Ubiquitous Memory Introspection

Qin Zhao[1,2] Rodric Rabbah[3]
Saman Amarasinghe[1,4] Larry Rudolph[1,4] Weng-Fai Wong[1,2]

[1] Singapore-MIT Alliance
[2] National University of Singapore
[3] IBM T.J. Watson Research Center
[4] Massachusetts Institute of Technology

Abstract

Modern memory systems play a critical role in the performance of applications, but a detailed understanding of the application behavior in the memory system is not trivial to attain. It requires time consuming simulations and detailed modeling of the memory hierarchy, often using long address traces. It is increasingly possible to access hardware performance counters to count relevant events in the memory system, but the measurements are coarse-grained and better suited for performance summaries than providing instruction level feedback. The availability of a low cost, online, and accurate methodology for deriving fine-grained memory behavior profiles can prove extremely useful for runtime analysis and optimization of programs.

This paper presents a new methodology for Ubiquitous Memory Introspection (UMI). It is an online and lightweight methodology that uses fast mini-simulations to analyze short memory access traces recorded from frequently executed code regions. The simulations provide profiling results at varying granularities, down to that of a single instruction or address. UMI naturally complements runtime optimizations and enables new opportunities for online memory specific optimizations.

We present a prototype runtime system implementing UMI. The prototype has an average runtime overhead of 14%. This overhead is only 1% more than a state of the art binary instrumentation tool. We used 32 benchmarks, including the full suite of SPEC CPU2000 benchmarks, for evaluation. We show that the mini-simulations accurately reflect the cache performance of two existing memory systems, an Intel Pentium 4 and an AMD Athlon MP (K7). We also demonstrate that UMI predicts delinquent load instructions with an 88% rate of accuracy for applications with a relatively high number of cache misses, and 61% overall. The online profiling results are used at runtime to implement a simple software prefetching strategy that achieves an overall speedup of 64% in the best case.

1. Introduction

The migration from offline to runtime optimizations provides the unique ability to perform workload-specific optimizations that are tailored to end-user scenarios. This paper presents a practical simulation-based profiling methodology for use in an online setting. The methodology calls for i) identifying frequently executed program regions during execution, ii) selectively instrumenting some of their operations and profiling their execution, and iii) periodically triggering a fast online mini-simulator to analyze the recorded profiles and derive detailed performance summaries, all while the application is running. It is not uncommon for offline simulators to use similar techniques to reduce simulation time with sampling and functional fast forwarding [4]. A key observation inspiring this work is that similar ideas and even simpler heuristics are possible at runtime.

The paper is focused on delivering detailed profiles of the application behavior in the memory hierarchy, at the level of individual instructions and addresses. Virtually all optimizations that attempt to mitigate the memory bottleneck rely on accurate application-specific profiles of the memory hierarchy performance. For example, data prefetching techniques are up to 60% more effective when they are targeted at high miss rate memory references. Similarly, locality enhancing optimizations can significantly benefit from accurate measurements of the working sets size and characterization of their predominant reference patterns. Ubiquitous Memory Introspection (UMI) provides online and application-specific profiling information that is necessary for runtime memory-centric optimizations. As a result and for the first time, UMI makes it possible for traditionally offline simulation-based optimizations to run online.

1.1. Common Practice

It is not uncommon to use simulators to model the memory system behavior of benchmarks and application codes.

Table 1. Running time for a range of HW counter sample sizes, compared to UMI.

Sample Size	0 (native)	1 (UMI)	10	100	1K	10K	100K	1M
Time (s)	35.88	35.90	773.81	152.71	48.21	39.20	36.30	36.24
% Slowdown	–	0.06	2056.66	325.61	34.36	9.25	1.17	1.00

Simulators are versatile and malleable, and can provide a wide range of profiling detail. They are however invariably slow, and often prohibitively time consuming for large and realistic applications. For example, Cachegrind [20] requires days to fully simulate the SPEC benchmark suite using the reference input workloads. As a result, detailed simulations are used for offline performance tuning and optimizations. They remain impractical for runtime memory optimizations whether in a virtual machine, or in a general-purpose code manipulation environment.

1.2. Worst Case Scenario For HW Counters

Increasingly, researchers have turned to hardware performance counters to quickly generate performance profiles that identify opportunities for optimizations. The counters are extra logic added to the processor to track events (e.g., cache misses) with little overhead. Many existing processors provide hardware counters, and because of their low overhead, they may naturally complement online optimization systems. However counters are designed to provide coarse summaries that span thousands of instructions. They add significant overhead to provide context-specific information, and gathering profiles at instruction granularity is an order of magnitude more expensive. This is because the counters generate interrupts when they saturate at a specified limit known as the sample size. The runtime overhead of using a counter increases dramatically as the sample size is decreased. A case study using one of the more memory intensive applications from the SPEC CPU2000 benchmark suite shows a $20\times$ slowdown compared to native execution when operating at near instruction level granularity. Table 1 summarizes the benchmark running time for `181.mcf` operating on its training input, with a single counter for measuring the number of primary cache misses it suffers. The sample size is varied from an allowed minimum size of 10 to an arbitrary maximum of 1M. The results were collected using PAPI [23] on a 2.2GHz Intel Xeon processor.

It is readily apparent from the results that hardware counters are not well suited for the extraction of fine-grained details such as context information surrounding a cache miss (e.g., an address sequence leading to a cache miss for individual instructions).

1.3. Practical Alternative

UMI offers the intriguing alternative of observing short runtime sequences of memory references, and analyzing them using online mini-simulations to reasonably approximate the memory system behavior of the host architecture. The simulations can provide results to guide heuristics used in online performance tuning mechanisms.

Ubiquitous memory introspection is carried out by judiciously instrumenting hot code regions to profile the memory references that occur while the code executes. The emphasis on frequently executed code applies the same insight at the heart of existing virtual machines and binary instrumentation and optimization systems. The instrumented code regions run periodically, and in bursts, to produce very short memory reference profiles. The profiles provide a brief history of the memory reference patterns. They are periodically analyzed using simple heuristics that are effective and practical for an online setting.

The analysis can provide a high level of detail, comparable to offline simulators. For example, a fast cache simulator can process the profiles to identify load instructions that are often likely to miss in the cache. Alternatively, a profile may record the sequence of addresses referenced by a single instruction, and then used to discover patterns suitable for prefetching. UMI provides a level of profiling detail that is not possible with hardware counters. Table 2 contrasts UMI to existing profiling methodologies.

Table 2. Tradeoffs in profiling methodologies.

	Simulators	**HW counters**	**UMI**
Overhead	very high	very low	low
Detail Level	very high	very low	high
Versatility	very high	very low	high

1.4. Contributions

We present in this paper a conceptual framework for ubiquitous memory introspection. We also present an implementation of UMI that is transparent, fully automatic, and lightweight. It is adaptive, accurate, inter-procedural in nature, and yields context and flow sensitive profiling information.

We used DynamoRIO [5] to build our prototype system, although implementations in similar binary instrumentation and optimization tools such as Pin [18] or Valgrind [20], or in a Java virtual machine, are also feasible. The prototype inherits DynamoRIO properties, and can be readily applied to programs running on existing commodity hardware. It does not require any programmer or user intervention, nor does it require any modification or knowledge

of the program source code and symbol information, and hence it works on any general-purpose program, legacy and third party binaries. As an example, we successfully used the prototype to profile several commonly used Linux desktop and server applications.

