award ITR/NGS-O325536 and by a DOE fellowship, provided under grant number DE-FG02-97ER25308. Part of this work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48 (UCRL-CONF-209855).

8. REFERENCES

[1] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 357–390, Oct. 1997.

[2] D. Andrews, D. Niehaus, and P. Ashenden. Programming models for hybrid CPU/FPGA chips. *IEEE Computer*, 37(1):118–120, Jan. 2004.

[3] M. Burtscher and M. Jeeradit. Compressing extended program traces using value predictors. In *Proceedings of the 2003 International Conference on Parallel Architectures and Compilation Techniques*, pages 159–169, Oct. 2003.

[4] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *Proceedings of the 8th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 139–149, Oct. 1998.

[5] B. Calder, T. Sherwood, E. Perelman, and G. Hamerley. Simpoint. http://www.cs.ucsd.edu/~calder/simpoint/, 2003.

[6] M. L. Corliss, E. C. Lewis, and A. Roth. DISE: A programmable macro engine for customizing applications. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 362–373, June 2003.

[7] L. DeRose, K. Ekanadham, J. Hollingsworth, and S. Sbaraglia. SIGMA: A simulator infrastructure to guide memory analysis. In *Proceedings of IEEE/ACM Supercomputing '02*, Nov. 2002.

[8] R. Desikan, D. Burger, and S. Keckler. Measuring experimental error in multiprocessor simulation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 266–277, June 2001.

[9] R. Desikan, D. Burger, S. Keckler, and T. Austin. Sim-alpha: a validated, execution-driven Alpha 21264 simulator. Technical Report TR-01-23, Department of Computer Sciences, The University of Texas at Austin, 2001.

[10] Digital Equipment Corporation. *ATOM User Manual*, Mar. 1994.

[11] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the 1999 ACM SIGPLAN Conference*

on Programming Language Design and Implementation, pages 229–241, May 1999.

[12] C. Ding and Y. Zhong. Compiler-directed run-time monitoring of program data access. In *First ACM SIGPLAN Workshop on Memory System Performance (MSP)*, pages 1–12, June 2002.

[13] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *Proceedings of the 2003 International Conference on Parallel Architectures and Compilation Techniques*, pages 220–231, Sept. 2003.

[14] Intel. *Intel Itanium Architecture Software Developer's Manual*, 2000.

[15] Intel. *Intel Architecture Software Developer's Manual Volume 3: System Programming Guide*, 2002.

[16] W. Karl, M. Leberecht, and M. Schulz. Optimizing data locality for SCI-based PC-clusters with the SMiLE monitoring approach. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 169–176, Oct. 1999.

[17] J. Marathe, F. Mueller, T. Mohan, B. de Supinski, S. McKee, and A. Yoo. METRIC: Tracking down inefficiencies in the memory hierarchy via binary rewriting. In *Proceedings of the First Annual Symposium on Code Generation and Optimization*, pages 289–300, Mar. 2003.

[18] M. Martonosi, D. W. Clark, and M. Mesarina. The SHRIMP performance monitor: Design and applications. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (Sigmetrics '96)*, pages 61–69, May 1996.

[19] M. Martonosi, D. Ofelt, and M. Heinrich. Integrating performance monitoring and communication in parallel computers. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (Sigmetrics '96)*, pages 138–147, May 1996.

[20] T. Mohan, B. de Supinski, S. McKee, F. Mueller, A. Yoo, and M. Schulz. Identifying and exploiting spatial regularity in data memory references. In *Proceedings of IEEE/ACM Supercomputing '03*, Nov. 2003.

[21] A.-T. Nguyen, M. Michael, A. Sharma, and J. Torrellas. The augmint multiprocessor simulation toolkit for intel x86 architectures. In *Proceedings of 1996 International Conference on Computer Design*, October 1996.

[22] M. Prvulovic and J. Torrellas. Reenact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 110–121, June 2003.

[23] M. Schulz, J. Tao, J. Jeitner, and W. Karl. A proposal for a new hardware cache monitoring architecture. In *Proceedings of the Workshop on Memory Systems Performance (MSP 2002)*, June 2002.

[24] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, Oct. 2002.

[25] B. Sprunt. The basics of performance-monitoring hardware. *IEEE Micro*, pages 64–71, July/August 2002.

[26] B. Sprunt. Pentium 4 performance-monitoring features. *IEEE Micro*, pages 72–82, July/August 2002.

[27] Sun Microsystems. *Ultra-SPARC-IIi User's Manual*, 1997.

[28] J. Teifel and R. Manohar. Highly Pipelined Asynchronous FPGAs. In *Proceedings of the ACM International Symposium on Field-Programmable Gate Arrays*, pages 133–142, Feb. 2004.

[29] J. Veenstra and R. Fowler. MINT: A front end for efficient simulation of shared-memory multiprocessors. In *Proceedings of the 2nd International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 201–207, Jan. 1994.

[30] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216, Dec. 1993.

[31] R. Wisniewski, P. Sweeney, K. Sudeep, M. Hauswirth, E. Duesterwald, C. Cascaval, and R. Azimi. Performance and environment monitoring for whole-system characterization and optimization. In $P = AC^{[}2]$ *Conference on Power/Performance Interaction with Architecture, Circuits, and Compilers*, pages 1–10, Oct. 2004.

[32] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.

[33] M. Xu, R. Bodik, and M. Hill. A flight data recorder for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 122–135, June 2003.

[34] Y. Zhong, C. Ding, and K. Kennedy. Reuse distance analysis for scientific programs. In *Proceedings of the Workshop on Languages, Compilers, and Runtime-Systems for Scalable Computers*, Mar. 2002.

[35] Y. Zhong, S. G. Dropsho, and C. Ding. Miss rate prediction across all program inputs. In *Proceedings of the 2003 International Conference on Parallel Architectures and Compilation Techniques*, pages 79–90, Sept. 2003.