Appeared in Proc. Intl. Symp. on Microarchitecture (MICRO 08), November 2008, pp.258-269.

Reducing the Harmful Effects of Last-Level Cache Polluters with an OS-Level, Software-Only Pollute Buffer

Livio Soares David Tam Michael Stumm
Department of Electrical and Computer Engineering
University of Toronto
{*livio,tamda,stumm*}@*eecg.toronto.edu*

Abstract

It is well recognized that LRU cache-line replacement can be ineffective for applications with large working sets or non-localized memory access patterns. Specifically, in last-level processor caches, LRU can cause cache pollution by inserting non-reuseable elements into the cache while evicting reusable ones. The work presented in this paper addresses last-level cache pollution through a dynamic operating system mechanism, called ***ROCS****, requiring no change to underlying hardware and no change to applications.*

ROCS employs hardware performance counters on a commodity processor to characterize application cache behavior at run-time. Using this online profiling, cache unfriendly pages are dynamically mapped to a pollute buffer *in the cache, eliminating competition between reusable and non-reusable cache lines. The operating system implements the pollute buffer through a page-coloring based technique, by dedicating a small slice of the last-level cache to store non-reusable pages. Measurements show that ROCS, implemented in the Linux 2.6.24 kernel and running on a 2.3GHz PowerPC 970FX, improves performance of memory-intensive SPEC CPU 2000 and NAS benchmarks by up to 34%, and 16% on average.*

1. Introduction

Cache pollution can be defined as the displacement of a cache element by a less useful one. In the context of processor caches, cache pollution occurs whenever a non-reusable cache line is installed into a cache set, displacing a reusable cache line, where reusability is determined by the number of times a cache line is accessed after it is initially installed into the cache and before its eviction.

Modern processor caches are designed with the premise that recency ordering serves as good prediction for subsequent cache accesses. Hence, caches typically install all cache lines that are accessed by the application, expecting subsequent accesses to the same lines due to *temporal and spatial locality* in the application's access pattern. Moreover, hardware data prefetchers are widely used to populate caches by tracking sequential or striding access patterns to predict future accesses.

Both LRU caching and prefetching have been shown to be effective for the performance of many applications. As such, these techniques have been incorporated into processor design for decades. However, it has also been noted that these two techniques can perform poorly for some access patterns found in real workloads. In essence, the *mispredictions* that occur in LRU caching and prefetching are responsible for cache pollution, where lines are brought into the cache with the expectation of timely reuse, but which in fact are not accessed in the near future, replacing potentially more useful cache lines.

The computer architecture community has extensively studied the problem of cache pollution caused by both LRU placement and prefetching. Numerous enhancements to the memory hierarchy have been proposed and shown to be effective in mitigating the negative performance impact of cache pollution [9][10][11][16][20][21][22][27]. Unfortunately, however, modern processors continue to be shipped with little or no tolerance to last-level cache pollution. *The goal of this work is to address last-level (L2, in this study) cache pollution caused by LRU placement and prefetching, providing a transparent, software-only solution.* We focus on the design of a run-time cache filtering technique at the operating system level that can be deployed on current processors.

The main insight this work builds upon is that coarse-grain (page-level) cache behavior is indicative of cache line behavior for the lines within the page, especially as it relates to pollution behavior. We show how it is possible to monitor and characterize cache pollution at page granularity using commodity hardware performance counters. With a full cache

The equipment used in this work was kindly donated by IBM T.J. Watson Research Center. This research was partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and by the Director, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

profile of an application's address space, we can identify per-page cache pollution effects from the observed miss rates.

We introduce the concept of a software-based *pollute buffer*, implemented in the last-level cache for the purpose of hosting application data likely to cause pollution. We exploit the use of the pollute buffer in conjunction with online cache profiles to improve the performance of memory intensive applications.

We describe our implementation of a Run-time Operating system Cache-filtering Service (ROCS), in the context of the Linux kernel. Running on a real PowerPC 970FX processor, we evaluate its benefits, showing performance improvements of up to 34% on workloads from SPEC CPU 2000 and NAS, and an average of 16% on 7 memory intensive benchmarks from these suites. We also show that, in addition to reducing L2 cache miss rates of application data, mitigating cache pollution can benefit performance by reducing the L2 cache miss rate of performance critical meta-data, such as page-tables.

The remainder of the paper is organized as follows. Section 2 describes background material on software-based cache partitioning and hardware performance counters. In Section 3, we describe the use of hardware performance counters for monitoring application L2 behavior, and provide empirical evidence that monitoring last-level cache behavior at page granularity is meaningful for tracking cache pollution. We describe our software-based cache pollute buffer and its use, in Section 4. The run-time OS cache-filtering service is discussed in Section 5. The evaluation is presented in Section 6. In Section 7, we discuss related work and finally conclude in Section 8.

2. Background

In this section we provide a brief background on the two essential components used in this work: software cache partitioning and hardware performance counters. We leverage the concept of software cache partitioning to implement a software-based *pollute buffer* in the last-level cache. In addition, we make unconventional use of hardware performance counters to obtain online cache characterization of the target application's memory pages.