Our main observations and results are summarized as follows:

- Periodic online mini-simulations of short memory reference profiles recorded from hot code regions are sufficient to yield actionable profiling information for runtime memory performance optimizers.
- We present a full prototype of a system implementing UMI. We show that the average runtime overhead is 14% for the entire SPEC CPU2000 benchmark suite using the reference input workloads. This overhead is only 1% greater than existing state of the art binary instrumentation tools.
- We show that for our two evaluation platforms (Intel Pentium 4 and AMD Athlon K7), there is a strong correlation between cache miss rates measured using UMI and hardware counters.
- We also show that UMI leads to high correlation with offline cache simulations. It identifies high miss rate load instructions with 61% accuracy compared to the Cachegrind cache simulator. The prediction accuracy is significantly higher (88%) for applications that are memory intensive. The profiling results are used in an online optimization scenario to implement a simple software prefetcher that outperforms the Pentium 4 hardware prefetcher in the best case.

UMI offers a practical and versatile alternative to existing profiling methodologies. It naturally complements runtime optimizations, and provides opportunities for new kinds of online optimizations that are otherwise largely infeasible. Optimizations that use UMI can replace or enhance hardware techniques such as prefetchers and cache replacement policies. UMI also provides opportunities to introduce novel, dynamic, and adaptive optimization techniques. As a radical example, UMI can be used to quickly evaluate speculative optimizations that consider multiple what-if scenarios. This can complement existing online compilers, and may create opportunities for online learning-based compilation and optimization systems.

1.5. Paper Organization

We present a conceptual overview of UMI in Section 2 and implementation details in Sections 3-5. In Section 6 we present an evaluation of our prototype in terms of its runtime performance, and present empirical correlation between mini-simulations and hardware counters. In Section 7 we show a more detailed correlation study that measures how well UMI predicts high miss rate load instructions. In Section 8 we demonstrate how to use the online profiling information to implement a simple software prefetcher. Sections 9 and 10 present related work and conclude the paper with final remarks.

2. Conceptual Framework

The thesis for this work is that online mini-simulations using short memory reference profiles from hot code regions can characterize memory system performance with sufficient detail. The key insight enabling UMI is that numerous virtual machines and binary instrumentation and optimization systems already exist, and they provide a natural setting for online introspection and profile-driven optimizations.

There are three basic components to a system that implements ubiquitous memory introspection: the region selector, the instrumentor, and the profile analyzer.

Region Selector. The first component is the hot code region selector. It dynamically identifies representative code regions to instrument. Typically such regions are frequently executed code fragments in an application. They may encompass loops or entire methods in a Java Virtual Machine, or sequences of basic blocks promoted to an instruction trace in binary code manipulation systems such as Pin or DynamoRIO. Virtually all runtime code manipulation systems provide some form of hot code selection. We believe they are readily amenable for UMI, and in essence provide this first component for free.

We further refine the hot code region selector using a sample-based methodology. This serves to further bias the profiling toward frequently occurring instructions, and increases the likelihood that UMI overhead is amortized well. Sampling also provides a natural mechanism to adapt the introspection according to the various phases of the application lifetime. There are two sampling strategies. The first uses a regular sampling period, and the second is event driven. In either case, the region selector maintains a counter for each hot code region. With each sample, the program counter is inspected to determine its parent code trace, and the counter for that trace is incremented. A code region is selected for instrumentation when its counter saturates at the *frequency threshold*. The counter is then reset in anticipation of future sampling periods.

Instrumentor. The second component is the instrumentor. It operates on a selected hot region to insert new in-

structions that instrument and profile the code. The application alternates between instrumented and native code regions. When the instrumented code is run, it generates short profiles that record instruction addresses and the memory locations they reference. The instrumentor is commissioned with filtering the instructions in a code region such that only informative operations are instrumented. The instrumentor also determines the frequency with which to trigger the profile analyzer.

Profile Analyzer. The profile analyzer or mini-simulator is the third and final component in a system implementing UMI. It analyzes the recorded memory reference profiles to provide various forms of information relevant to an online optimizer. It is customizable and in this paper we present an example use of the analyzer as a fast cache simulator. It can perform simple hit and miss accounting as a hardware counter does. It may also simulate the hit and miss behavior for individual instructions to identity those that are more likely to miss. Such information is useful for optimizations that dynamically perform data prefetching.

3. Prototype System

We extended DynamoRIO to perform UMI. DynamoRIO is a dynamic binary rewriting framework for runtime instrumentation and optimization [5]. The prototype performs bursty profiling on running applications to collect short memory reference sequences for frequently executed code regions, and then uses a fast cache simulator to collect cache statistics dynamically. While our prototype was implemented in DynamoRIO, UMI can be realized in other similar systems or Java virtual machines.

DynamoRIO is a general-purpose runtime code manipulation system designed for transparency and efficiency. It can run large real world applications on off-the-shelf IA-32 hardware. DynamoRIO executes the input application by copying the user code, one basic block at a time, into a code cache before executing the code there. All runtime control flow is directed through the cache to provide a programmable mechanism for instrumentation, profiling, and optimizations. DynamoRIO reduces its overhead by directly linking blocks that are joined with direct branches, and using a fast lookup to transition between blocks that are linked with indirect branches. The system performs other optimizations that remove unconditional branches, and stitches together frequently executed sequences of blocks into traces, also called code fragments. The traces are kept in the trace cache with fast lookups for indirect branches.

DynamoRIO initially executes all user code from the basic block cache, until some set of blocks is considered hot. At that point, the blocks are inlined into a single-entry,

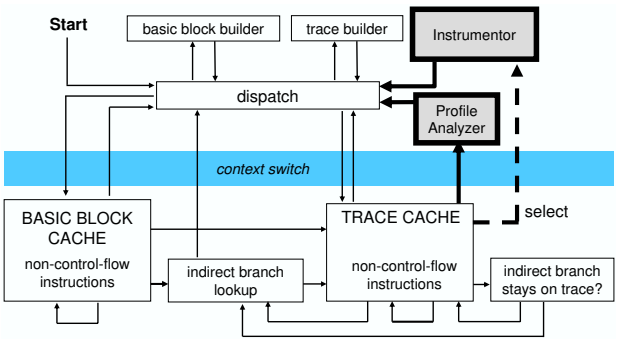

Figure 1. DynamoRIO and extensions for UMI.

multiple-exits trace, and placed in the trace cache via the trace builder.

The trace builder implicitly serves as the UMI region selector. It is reinforced by our sample-based selection. We use the program counter sampling utility in DynamoRIO to implement a time-based sampling strategy with a sampling period of 10 milliseconds, and a default frequency threshold of 64. We added two new components to DynamoRIO to implement the instrumentor and profile analyzer. They are highlighted in Figure 1. The figure also shows the other main components of DynamoRIO.

The instrumentor performs the following tasks for every selected code trace T:

1. It creates a clone T_c of the trace. The clone allows us to quickly turn the profiling on and off.

2. The instrumentor then scans T and filters out certain memory operations. The filtering serves to reduce the profiling overhead. The selected memory operations are instrumented for profiling.

3. A prolog is added to T in order to update various bookkeeping counters, and to determine when to trigger the profile analyzer.

The profile analyzer is a fast cache simulator. When triggered, it performs a context switch to save the application state, and then simulates the cache behavior using the recorded memory reference profiles as input. At this stage, we can obtain detailed memory behavior information at instruction granularity.

After simulation, control is relinquished to DynamoRIO, the instrumented code fragment (T) is replaced with its clone (T_c), and the application continues to execute without profiling. The context switch from the analyzer back to DynamoRIO provides a natural boundary to replace a trace with a newly optimized one, i.e., before replacing T with T_c, one can perform optimizations on T_c based on the mini-simulation results.

Table 3. Profiling statistics.

Benchmark	Static Loads	Static Stores	Profiled Operations	% Profiled	Profiles Collected	Analyzer Invocations
168.wupwise	6416	5148	1739	15.04%	285	36
171.swim	6285	4246	2688	25.52%	279	38
172.mgrid	5651	3615	2691	29.04%	318	27
173.applu	12277	6753	5578	29.31%	379	62
177.mesa	7163	5411	2050	16.30%	272	34
178.galgel	27306	18402	13951	30.52%	1226	331
179.art	3601	2254	1178	20.12%	188	73
183.equake	5571	3270	1950	22.06%	293	47
187.facerec	10166	6798	3586	21.14%	581	67
188.ammp	7027	4198	3084	27.47%	388	82
189.lucas	8179	4016	1963	16.10%	158	41
191.fma3d	16109	16506	4043	12.40%	756	119
200.sixtrack	22033	28204	9349	18.61%	1358	110
301.apsi	16303	11545	8531	30.63%	1027	94
164.gzip	3607	2745	931	14.66%	264	42
175.vpr	10937	8501	2359	12.14%	525	89
176.gcc	84642	69350	35079	22.78%	11188	254
181.mcf	3785	2377	1554	25.22%	237	60
186.crafty	23669	16237	6541	16.39%	1468	88
197.parser	18399	13916	10081	31.20%	3337	197
252.eon	20026	30287	5934	11.79%	579	56
253.perlbmk	34748	27951	12149	19.38%	3513	98
254.gap	26032	20489	11256	24.20%	2560	292
255.vortex	38264	56499	9120	9.62%	2307	83
256.bzip2	4956	3490	1619	19.17%	378	65
300.twolf	20059	12544	8289	25.42%	1498	220
em3d	1435	812	410	18.25%	69	22
health	2008	1270	322	9.82%	75	19
mst	1327	828	140	6.50%	29	10
treeadd	1220	713	224	11.59%	41	10
tsp	1832	1092	374	12.79%	58	12
ft	1871	1156	489	16.15%	87	18

4. Instrumentor

The instrumentor carefully manages the instrumentation overhead so that the introspection remains practical. First, we describe a filtering step designed to reduce the number of memory operations to profile, and then we describe some important implementation details.

4.1. Operation Filtering

Some architectures such as the Intel x86 platform allow most instructions to directly access memory. As a result, profiling all instructions that access memory is prohibitively expensive. The instrumentor uses two simple heuristics to prune the set of memory operations that are profiled.

The first is straightforward: only frequently executed code is instrumented. This is easily achieved by instrumenting only hot code regions. In DynamoRIO, these are the instruction traces that are formed from smaller basic blocks.

The second heuristic excludes from instrumentation any instructions that reference the stack or static addresses. The underlying assumption is that such references typically exhibit good locality. In x86 architectures, stack references use the `esp` or `ebp` registers. Hence, any memory accessing instruction whose operands are either a static address (e.g., a label with a literal offset), `esp` or `ebp` is ignored.

These simple heuristics reduce the set of candidate instructions for instrumentation by nearly 80%, as shown in Table 3. Each row shows the total number of static instructions that perform loads or stores, and the number of instructions selected for profiling, averaging 19.42%. The resulting reduction in profiled operations significantly lowers the profiling and analysis overhead. The last two columns of Table 3 show the number of collected profiles (i.e., memory reference sequences) and the total number of analyzer invocations. The results in Table 3 are in the absence of sample-based reinforcement, and hence provide an empirical upper bound on the instrumentation overhead.

4.2. Instrumentation Details

There are two parts to the instrumentation code. The first is a prolog that conditionally triggers the analyzer (mini-simulator). The second consists of profiling instructions that create a record of accessed memory locations. Memory references are recorded in a two-level data structure. A unique *address profile* is associated with each code trace. The address profile is two-dimensional, with each row corresponding to a single execution of the trace. The columns are organized such that each records the sequence of memory addresses referenced by an individual operation in the code fragment, spanning multiple executions of the trace. The two-dimensional representation simultaneously captures trace and instruction level profiling information, and is useful for various optimizations. On every trace entry, a record is allocated in a *trace profile* to point to a new row in the address profile. The sequence of addresses referenced during that execution of the trace is recorded in the corresponding row of the address profile.

The prolog code initiates the analyzer when either the trace profile or the address profile is full. The prolog requires two conditional jumps. We reduce this overhead to a single conditional jump by observing that in the common case, the cap on the size of address profile triggers the analyzer. The trace profile is recorded in a buffer that is guarded by a protected memory page. When the buffer is full, the analyzer is automatically triggered as writes are not allowed to the protected page. This allows the prolog code to only check for available slots in the address profile.

The length of the trace profile is 8,192 entries by default. The address profile has a default limit of 256 operations and 256 entries per operation (i.e., 256 executions of the code trace). In the worst case, the space overhead is 32 KB of storage for the trace profile, and 256 KB for each address profile. Another 64 KB are needed for the analyzer, leading to a total space overhead of 2 GB if all 8,192 distinct profiles are live simultaneously. In our experiments, we found that an average of 3 trace profile entries are used at any given time, with an average of 5 instrumented instructions per code fragment. Thus, our scheme adds between 80 KB-128 KB of memory overhead, including the 64 KB required for profile analysis.

A naive injection of instrumentation code to record the memory reference information is potentially too expensive. A memory reference is the tuple (`pc`, `address`), and to

record this information requires nine operations in a straight forward approach. We implemented a number of optimizations to reduce the overhead to between four to six operations. These details are omitted here.

5. Profile Analyzer

The analyzer for this paper is a fast cache simulator. It is configured to match the number of sets, the line size, and the associativity of the secondary cache on the host machine. The simulator implements an LRU replacement policy although other schemes are possible. The mini-simulation results were observed to be far more dependent on the length of the address profiles, than on the actual configuration of the simulated cache. We observed statistically insignificant variations in our results when simulating caches that are much smaller than that of the host machine. This is not surprising since mini-simulations span much shorter address spaces compared to longer or full simulations.

The simulator is similar to the one used in Cachegrind [20]. It tracks the miss ratios for individual operations, and also maintains coarser level performance details. During simulation, each reference is mapped to its corresponding set. The tag is compared to all tags in the set. If there is a match, the recorded time of the matching line is updated. Otherwise, an empty line, or the oldest line, is selected to store the current tag. We use a counter to simulate time.

Since not all memory references are profiled, the simulated results are approximations of the application behavior. Furthermore, because only a small fraction of the memory references is simulated, the simulator must be tuned to account for the high number of compulsory misses, and the low number of conflict and capacity misses that would otherwise arise. Thus, in order to improve the simulated results, cache miss accounting only starts after the first few accesses in the address profile, typically two executions of the trace. This has the effect of warming up the cache, and is akin to functional warming in offline cache simulations that use fast forwarding. We also use a single logical cache to analyze all of the recorded address profiles. In other words, the state of the cache is carried over from the analysis of one profile to the next. We periodically flush the cache state to avoid long term contamination. In our experiments, the flush occurs whenever the analyzer is triggered and more than 1M processor cycles (obtained using `rdtsc`) have elapsed since it last ran.