2.1. Software Cache Partitioning

Software partitioning of physically indexed processor caches (L2/L3) is possible through operating system page-coloring [14][17][24][29]. In physically indexed caches, physical addresses of data are used to map data into cache sets. The hashing function used for indexing into the cache must utilize enough bits from the address to cover the entire cache. Due to the relatively large size of current caches, these surpass the page offset bits, as shown in Figure 1. As a consequence, the choice of virtual to physical mapping influences the specific cache sets which store application data.

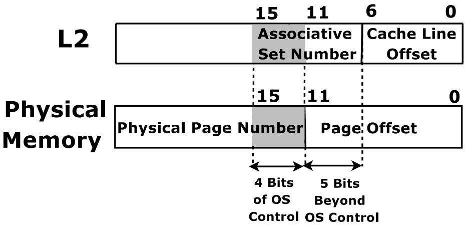


Figure 1. Cache indexing of physical addresses in the PowerPC 970FX processor, with 128B cache lines, L2 cache with 512 sets, and 4KB pages.

Conceptually, the set of pages which share the indexing bits above the page offset form a congruence class. Each congruence class is mapped to a fixed partition of the cache. The number of congruence classes available is equal to 2^n, where n is the number of bits used for cache hashing above the page offset. In the case of the processor used in this study, the PowerPC 970FX, the L2 is organized into 512 sets, resulting in $\log_2(512) = 9$ bits used for indexing. The four most significant of those bits can be used to determine congruence classes. As a consequence, the operating system can control $2^4 = 16$ different cache congruence classes or *partitions*.

2.2. Hardware Performance Counters

Processor manufacturers have equipped modern processors with performance monitoring units (PMU). These are exposed to system software in the form of hardware performance counters (HPCs) and attendant control registers. The PMU can be programmed to count a wide range of micro-architectural events, including committed instructions, branch mispredictions, and L1/L2 hits and misses. Depending on the specific processor, the PMU events can number in the hundreds. However, the number of physical HPCs is much lower, typically less than 10, limiting the number of events that can be monitored concurrently.

HPCs can be read either through polling or through interrupts. For fine-grained monitoring, where performance characterization of a small time-slice or section of code is desired, the monitoring software can poll the content of the HPCs at the beginning and end of the slice. Coarse-grained monitoring, on the other hand, is done by programming the PMU to generate an interrupt when an HPC overflows. The operating system can then notify monitoring software, or simply accumulate the values.

Instruction sampling, a technique used by numerous profiling software such as DCPI [1], Oprofile [1], and VTune [2], also uses hardware performance counters. For instruction sampling, the PMU is programmed with a sampling threshold and an interrupt is raised every time the threshold number of events of a specified type occur in the processor. Profiling software then attributes the event to the instruction of the current application program-counter, resets the HPC and continues to profile. After many samples, it is possible to determine the contribution of each instruction in the occurrence of the programmed event.

The wide-spread use of aggressive out-of-order processors has made interrupt-based instruction sampling less accurate. Since performance monitoring interrupts are *imprecise*, it is difficult to determine the exact instruction and/or address which triggered an event. This motivated ProfileMe, which attempts to provide accurate sampling by *marking* a single instruction in the pipeline and reporting events triggered by that instruction [8]. The PowerPC processor used in our work supports a similar mechanism, called *instruction marking*.

The biggest disadvantage of instruction marking is that of low recall: only a small subset of the instructions which cause an event of interest are profiled. This comes from the fact that only one instruction can be marked in the pipeline at a time. While the marked instruction is traversing the pipeline, other instructions of interest may be concurrently executing and will pass undetected. Another contributor to low recall is the fact that marking occurs early in the pipeline, typically in the fetch unit. At that stage, it is not yet possible to determine if the marked instruction will cause any of the events of interest. In the case that it does not, the PMU must wait for the current instruction to commit before a next instruction can be marked.

3. Address-Space Cache Characterization

Our dynamic cache-filtering mechanism is based on runtime cache characterization at the page level. Our system uses address-space cache profiles to identify memory pages that cause last-level cache pollution. In this section, we demonstrate how hardware performance counters can be used to build cache behavior profiles at page granularity. In addition, we present the characterization of 8 benchmarks from the SPEC CPU 2000 benchmark suite, providing insights for the creation of our software pollute buffer. We provide evidence that these workloads exhibit cache pollution that can be accurately identified at page granularity.

3.1. Exploiting Hardware Performance Counters

For obtaining page-level L2 cache profiles of applications, we have built a Linux kernel module which uses the PMU of the processor. In essence, the monitoring module identifies the data addresses of load requests that miss the L1 data cache, as well as the level of the cache hierarchy in which the data was found (either the L2 or main memory on our hardware).

The kernel module configures the PMU to mark load/store instructions for monitoring, as described in Section 2.2. We specifically target loads that miss the L1 data cache (i.e., L2_HITS and L2_MISSES) so that a PMU interrupt is generated on every such event. For every PMU interrupt received, the module determines which HPC overflowed to determine from where the cache line is being fetched. It also reads the address provided by the *sampled-address data register* (SDAR) to obtain the virtual address of the cache access. In the PowerPC architecture, the SDAR provides the data address of the last marked load/store instruction, which in the case of our PMU configuration, will contain the loaded data address that caused the PMU interrupt.