6. Experimental Methodology and Analysis

We ran the experiments on a 3.06 GHz Intel Pentium 4 with 1 GB of RAM. The operating system is Linux Fedora Core 1. The memory hierarchy consists of an 8-way associative L1 instruction cache, with a capacity to hold 12 K micro instructions. The L1 data cache is an 8 KB 4-way associative cache with 64-byte cache lines. The L2 cache is a 512 KB, 8-way associative unified cache with 64-byte cache lines. The benchmarks are x86 binaries compiled with gcc version 3.3 using the `-O3` flag. We used all of the benchmarks from SPEC CPU2000 and their reference input workloads.

We also selected `em3d`, `health`, `mst`, `treeadd`, and `tsp` from Olden [21], and `ft` from the Ptrdist [3] benchmark suite. Olden and Ptrdist are commonly used in the literature when evaluating dynamic memory optimizations. The other benchmarks from these suites have too short a running time (less than 5 seconds) for meaningful measurements, and were therefore omitted.

We repeated many of the experiments on an older AMD Athlon MP 1400+ (1.2 GHz K7 architecture). It has a 64 KB, 2-way associative L1 data cache, a 64 KB L1 instruction cache, and a 256 KB L2 unified cache that is 16-way associative. Both cache levels have 64-byte cache lines.

6.1. UMI Runtime Overhead

Figure 2 shows the overhead of our system compared to native execution. The 14 SPEC CFP2000 benchmarks are shown first, followed by the 12 SPEC CINT2000 benchmarks. The Olden and Ptrdist codes appear last. In native execution, the application is compiled with `-O3` and executed without DynamoRIO. The first bar shows the relative running time of the application running with DynamoRIO. A value greater than one implies performance degradation, and a value less than one implies a speedup. The second bar shows the relative performance for UMI (i.e., application running time with DynamoRIO augmented with our profiling and analysis) when no sampling is used. The third bar accounts for the sampling overhead as well.

In general, sampling reduces the running time for applications that spend the bulk of their execution iterating through a small set of traces. This is the case for `179.art`, `181.mcf`, and `256.bzip2` for example. Sampling also leads to lower runtime overhead because it effectively delays the instrumentation of certain code fragments until it appears more profitable to do so, and it may even prevent the process altogether. This effect is most prominent in `176.gcc` which spends less than 70% of its execution running from the trace cache. As a result, the overheads from trace formation, instrumentation, and profile analysis are not amortized well in `176.gcc`, leading to a significant slowdown in the absence of sampling. For comparison, many of the other benchmarks execute from the code cache more than 95% of the time.

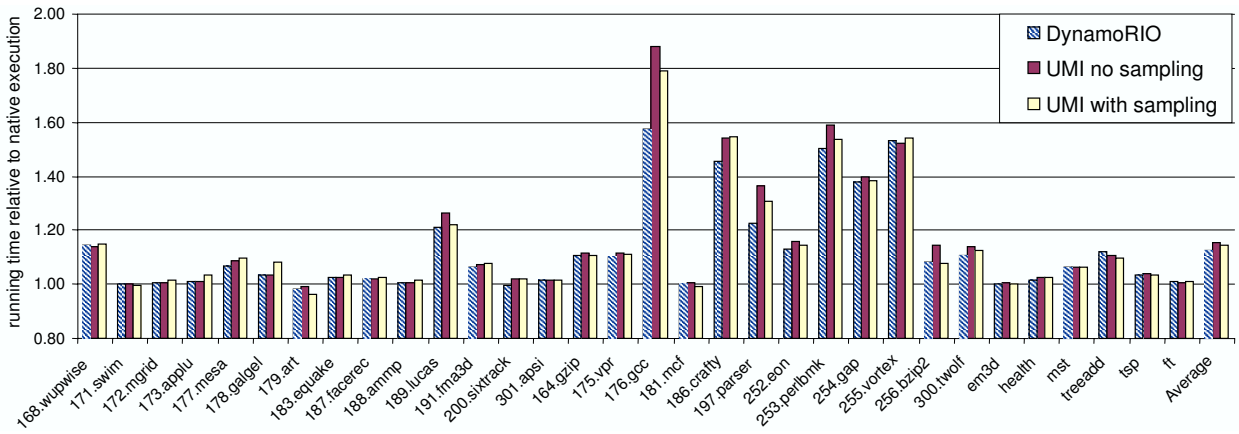

Figure 2. Runtime overhead on Pentium 4 with hardware prefetching enabled.

It is evident from the data that DynamoRIO has little overhead in general, with an average slowdown that measures less than 13%. Some benchmarks actually run faster with DynamoRIO because they benefit from code placement and trace optimizations performed by DynamoRIO. The system performance suffers most in the case of the CINT2000 benchmarks because of their control intensive nature. The overhead incurred for UMI (i.e., region selector, instrumentor, and analyzer) averages to a 14% slowdown overall. This slowdown is only 1% more than DynamoRIO alone. These results suggest that online mini-simulations and detailed introspection will become increasingly practical since the performance of binary instrumentation tools like DynamoRIO have steadily improved over the years.

6.2. Correlation to Hardware Counters

We evaluate the mini-simulations by comparing the simulated cache miss rates to the actual miss rates measured using the Pentium 4 and AMD K7 hardware performance counters. The miss rates reported by the mini-simulations will differ from the actual rates reported by the hardware counters, but a strong correlation between the two is important because it implies that relative observations derived using UMI accurately reflect actual phenomena.

We divide the benchmarks into three groups: CFP2000, CINT2000 and Olden (which includes `ft` for convenience). The group *coefficient of correlation* $C(s,h)$ is calculated using the equation

$$C(s,h) = \frac{\sum_i (s_i - \bar{s})(h_i - \bar{h})}{\sqrt{\sum_i (s_i - \bar{s})^2 (h_i - \bar{h})^2}}$$

where s_i equals the simulated cache miss ratio for each benchmark i in the group, h_i equals the cache miss ratio obtained using the hardware performance counters, and $\bar{s}$ and $\bar{h}$ equal the average miss ratios calculated respectively from all s_i and h_i in the group. The miss ratios are obtained by dividing the number of L2 miss counts by the number of L2 references, for both loads and stores.

Table 4. Coefficients of correlation.

Platform	Cachegrind			UMI			
	CFP2000	CINT2000	Olden	CFP2000	CINT2000	Olden	All
Pentium 4 without HW prefetching	0.997	1.000	0.992	0.929	0.782	0.920	0.883
Pentium 4 with HW prefetching	0.992	0.999	0.957	0.896	0.796	0.861	0.852
AMD K7	—	—	—	0.825	0.689	0.909	0.828

The results are reported in Table 4. The Pentium 4 includes two hardware prefetchers, and so we measured the miss ratios under two scenarios. In the first, we disabled both prefetchers. The correlation between UMI and the hardware counters is 0.883 when all the benchmarks are grouped into a single category. The correlation is highest for the CFP2000 and Olden groups, and lowest for the CINT2000 group. The former are loop intensive applications, and we expect simulations from short memory profiles to extrapolate well to the application as a whole. The CINT2000 benchmarks are more control intensive with irregular access patterns that require longer simulations to improve the correlation.

For comparison, we also present the correlation between the hardware counters and Cachegrind, a cache profiler and simulator distributed with Valgrind [20]. Cachegrind simulates the memory hierarchy using a complete trace. It adds a runtime overhead between 20×-100×. With the Pentium 4 prefetcher disabled, Cachegrind achieves a near perfect overall correlation of 0.994. The correlation for the CFP2000 and Olden groups is lower than the CINT2000 group. This is likely caused by a mismatch in the way floating-point values that cross multiple cache lines are handled in the simulator versus hardware.

In the second scenario, we measure the miss ratios when the Pentium 4 prefetchers are enabled. In this case, the UMI and Cachegrind miss ratios are unchanged since they ignore

Table 5. SPEC2006 coefficients of correlation.

	CFP2006	CINT2006	SPEC2006
Pentium 4 with HW prefetching	0.94	0.79	0.85

any prefetching side effects. The hardware measured miss ratios however are relatively lower, although the prefetching impact on the number of misses varies with each application. In general, the overall Cachegrind correlation decreases to 0.952, and the UMI correlation reduces to 0.852 since neither simulates prefetching side effects, which typically reduce miss rates.

In addition to the Pentium 4, we also measured the correlation between UMI and the AMD K7 hardware counters. The overall correlation is 0.828, which is lower than the correlation measured on the Pentium 4. The UMI mini-simulator does not simulate an instruction cache, and the impact of instruction caching may be magnified on the AMD K7 architecture since its unified L2 cache is half the size of the unified cache on the Pentium 4. The Cachegrind simulations on the Pentium 4 required a week to complete, and were not repeated for the slower AMD K7.

6.3. Other Benchmarks and Applications

We used our prototype to profile many more benchmarks than the ones reported here. Our extended benchmark collection includes the SPEC CPU2006 suite and several commonly used Linux applications such as Adobe Acrobat, Apache, MEncoder, and MySQL. We found the HW measured miss ratios to be very low for the Linux applications. Of the CPU2006 benchmarks, we evaluated the following subset which does not overlap with CPU2000: `433.milc`, `435.gromacs`, `444.namd`, `450.soplex`, `453.povray`, `470.lbm`, and `482.sphinx3` from CFP2006, and `445.gobmk`, `456.hmmer`, `458.sjeng`, `462.libquantum`, `464.h264ref`, `471.omnetpp`, `473.astar`, and `483.xalancbmk` from CINT2006. The correlation for these benchmarks is summarized in Table 5.

7. UMI for Delinquent Load Identification

In addition to the coarse-grained mini-simulation results, we can use UMI to identify high miss ratio load instructions in a given program. Such profiling information can greatly improve the performance of data prefetching strategies as it helps to focus the optimizations on memory references that are likely to miss in the cache. For example in our own work, we were able to implement a simple software prefetcher that achieved an average speedup of 11% on two different architectures, with a best case performance gain of 64%. It is worthwhile to note that information of such fine granularity is hitherto only available through full cache simulation or with specialized hardware.

We used Cachegrind as a baseline for evaluating the quality of our online analysis. We modified Cachegrind to report the number of cache misses for individual memory references rather than for each line of code in the source program. We define the set of delinquent load instructions, $\mathcal{C}$, as the minimal set of instructions that account for at least x percent of the total number of load misses. We report results for $x = 90\%$. We can calculate C by sorting the instructions in descending order of their total number of L2 load misses, as reported by Cachegrind. Then, starting with the first instruction, we add instructions to the set until the number of misses in the set is at least 90% of the total number of misses reported for the entire application.

An offline technique that identifies delinquent loads in this manner uses global information about all memory references in the application. In contrast, a runtime system with memory introspection needs immediate profiling results that it can readily act on for optimizations. Therefore, it must predict delinquent loads with only local knowledge. Let $\mathcal{P}$ be the set of memory load instructions predicted by UMI as delinquent. In our prototype, at the end of a mini-simulation, the profile analyzer labels memory load instructions with a miss ratio higher than a *delinquency threshold* α as delinquent loads.

7.1. Summary of Results

Table 6 reports the quality of the results. The size of $\mathcal{C}$ is $|\mathcal{C}|$, and the size of $\mathcal{P}$ is $|\mathcal{P}|$. The set $\mathcal{P} \cap \mathcal{C}$ represents loads found to be delinquent by exhaustive simulation and online introspection. The *miss coverage* of $\mathcal{P}$ represents the fraction of the total number of misses in the application that members of the set $\mathcal{P}$ account for. Similarly, the miss coverage of $\mathcal{P} \cap \mathcal{C}$ is the fraction of the total number of misses in the application that the members of the set $\mathcal{P} \cap \mathcal{C}$ account for. We use the *recall* and *false positive* measures (last two columns) to quantify the accuracy of the predictions. The recall is the ratio of the number of correctly identified delinquent loads ($|\mathcal{P} \cap \mathcal{C}|$) to the total number of delinquent loads ($|\mathcal{C}|$). The false positive measure is the ratio of the number of incorrect predictions ($|\mathcal{P} - \mathcal{C}|$) to the total number of predictions $|\mathcal{P}|$. Ideally, the recall is 100% with a 0% false positive ratio.

The miss coverage is 86.15% for benchmarks with a L2 miss ratio greater than 1%, and 40.13% for all others. It is greater than 65% for the benchmarks overall, with some notable exceptions: `164.gzip`, `176.gcc`, and `252.eon`. These benchmarks have very low miss ratios as indicated in the second column. In `164.gzip`, one instruction causes more than 90% of the cache misses. It performs a byte-by-byte memory copy and has a 2% miss ratio as reported by

Table 6. Quality of delinquent load prediction.

Benchmark	L2 Cache Miss Ratio (Cachegrind)	$\|\mathcal{P}\|$	Ratio of $\|\mathcal{P}\|$ to total # of loads	$\mathcal{P}$ Miss Coverage	90% delinquency $\|\mathcal{C}\|$	$\|\mathcal{P} \cap \mathcal{C}\|$	$\mathcal{P} \cap \mathcal{C}$ Miss Coverage	$\frac{\|\mathcal{P} \cap \mathcal{C}\|}{\|\mathcal{C}\|}$	$\frac{\|\mathcal{P} - \mathcal{C}\|}{\|\mathcal{P}\|}$
168.wupwise	0.82%	20	0.31%	74.94%	11	7	70.33%	63.64%	65.00%
171.swim	4.71%	64	1.02%	99.80%	32	32	90.23%	100.00%	50.00%
172.mgrid	1.30%	48	0.85%	95.37%	18	18	90.59%	100.00%	62.50%
173.applu	1.26%	137	1.12%	76.17%	75	50	73.49%	66.67%	63.50%
177.mesa	0.02%	20	0.28%	22.86%	10	2	22.85%	20.00%	90.00%
178.galgel	1.93%	78	0.29%	93.19%	10	8	87.89%	80.00%	89.74%
179.art	27.13%	81	2.25%	94.26%	43	41	88.79%	95.35%	49.38%
183.equake	3.83%	56	1.01%	68.00%	34	26	63.83%	76.47%	53.57%
187.facerec	0.83%	38	0.37%	87.92%	12	8	81.12%	66.67%	78.95%
188.ammp	1.48%	136	1.94%	88.33%	101	80	84.61%	79.21%	41.18%
189.lucas	1.12%	230	2.81%	94.82%	70	66	87.84%	94.29%	71.30%
191.fma3d	1.73%	117	0.73%	84.54%	45	42	78.00%	93.33%	64.10%
200.sixtrack	0.12%	6	0.03%	19.72%	37	2	17.95%	5.41%	66.67%
301.apsi	1.07%	142	0.87%	90.16%	69	59	85.00%	85.51%	58.45%
164.gzip	0.06%	4	0.11%	0.00%	1	0	0.00%	0.00%	100.00%
175.vpr	0.92%	45	0.41%	89.70%	26	23	87.34%	88.46%	48.89%
176.gcc	0.48%	1	0.00%	0.00%	293	0	0.00%	0.00%	100.00%
181.mcf	20.10%	54	1.43%	97.67%	15	15	90.24%	100.00%	72.22%
186.crafty	0.03%	2	0.01%	31.28%	25	2	31.28%	8.00%	0.00%
197.parser	0.50%	72	0.39%	60.90%	117	34	60.52%	29.06%	52.78%
252.eon	0.00%	7	0.03%	0.00%	47	0	0.00%	0.00%	100.00%
253.perlbmk	0.15%	5	0.01%	33.23%	81	5	33.23%	6.17%	0.00%
254.gap	0.33%	20	0.08%	59.88%	10	1	59.87%	10.00%	95.00%
255.vortex	0.19%	2	0.07%	20.48%	21	2	20.48%	9.52%	0.00%
256.bzip2	0.89%	19	0.38%	76.88%	27	14	76.88%	51.85%	26.32%
300.twolf	1.78%	117	0.58%	98.07%	38	38	90.29%	100.00%	67.52%
em3d	24.49%	6	0.42%	99.75%	3	3	94.76%	100.00%	50.00%
health	12.44%	16	0.80%	86.92%	3	2	78.35%	66.67%	87.50%
mst	7.53%	7	0.53%	99.41%	5	5	94.75%	100.00%	28.57%
treeadd	1.90%	3	0.25%	99.98%	2	2	99.97%	100.00%	33.33%
tsp	1.12%	6	0.33%	72.32%	7	3	77.32%	42.86%	50.00%
ft	49.63%	1	0.05%	99.84%	1	1	99.84%	100.00%	0.00%
Average (miss ratio $<$1.00%)		19	0.17%	41.27%	51	7	40.13%	25.63%	58.83%
Average (miss ratio $\geq$1.00%)		72	0.96%	91.03%	32	27	86.15%	87.80%	55.16%
Average (all benchmarks)		49	0.62%	69.26%	40	18	66.02%	60.60%	56.76%