Our module assembles page-level statistics on the addresses sampled, and creates a cache profile for each targeted address space. Each profile contains miss rates, as well as frequency of accesses, for every virtual page in the address space. In essence, our profiling module performs accurate *data sampling* of L2 cache events. This technique is analogous to the widely used *instruction sampling*.

Aggressive interrupt handling affects the accuracy of the profiles generated in that the PMU interrupt handler itself introduces L1 pollution. When handling an interrupt on an L1 miss, the data items used by the interrupt handler can cause (hot) application data to be evicted to the L2. If the evicted line is still hot, it will immediately be fetched from the L2, adding artificial L2 hits for some pages in the profile.

To eliminate the effects of interrupt handling interference, we throttle the rate of interrupts so that hot cache lines evicted from the L1 by the interrupt handler have time to be brought back into the L1 by subsequent application accesses. We have empirically verified that for memory intensive SPEC CPU 2000 benchmarks, interrupt rates lower than 1 interrupt per 5K to 10K cycles cause minimal impact to the cache profile.

3.2. Page-Level Cache Behavior

Page-level profiling is oblivious to potential miss-rate[3] variances of cache-lines within a page. Nonetheless, we demonstrate that it provides insightful information on the application cache behavior at run-time. We show, in the next section, how page-level profiling can help identify pages that cause cache pollution. In addition, we analyze the cache profile of the *art* benchmark, as a case study, showing how the profile relates to *art*'s source code.

[1] http://oprofile.sf.net/

[2] http://www.intel.com/cd/software/products/asmo-na/eng/vtune/

[3] In this work, L2 miss rate is defined as the number of L2 misses divided by all L2 requests.

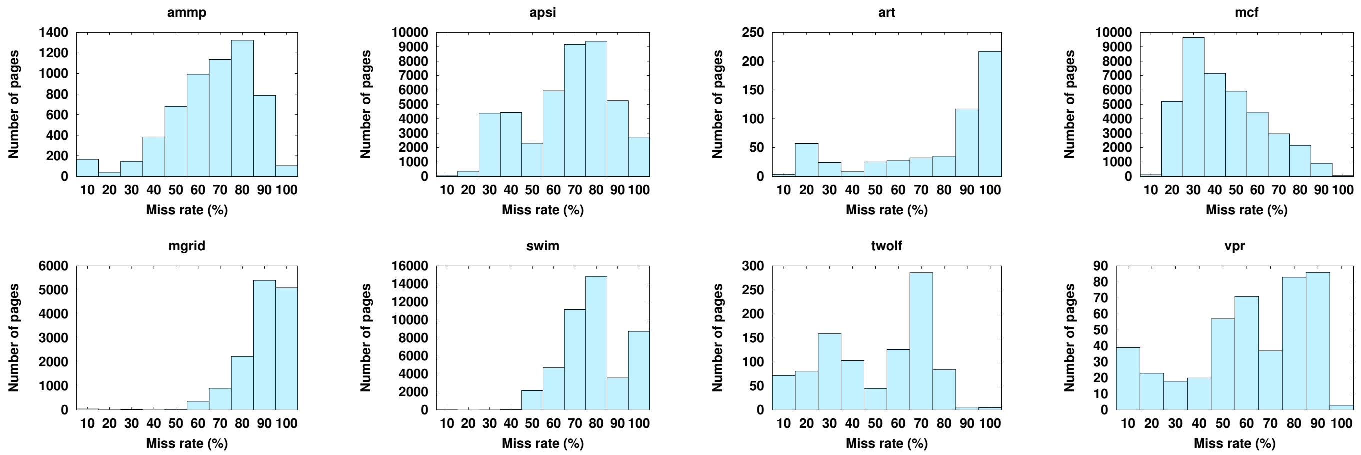


Figure 2. Page-level L2 cache miss rate characterization. The histograms show per-page distribution of miss rates.

3.2.1. Classifying Pollution. An essential function of our cache filtering system is to classify pages with respect to pollution, such that it is possible to determine which pages should have restricted cache access. Previous work has attempted to classify cache pollution, at cache line granularity, based on single-use or zero reuse of cache lines [20][22].

Given the lack of fine-grained monitoring in commodity hardware, we make the simplifying assumption that the L2 miss rate of a page directly correlates with the degree of its cache pollution. The empirical justification is that pages with high miss rates experience little benefit from being cached, since each miss results in an eviction of a potentially useful cache-line. That is, pages with high miss rates cause high rates of cache-line evictions.

The per-page miss rate can be viewed as an inverse measure of the probability of reusing a cache block of the page after its insertion in the cache. We show in the results section that pages with low probability of reuse (1) have limited benefit from caching and (2) negatively impact pages with high probability of reuse. Our approach uses this assumption to constrict the caching of pages with low reuse probability, consequently increasing the effective cache space for pages with higher probability of reuse.

Figure 2 shows the distribution of per-page miss-rates for 8 memory intensive workloads from the SPEC CPU 2000 benchmark suite over the entire execution of the application. The graphs show that it is possible to identify a significant number of pages that exhibit high miss rates, and therefore, are likely to cause pollution in the cache. From the graphs shown, the only benchmark which does not show a large proportion of pages with high miss rates is *mcf*.

3.2.2. Case Study: *art*. We show that for some workloads cache miss behavior can be characterized at an even coarser granularity than pages, using the *art* benchmark. Figure 3 shows an example of the collected cache profile for the entire execution of *art*, depicting a compact view of its address space. In the profile, both the total number of accesses and the miss rates are shown for each page.

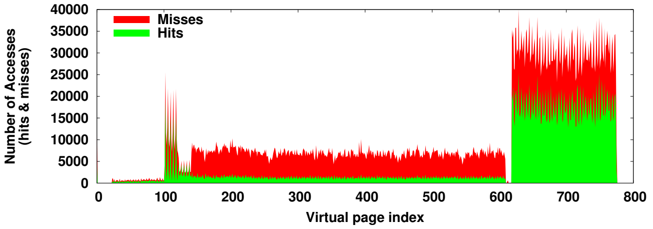


Figure 3. Page-level L2 cache miss rate characterization for *art*. The histogram shows a compact view of the address space, each bar representing accesses (hits and misses) to a page.

Art implements a neural network for image recognition. The significant data structures of *art* are comprised of three 2-dimensional arrays: f1_layer, an array of neurons, and; tds and bus, arrays of weights. In the profile shown, there are two large memory regions with distinct L2 cache behavior. The contiguous memory region to the left (pages 100 to 600) contains the two arrays bus and tds. Accesses to these two arrays correspond to 39% of all accesses to L2 and their pages obtain an 81% miss rate in the L2. The rightmost memory region (pages 620 to 780) contains the f1_layer array. This regions corresponds to 56% of all L2 accesses and has an average miss rate of 42%.

This profile shows that the leftmost memory region (bus and tds) does not benefit significantly from the L2 cache. On the other hand, the benefit from caching the rightmost memory region is visibly higher. As we will show in our evaluation, limiting the L2 cache space of bus and tds arrays improves the cache hit rate of f1_layer by preventing L2 pollution and consequently improving the overall L2 hit rate and application performance.

This example provides an important insight: applications contain distinct memory regions, each with its own uniform cache behavior. These regions are sufficiently coarse-grain so that page-level cache management is applicable. Coarse-grain tracking and management of memory has been observed before in the literature for improving snoop-coherence and data prefetching [7][32]. This work confirms that the same observation applies to caching behavior.

3.2.3. Prefetching Interference. Modern high performance processors employ data prefetchers in order to minimize access latency. Due to their significant performance benefits, data prefetchers have been implemented in multiple levels of the memory hierarchy. The L1 data prefetcher implemented in the PowerPC processor, although quite simple, is able to significantly improve performance of programs with sequential memory references.

For characterizing cache behavior, and specifically for classifying cache pollution of pages based on cache miss rates, the prefetcher poses two problems. The first problem is the existence of *invisible lines*. Cache lines from pages that are prefetched into L1, although occupying entries of the L2, are not fully counted in the profile, because the profile is based on the source resolution of demand loads that miss L1, leading to a perceived lower occupancy in the L2.

The second problem is that of *artificial hits*. An artificial cache hit occurs when prefetching from memory is not timely enough to bring the line to L1, but timely enough for L2 insertion. As a consequence, an L1 miss occurs, which is satisfied from the L2. This causes L2 cache hits to be incorporated into the profile even though these hits are *not* a result of cache line reuse, but a result of prefetching. Effectively, *artificial hits* make pages appear to be more reusable than they are and, therefore, they appear to benefit from caching.

Ironically, the opposite conclusion should be drawn from highly prefetchable pages. For pages that exhibit hits from prefetching, caching becomes less important for hiding memory access latency. For overall performance, it would be better to give pages that are not prefetchable higher priority in the cache. In addition, prefetching can bring useless lines into the cache when prefetch predictions are too aggressive, thus causing pollution. The possibility of pollution is another reason why prefetchable pages should have restricted cache access.

To overcome these issues, we disable the hardware data prefetcher while generating cache profiles, but enable it for the remainder of the application. To illustrate both problems mentioned above, Figure 4 shows the memory profile of the *wupwise* benchmark with and without prefetching. The eight rightmost memory regions starting at virtual page index 20000 have significantly different characteristics depending on whether prefetching is enabled or not; prefetching reduces the perceived occupancy in L2 (*invisible lines*), and increases perceived reuse probability (*artificial hits*).

4. Software-Based Cache Pollute Buffer

The insights from the previous section motivate cache management at memory page granularity. For this purpose, we have designed a software-based cache *pollute buffer*. The pollute buffer provides a mechanism to restrict specific memory pages, deemed to pollute the cache, to a small partition of the cache. It is meant to serve as a staging space for cache lines that exhibit bursty or no reuse before eviction. By restricting cache unfriendly pages to the pollute buffer, we eliminate competition between pages that pollute the cache and pages that benefit from caching.