Cachegrind. In `176.gcc`, the cache misses are distributed across 293 memory references, each having a very low miss ratio. Lastly, `252.eon` is computationally intensive and exhibits very good reference locality. Other benchmarks with low coverage (e.g., `186.crafty`, `253.perlbmk`, `treeadd` and `tsp`) exhibit similar characteristics.

The recall and false positives are dependent on the delinquency threshold. A high delinquency threshold means relatively few loads are labeled as delinquent. This reduces the false positives, but may also reduce the recall. If it is set too low, then it leads to many false positives but also improves the recall. We found that dynamically tuning the delinquency threshold can significantly reduce the number of false positives. This is accomplished by assigning each code trace a unique delinquency threshold, initially equal at 0.90. This threshold is reduced by 0.10 following every profile analyzer invocation that the trace is responsible for, down to a minimum threshold of 0.10. This approach, compared to a singular global delinquency threshold, significantly reduces the false positives from 82.61% to 56.76% overall, and marginally increases the recall from 86.81% to 87.80% for benchmarks with a L2 miss ratio greater than 1%. A straightforward comparison to Moshovos et al.'s work [19] shows that we report 18$\times$ fewer false positives. Most other papers report only performance speedups and prevent a direct comparison. We believe that for delinquent load identification, UMI delivers the best results so far relative to all published data we found.

7.2. Sensitivity Analysis

There are many instrumentation and profile analysis parameters that can impact the mini-simulation results. For example, we use a frequency threshold (Section 2) of 64 for the sample-based reinforcement, but a lower threshold can potentially increase the recall since it admits a greater number of code traces for instrumentation and profiling. A higher threshold has the opposite effect of reducing the recall, while also reducing the false positive ratio. The default frequency threshold was observed to work well for the collection of benchmarks we used. Future work may explore adaptively tuning the threshold according to the application and trace characteristics.

We performed two case studies to provide some insight as to the relationship between the frequency threshold, recall, and false positive ratio. We used `181.mcf` and `197.parser` as representative benchmarks. The former is memory intensive with a 20% L2 miss ratio, and the latter has a miss ratio of 0.50%. We increased the frequency threshold by powers of two, from a minimum of 1 to a maximum of 1024, and observed the following trends which affirm intuition. As the threshold increased, the recall rate generally decreased. For `181.mcf`, the recall was constant for thresholds of 1-256, and then dropped to 87%, and then again to 73% as the threshold reached 512 and 1024, respectively.

For `197.parser`, the loss in recall was more dramatic, falling exponentially with each threshold increase from a

maximum of 100% for a threshold of 1, down to 0% for a threshold of 512 and 1024. The contrast between the two benchmarks is largely due to their execution patterns. In `181.mcf`, execution spans a few memory intensive and long running loops, and is mostly insensitive to variations in the sampling frequency. In contrast `197.parser` has a much more dynamic control flow, and many loops run for only a few iterations. As a result, recall is best when the frequency threshold is at its lowest value. The low threshold admits virtually all code traces for instrumentation and profiling, but has the undesirable side effect of an 88% false positive ratio. The runtime overhead for both benchmarks was constant for a frequency threshold greater than 32.

In general, the recall is inversely related to the frequency threshold. However the relationship between the recall and the length of the simulated address profile is less obvious. In `181.mcf`, varying the length of the address profile has no effect on the recall, and only marginally improves the false positive rate. We varied the length of the address profile from a minimum of 64 trace executions to a maximum of 32K executions (in powers of two). The actual length of the recorded address sequence is proportional to the number of trace executions and the number of memory references in each trace iteration. Hence increasing the size of the address profile increases the length of the recorded memory reference sequence. In `197.parser`, the same experiment reduced the recall significantly from 34% at the minimum address profile size, to less than 17% for address profiles of 4K iterations or more. In `197.parser`, as in other benchmarks with a low overall miss ratio, the longer simulations have the effect of rapidly lowering the miss ratio of individual instructions, and at a faster rate than we tune the delinquency threshold. As a result fewer instructions are identified as delinquent, thus reducing the recall. The false positive ratio however is affected more favorably, with a ratio of 36% at 8K iterations, and 23% at 16K iterations. The overall runtime overhead is largely unchanged for the two benchmarks, ranging up to 3% for `181.mcf`, and 27-30% for `197.parser`.

In order to systematically reduce the false positive ratio, it may prove necessary to eventually weigh in the collective effects of multiple memory operations, rather than label each operation independent of any other, as we do in the current runtime system.

8. Example Runtime Optimization Using UMI

We illustrate an example use scenario for UMI by implementing a simple stride prefetching optimization in software. The optimization issues L2 prefetch requests for loads labeled as delinquent by the introspection phase. We modified the profile analyzer to also calculate the stride distance between successive memory references for individual loads. The profiling information is used online to modify the instruction code trace to inject prefetch requests. Of the 32 benchmarks in our suite, we discovered prefetching opportunities for 11 of them. The results are shown in Figures 3 and 4 for the Pentium 4 and AMD K7 processors respectively. The figures report the normalized running time compared to native execution. The first bar shows the running time when the introspection is carried out and no optimizations performed. The runtime is normalized to native execution (with hardware prefetching turned off), hence lower values indicate a greater speedup. The second bar indicates the normalized running time with online introspection and software prefetching. The results show an 11% average performance improvement on both processors.

We investigate the efficacy of the prefetching further by comparing against the hardware prefetching strategies available on the Pentium 4. It implements two prefetching algorithms for its L2 cache. They are *adjacent cache line* prefetching and *stride* prefetching [14]. The latter can track up to 8 independent prefetch streams. The prefetchers can be disabled independently but for our experiments, adjacent line prefetching is always on. The AMD K7 does not have any documented hardware prefetching mechanisms.

Figure 5 shows the running time for the same benchmarks when the Pentium 4 hardware prefetchers are enabled. The performance is normalized to native execution and no prefetching. The first bar shows the performance of UMI with our software prefetching scheme. The second bar shows the performance of the hardware prefetcher, and the third bar combines UMI with software and hardware prefetching. We note from the data that while software prefetching is effective on its own, the combination with the hardware prefetcher does not lead to cumulative gains for many of the benchmarks. It is plausible that the software and hardware prefetchers are occasionally redundant. In other words, the software prefetcher requests the same references as the hardware prefetcher.

We examine if this is the case using the hardware counters to measure the number of L2 misses on the Pentium 4. This provides a measure of prefetching coverage. The results are reported in Figure 6. The data shows the number of misses normalized to native execution, with lower ratios indicating a greater reduction in misses. We observe that there is a cumulative effect in reducing the number of cache misses. In other words, the combination of software and hardware prefetching leads to a greater reduction in the number of L2 cache misses. This is observed for most of the benchmarks, with an average of 62% reduction in misses compared to 71% and 69% for software and hardware prefetching alone. Since the results show the combination of prefetching schemes lead to fewer misses, it is likely that the combination also increases contention for resources, and affects timeliness.

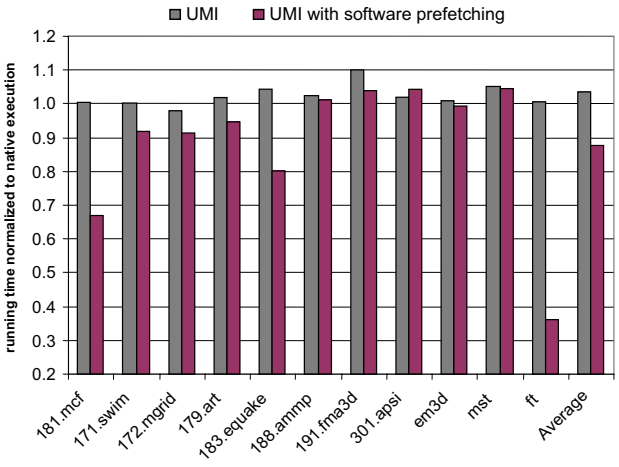

Figure 3. Running time on Pentium 4 with hardware prefetching disabled.

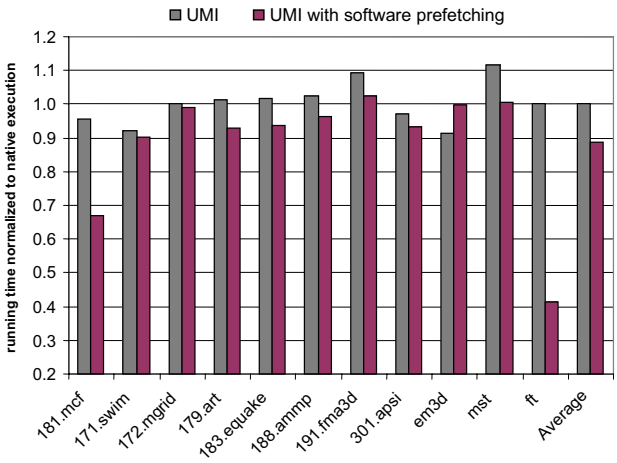

Figure 4. Running time on AMD K7.

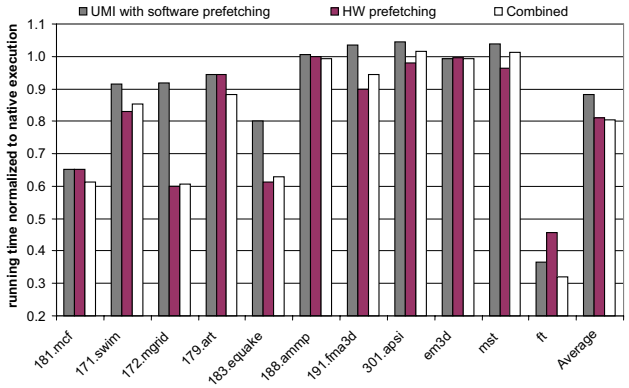

Figure 5. Running time on Pentium 4 with hardware prefetching enabled.

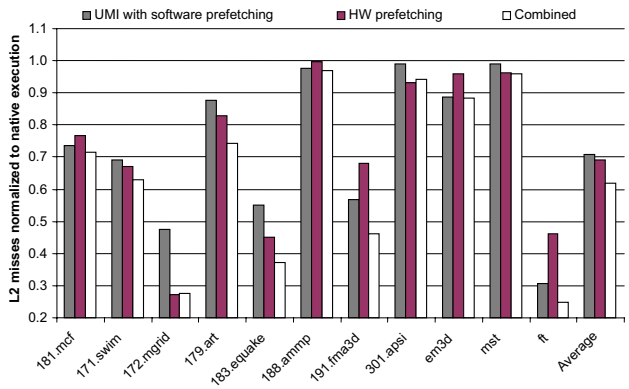

Figure 6. L2 misses on Pentium 4.

We attempted the same coverage experiment on the AMD K7, but determined that we cannot distinguish between refills due to L2 misses and those due to prefetching. As a result we observed no significant differences in the refill counts when software prefetching was enabled.

We probed further into the performance of `ft` and found that it was very sensitive to the choice of prefetch distances. It turns out that UMI was able to pick a prefetch distance that is closer to the optimal prefetching distance compared to the hardware prefetcher. This highlights an important advantage of UMI, namely that a more detailed analysis of the access patterns is possible in software than is usually feasible in hardware.

The goal of this paper is not to champion a better software prefetching algorithm. We present these results only as a demonstration of the potency of the information afforded by UMI. We believe other performance enhancing mechanisms can also benefit from UMI.

9. Background and Related Work

Cache Modeling and Evaluation. There are three approaches for evaluating or modeling the performance of memory systems: hardware monitoring, software simulation, and analytical modeling. Hardware monitoring has the advantages of being accurate with relatively low runtime overhead. There are a number of proposals for architectures to monitor cache behavior [8, 25]. However, hardware-based approaches lack generality because they require non-trivial architectural changes. Modern processors support a restricted set of mechanisms that sample and count certain hardware events. This form of sampling lacks contextual information, and is generally only suitable for computing statistical summaries, rather than fine-grained analysis of individual memory access operations. Some researchers have successfully used the performance monitoring units to collect performance profiles and identify delinquent loads

on specific processors [16, 17]. Such schemes however are generally not portable across platforms.

Software simulators such as SimpleScalar [6], Cachegrind [20] and Dinero [12] are able to simulate detailed cache behaviors. However, the associated overhead is often too significant to evaluate realistic workloads. Often it takes hours to complete the entire process, even for medium-sized workloads. So it is hard to scale this approach to large real-world applications. We have already successfully used our UMI prototype to profile several commonly used Linux desktop and server applications.

There is a large body of work on analytical cache models (see [1] and [15] and for examples). These models are built on probabilistic assumptions that may not hold in practice, and often require entire address traces to be stored for analysis. The models are typically used to reason about general trends, and do not provide fine-grained details.

Delinquent Load Identification. Nearly all prefetching techniques necessitate some form of delinquent load identification. Typically this is done using profiling and complete cache simulations, both of which are very time and resource consuming, and can only be used offline as part of a profile-guided optimization framework.