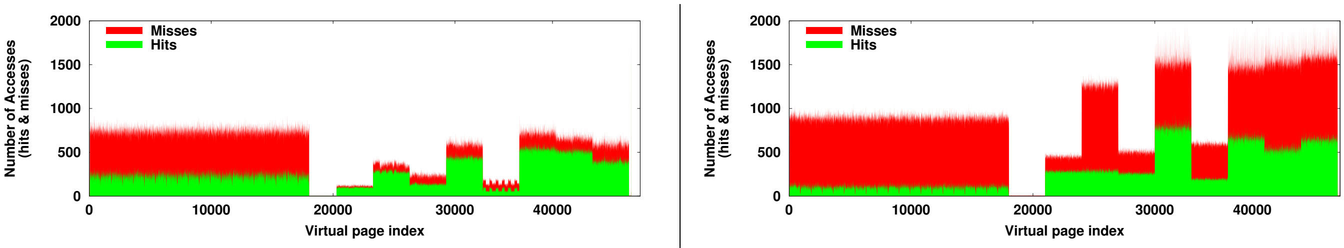
In our system, the pollute buffer is implemented with software-based cache partitioning. We do so by dedicating a single *partition* of the L2 to act as the pollute buffer. Figure 5 illustrates the design of the pollute buffer using page-coloring. As described in Section 2.1, this is possible by allocating physical pages that map to a specific section of the cache, whose cache indexing bits are in the same congruence class.

An inherent property of the pollute buffer is that, since it uses a partition of the last-level cache, it is amenable to (1) errors in the classification of pages with respect to pollution, and (2) variances in the miss rate of individual cache lines of a pollute page. The pollute buffer, although small in size, continues to allow hits on frequently accessed cache lines since the LRU replacement policy remains unchanged. After all, the pollute buffer is part of the last-level cache.

In order to manage cache pollution at run-time, our system requires moving application pages from one cache partition to another (from the non-pollute part of the cache to the pollute buffer, or vice versa). To perform this task, we must *copy* the content of the old page to a newly allocated page that maps to the target partition of the cache. This involves (1) allocating a new empty page that maps to the desired partition, (2) removing the appropriate page-table entry in the application page-table, potentially flushing a TLB entry, (3) performing a physical page copy, and (4) reinserting the page-table entry, with the physical address of the new page.

4.1. Kernel Page Allocator

We have modified the Linux kernel page allocator to efficiently allocate physical pages so that they map to specific partitions of the cache. Default Linux relies on two structures to manage free pages for allocation. Each processor contains a local list of recently freed (hot) pages for fast allocation. A global structure, which employs the buddy allocator algorithm, contains the majority of free pages. The buddy structure is organized as a binary tree that clusters



a) Profile with prefetching enabled

b) Profile with no prefetching

Figure 4. Page-level cache miss rate characterization for *wupwise*, with and without prefetching.

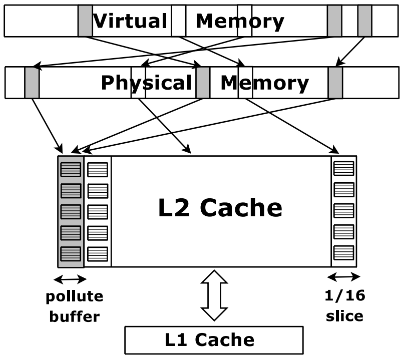


Figure 5. The software pollute buffer is implemented by dedicating a partition of the last-level cache to host lines from pages that cause cache pollution.

contiguous physical pages hierarchically. The leaf nodes (0-order) contain single pages, the next level (1-order) contains clusters of 2 physically contiguous pages, and so on. This organization allows for fast allocation of physically contiguous pages, which are needed by devices that do not support virtual memory.

Our modified Linux splits both the CPU and buddy allocator lists into 16 lists; one list for each partition of the L2. When populating the list with a new free page, the physical address of the page is used to determine the correct list to use. To satisfy new page allocation requests that map to a specific cache partition, the allocation can be quickly serviced by removing a page from the appropriate free-list. In cases where the allocation specifies multiple allowed partitions, round-robin is used between the lists, emulating bin-hopping [12].

5. Run-Time OS Cache-Filtering Service

In the previous two sections, we have presented page granularity cache characterization and the concept of a software-based cache pollute buffer. With these two constructs, we now describe ROCS, our implementation of a run-time operating system cache-filtering service. We show how online memory page cache profiles can be collected and used to determine which application pages should be mapped to the pollute buffer.

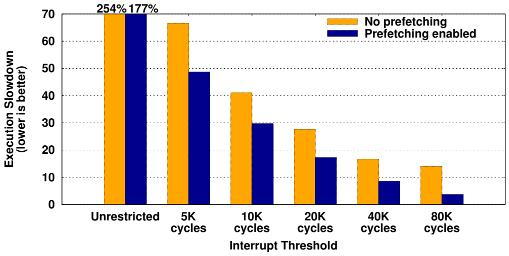


Figure 6. Overhead sensitivity of monitoring *art*.

5.1. Online Profiling

In Section 3 we presented a collection of address space cache profiles. The profiles shown were gathered from *complete* execution runs. Unfortunately, this is impractical to do for run-time software cache management, as the overhead is prohibitively high: profiling involves recording L1 misses through an operating system interrupt handler where each interrupt entails a complete pipeline flush, interrupt delivery, fetch of interrupt handling code and execution of the handler itself.

Figure 6 shows the overhead, in terms of execution time slowdown, of *art* with varying interrupt frequencies. As discussed in Section 3.1, unrestricted monitoring of L1 misses distorts profiling as the interrupt handler evicts application data from L1. Fortunately, interrupting every 5 to 10 thousand cycles has a two-fold benefit: more precise L2 characterization (as explained in Section 3.1) and significantly lower overhead. Unfortunately, overheads of 15-65% are still prohibitive for online cache reconfiguration.

To reduce overhead, ROCS uses phase-based sampling. Application cache profiles are gathered for a short period of application execution. During each period, we profile with the smallest threshold that yields acceptable accuracy (5K cycles). We use this sample profile as a representation of the current application phase. Further samples are taken when a *coarse-grain* phase change is detected. Phase changes can be cheaply monitored with hardware performance counters, by measuring, for example, IPC or the L2 misses per kilo instructions (MPKI) of the application [23]. Since only coarse-grain phases are tracked, the IPC may be computed every 1 billion cycles, which incurs negligible overhead (one interrupt per billion cycles).

It is important to note that ROCS, as an operating system component, is able to solely target processes or threads that exhibit high L2 miss rate. When non-targeted threads are scheduled, profiling can be disabled or restricted to monitoring L2 miss rate. In this way, no overhead is observed for application threads exhibiting low L2 miss rates.

5.2. Dynamic Page-Level Cache Filtering

Given a page-level cache profile, a fundamental challenge is to identify which pages should be restricted to the pollute buffer in order to improve application performance. Addressing this challenge requires predicting the effects of restricting pages to the pollute buffer. This is analogous to predicting the effect of partitioning shared caches between different applications, with the difference being that we are potentially partitioning the cache between different memory pages of the *same* application.

Previous work on online prediction of the performance impact of shared cache partitioning between different applications have required information on per-application miss-rates as a function of cache size (e.g., miss-rate curves) [19][22][25]. In our context, this would require obtaining per memory-region utility information, which is too expensive to compute in software at run-time – Berg *et al.* report a 40% average run-time overhead with sparse sampling [3], but their technique is oblivious to memory regions.

Given the high overhead and complexity of analytically predicting interference in the cache, we instead employ an empirical search algorithm. From the cache profile described, we construct a miss-rate stack, containing all pages in the monitored address space, ordered by miss rate (highest miss-rate at the top). We then proceed to monitor the improvement of mapping different number of pages from the stack to the pollute partition.

The pseudo-code of the search algorithm is listed in Algorithm 1. Starting at the top of the miss rate stack, where the pages are most likely to be cache polluters, a number of pages are remapped from their original cache partition to the dedicated pollute buffer slice. We use the hardware performance counters to evaluate the performance (IPC) of this mapping. Next, a subsequent number of pages are remapped to the pollute buffer, adding to the pages already mapped to the pollute buffer. This iterative algorithm continues until a minimal set of pages is reached. The best configuration is recorded, and the stack is traversed upwards, restoring the excess pages of the pollute buffer to the non-pollute slices of the cache, if necessary.

Algorithm 1 FindPollutePages: returns the number of pages from $mrateStack$ mapped to the pollute buffer.

```
procedure FINDPOLLUTEPAGES($mrateStack$, $stepSize$)
    $index \leftarrow mrateStack.size()$;
    while $index > minPages$ do
        MAPTOPOLLUTEBUFFER($mrateStack$,$index$,$stepSize$);
        $performance \leftarrow$MONITORPERFORMANCE();
        if $performance > best$ then
            $best \leftarrow performance$;
            $pollute\_index \leftarrow index$;
        end if
        $index \leftarrow index - stepSize$;
    end while
    UNMAPFROMPOLLUTEBUFFER($mrateStack$,$index$,
                              $pollute\_index - index$);
    return $mrateStack.size() - pollute\_index$;
end procedure
```

Despite the simplicity of the algorithm, we found that the algorithm takes 3 billion cycles, on average, and 7 billion, in the worst case, for SPEC CPU 2000 benchmarks. On our evaluation platform, with a 2.3GHz processor, this is equivalent to average of 1.4 seconds, and a worst case of 3.1 seconds. This search is significantly faster than published approaches for deriving L2 cache miss-rate curves in software. In addition, we show in the next section, that the overhead of searching for a good pollute mapping incurs, in most cases, less overhead then profiling the application address space.

6. Evaluation

Table 1 lists the relevant architectural parameters of our evaluation platform. The system under test is a PowerMac G5, with 2 PowerPC 970FX processor chips, clocked at 2.3GHz, built on a 90nm process. For all cases in our evaluation, we have restricted application, monitoring and remapping to a single processor, disabling the second CPU in the operating system. Baseline results were obtained using the Linux kernel version 2.6.24. ROCS was developed using the same Linux kernel version.