A common strategy to reduce the overhead relies on periodic sampling of the memory references [2]. An implementation in Jalapeño achieved an average overhead of 3%. Hirzel and Chilimbi [13] implemented the same scheme for x86 binaries and found the average overhead to be between 6-35%. They managed to reduce the overhead to 3-18% by coalescing dispatchers, but their scheme requires some static code analysis. Neither approach explored the idea of recording traces and using online mini-simulations.

Others have proposed static techniques [24] to identify delinquent operations without simulation, while some schemes use profiling to improve accuracy [22]. These strategies require suitable training data that are representative of real workloads. To reduce the overhead, many hardware based delinquent load identification and prefetch schemes were proposed [9, 7, 10, 19, 11], but they suffer from the need of specific hardware support.

In contrast to previous work, UMI is well suited for runtime optimizers and virtual machines. It does not require any static analysis of the source code, and can be readily applied to large programs running on off-the-shelf hardware, without any modifications to the application code.

10. Concluding Remarks

This paper contributes a lightweight and practical alternative to offline profiling with simulators, and performance tuning using hardware counters. We introduced Ubiquitous Memory Introspection (UMI) as a new methodology that provides online and application-specific profiling information necessary for runtime memory-centric optimizations.

UMI is based on the insight that bursty online profiling and mini-simulations of short memory reference traces can reasonably approximate the underlying memory system behavior. UMI permits the development of online memory optimizations that have the new capability of inspecting memory performance at its finest granularity (instructions and addresses). Runtime optimizers have the unique advantage of customizing optimization plans in a workload-specific manner, and can lessen the impact of offline performance tuning that may have used training workloads that do not accurately reflect actual use scenarios. UMI fills a gap between time consuming profiling using offline simulations and hardware counters designed for medium to large granularity performance monitoring.

Our implementation of UMI has a 14% overhead compared to native execution. This cost is only 1% greater than an existing state of the art binary instrumentation and optimization tool. We presented three applications of UMI that we can verify against actual systems. First, we showed that UMI can accurately model the cache performance on existing memory systems for 32 benchmarks, including the full suite of SPEC CPU2000 benchmarks. On a Pentium 4 and an AMD K7, we observed strong correlation between the mini-simulation miss rates, and the hardware measured miss rates.

Second, we presented an application of UMI at a much finer level. We showed how to use UMI to identify delinquent load instructions in a program. We validated our results against full cache simulations. We showed that we can accurately identify 88% of the delinquent loads for programs with relatively high miss rates, and 61% overall. Although the false positive ratio is 57%, we believe it is sufficiently low to make some optimizations practical. We continue to look for ways to reduce the number of false positives. We believe refinements to our methodology will significantly improve accuracy and utility.

Third, we used the results of the introspection to implement an example runtime optimization. We implemented a simple online software prefetcher. Its performance was competitive with a hardware prefetcher, achieving an 11% performance gain. In the best case, the software prefetcher discovered a prefetching opportunity that outperformed the Pentium 4 prefetcher. We believe there are many other memory optimizations that can use UMI, and this is an active area of research that we are pursuing. We also believe UMI presents new opportunities in the context of emerging multicore architectures where memory performance poses a serious challenge to performance scalability.

Acknowledgements

This research was sponsored in part by the Singapore-MIT Alliance, NUS Research Grant R-252-000-248-112, and DARPA through the Department of the Interior National Business Center under grant numbers NBCH104009, PCA-F29601-03-2-0065 and HPCA/PERCS-W0133890. We thank Martin Hirzel and the anonymous reviewers for their valuable comments on earlier drafts of this paper.

References

[1] A. Agarwal, J. Hennessy, and M. Horowitz. An analytical cache model. *ACM Trans. Comput. Syst.*, 7(2):184–215, 1989.

[2] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 168–179, New York, NY, USA, 2001. ACM Press.

[3] T. Austin. Pointer-intensive benchmark suite. http://www.cs.wisc.edu/~austin/ptr-dist.html.

[4] K. Barr. *Summarizing Multiprocessor Program Execution with Versatile, Microarchitecture-Independent Snapshots*. PhD thesis, Massachusetts Institute of Technology, September 2006.

[5] D. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, September 2004. http://www.cag.csail.mit.edu/rio/.

[6] D. Burger and T. Austin. The SimpleScalar Tool Set, Version 2.0. Technical report, University of Wisconsin-Madison Computer Science Department, 1997.

[7] J. D. Collins, S. Sair, B. Calder, and D. M. Tullsen. Pointer cache assisted prefetching. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 62–73, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[8] J. D. Collins and D. M. Tullsen. Runtime identification of cache conflict misses: The adaptive miss buffer. *ACM Trans. Comput. Syst.*, 19(4):413–439, 2001.

[9] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen. Dynamic speculative precomputation. In *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 306–317, Washington, DC, USA, 2001. IEEE Computer Society.

[10] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen. Speculative precomputation: long-range prefetching of delinquent loads. *SIGARCH Comput. Archit. News*, 29(2):14–25, 2001.

[11] R. Cooksey, S. Jourdan, and D. Grunwald. A stateless, content-directed data prefetching mechanism. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 279–290, New York, NY, USA, 2002. ACM Press.

[12] J. Edler and M. Hill. Dinero IV Trace-Driven Uniprocessor Cache Simulator. http://www.cs.wisc.edu/~markhill/DineroIV.

[13] M. Hirzel and T. M. Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *ACM Workshop on Feedback-Directed and Dynamic Optimization*, 2001.

[14] Intel Corporation. *IA-32 Intel Architecture Optimization Reference Manual*.

[15] S. Laha, J. H. Patel, and R. K. IYER. Accurate low-cost methods for performance evaluation of cachememory systems. *IEEE Transactions on Computers*, 37(11):1325–1336, Nov 1988.

[16] J. Lu, H. Chen, R. Fu, W.-C. Hsu, B. Othmer, P.-C. Yew, and D.-Y. Chen. The performance of runtime data cache prefetching in a dynamic optimization system. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 180, Washington, DC, USA, 2003. IEEE Computer Society.

[17] J. Lu, A. Das, W.-C. Hsu, K. Nguyen, and S. G. Abraham. Dynamic helper threaded prefetching on the sun ultrasparc cmp processor. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 93–104, Washington, DC, USA, 2005. IEEE Computer Society.

[18] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation*, pages 190–200, Chicago, IL, June 2005.

[19] A. Moshovos, D. N. Pnevmatikatos, and A. Baniasadi. Slice-processors: an implementation of operation-based prediction. In *ICS '01: Proceedings of the 15th international conference on Supercomputing*, pages 321–334, New York, NY, USA, 2001. ACM Press.

[20] N. Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, University of Cambridge, November 2004. http://valgrind.org/.

[21] OLDEN benchmark suite. http://www.cs.princeton.edu/~mcc/olden.html.

[22] V.-M. Panait, A. Sasturkar, and W.-F. Wong. Static identification of delinquent loads. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 303, Washington, DC, USA, 2004. IEEE Computer Society.

[23] PAPI: Performance application programmer interface. http://icl.cs.utk.edu/papi/.

[24] Y. K. Toshihiro Ozawa and S. Nishizaki. Cache miss heuristics and preloading techniques for general-purpose programs. In *Micro28: Proceedings of the 28th International Symposium on Microarchitecture*, pages 243 – 248, 1995.

[25] C. Zilles and G. Sohi. A programmable co-processor for profiling. In *HPCA '01: Proceedings of International Symposium on High Performance Computer Architecture*, 2001.