We evaluated ROCS using SPEC CPU 2000 and NAS-serial [2] (serial version of NAS 3.3) benchmark suites. For SPEC CPU 2000, the reference inputs were used, and "Class B" inputs were used for NAS-serial. Table 2 lists the benchmarks from these suites that exhibit L2 miss rates greater than 25% on our platform, along with the most relevant characteristics collected using hardware performance counters. All other benchmarks from the suites with less than 25% miss rate displayed far lower misses per kilo instructions (MPKI). We did not consider applications with low L2 miss rates, since our technique is targeted at workloads that exhibit L2 cache pollution. Recall that ROCS is able to identify

Table 1. Characteristics of the 2.3GHz PowerPC 970FX.

Component	Specification
Issue width	8 units (2 FXU, 2 FPU, 2 LSU, 1 BRU, 1 CRU)
Reorder Buffer	100 entries (20 groups of 5 instructions)
Cache line	128 B for all caches
L1 i-cache	64 KB, direct-mapped, 1 cycle latency
L1 d-cache	32 KB, 2-way, 2 cycle FXU latency, 4 cycle FPU latency
L2 cache	512 KB, 8-way, 12 cycle latency
Memory	2GB, 4KB pages, 300 cycle latency (avg.)

Table 2. Benchmark characteristics.

Benchmark	Exec. time	Instrs.	IPC	L2 MPKI	L2 Miss Rate
ammp	9m00s	365B	0.30	7.5	52%
apsi	5m29s	334B	0.45	6.5	61%
art	3m10s	44B	0.10	69.0	75%
mcf	8m23s	51B	0.05	68.3	54%
mgrid	2m43s	255B	0.70	3.0	25%
swim	18m33s	262B	0.10	22.7	75%
twolf	9m11s	261B	0.22	9.7	35%
vpr	2m10s	96B	0.33	5.9	25%
CG	22m42s	137B	0.16	42.1	59%

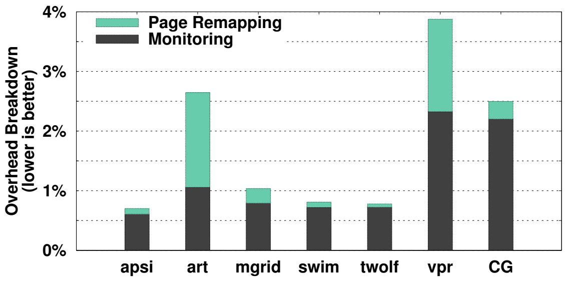


Figure 7. Run-time overhead breakdown of ROCS.

applications with low L2 miss rates while incurring negligible overhead (see Section 5.1).

The size of the pollute buffer used in all experiments was 1/16th of the L2 cache; in our case, 32KB. All benchmarks were compiled for a 64-bit environment. We always present the average results obtained from three consecutive complete runs. An initial (discarded) run was used to ensure that all necessary files and binaries were resident in memory.

We have excluded the *ammp* and *mcf* benchmarks from further performance analysis, as these benchmarks showed only around 1% improvement with ROCS. As shown in Figure 2, *mcf* does not contain a significant proportion of pages with high L2 miss rate. *Ammp*, on the other hand, has multiple short phases (some with 1 billion instructions), which limits the potential benefits of our cache-filtering technique.

6.1. Overhead

Figure 7 depicts the run-time overhead of ROCS, split into two components: monitoring and page remapping. The large variance in overheads between different applications is primarily due to the different execution lengths of the benchmarks (listed in Table 2). Since most of these benchmarks exhibit stable IPC after initialization (when monitoring IPC at 1 billion cycle granularity), ROCS initiates only 1 or 2 monitoring phases. Furthermore, it is interesting to note that the overhead caused by monitoring overshadows the overhead due to remapping when application address spaces become large, despite the increase in pages remapped (see Table 3). Fortunately, applications that consume many pages typically also run for longer periods of time in stable phases in order to consume their entire data set. Consequently, we see an overall average overhead of 1.6%, with 3.8% being the worst case. As we show in the next section, this overhead is more than recovered by the performance improvements obtained through our technique.

6.2. Performance Results

Figure 8 shows the run-time speedup of three different cache filtering schemes. In all three schemes, the page miss rate stack was collected at run-time, after the first 4 billion cycles, in order to avoid application initialization. *Best Offline* consists of a static exhaustive search for optimal stack values (number of pollute pages). This involves running the application multiple times, varying the number of pages to remap to the pollute buffer. The *ROCS* system incorporates the dynamic search algorithm. In *ROCS*, the hardware data prefetcher is disabled while monitoring the application for its miss rate stack, but is enabled otherwise. Finally, we also show the performance of *ROCS* given a miss-rate stack generated with the hardware prefetcher enabled.

The average improvement of ROCS over Linux for the 7 benchmarks is 16.6%. The largest performance win of 34.2%, comes from *swim*. In all cases, we see that ROCS is able to approach the performance of optimal offline search. The worst case occurs with *apsi* where ROCS achieves 2.1% less speedup than the offline search.

The MPKI reductions of the benchmarks running under ROCS are shown in Figure 9, and average of 12.2%. For the most part, the MPKI improvements correlate with the performance improvements. The glaring exception is *swim*, which we analyze separately in Section 6.4.

The number of pages chosen as polluters by ROCS, and remapped to the pollute buffer is shown in Table 3. It is

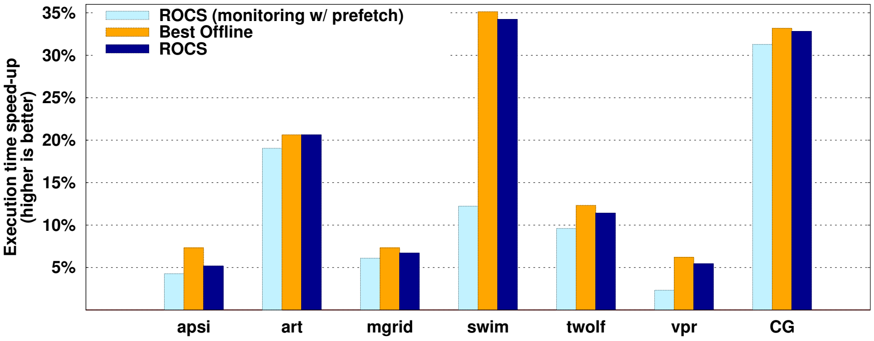


Figure 8. Performance improvement of ROCS over default Linux.

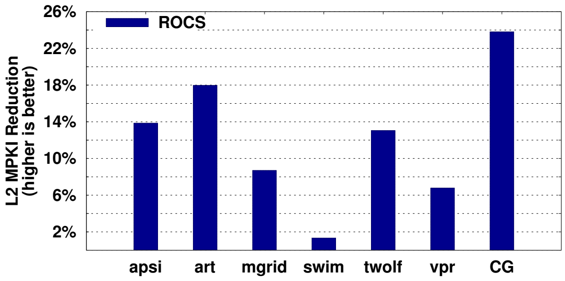


Figure 9. MPKI reduction with ROCS over a default Linux.

Table 3. Classification of pollute pages.

Benchmark	Number of Pages	Number of Pollute Pages (% of all)
apsi	44159	2676 (6%)
art	865	607 (70%)
mgrid	14433	3157 (21.8%)
swim	45490	14335 (31.5%)
twolf	1530	181 (11.8%)
vpr	812	183 (22%)
CG	40818	7026 (17.2%)

interesting to note that, with the exception of *swim*, there is a correlation between the fraction of pollute pages chosen by ROCS, and the profile information shown in Figure 2. Applications that were shown to contain a higher fraction of pages with high miss rate, obtained their best improvement by classifying higher fractions of pages as cache polluters. This correlation corroborates our initial assumption that the degree of cache pollution, at the page-level, is directly related to its observed miss rate.

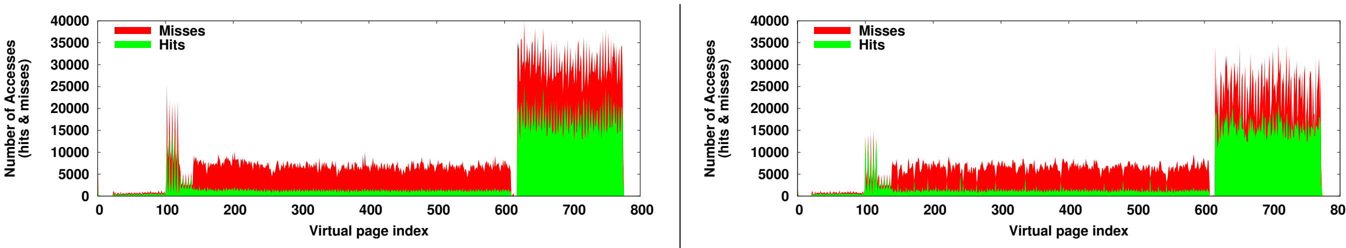
To further analyze our results, we discuss two specific cases in greater detail: *art* and *swim*.

6.3. Case study: *art*

Figure 10 shows two cache profiles of *art*'s address space: on the left (a) we replicate the image from Section 3.2.2 containing the characterization of *art* on default Linux, and on the right (b) we show the profile of *art* on ROCS. As discussed, the leftmost memory region, with visibly high miss rates, contains two 2-dimensional arrays, bus and tds arrays. The rightmost region contains a single 2-dimensional array, f1_layer.

ROCS chooses to predominantly classify pages from the leftmost memory region as polluters, mapping them to the pollute buffer. The effects on the L2 access pattern can be seen on the graph to the right (b). With less competition from polluting pages, the f1_layer array sees a 16% reduction in its L2 miss rate (from 42% to 35%). In addition, a decrease of 6.7% in the *total* number of accesses to this region is visible in the graph. This decrease comes mainly from the L1 data prefetcher; with the reduced L2 miss rate, data prefetching becomes more effective, since prefetched data is found in L2 instead of main memory. Consequently, L1 is able to capture more accesses to this region of memory. In fact, we verify that the L1 MPKI receives on overall reduction of 7.7% for all of *art*.

It is also important to observe the impact of restricting the leftmost memory region to the pollute buffer. In this particular case, the memory region did not suffer an increase in L2 miss rate, as may have been expected. In fact, a reduction of 1% was measured, as ROCS kept some pages from the leftmost region in the non-pollute partition of the cache. In essence, the hits seen in the leftmost memory region are primarily due to the short-term reuse of lines from the bus and tds arrays. The pollute buffer, while quite small, is still able to cache these lines for a short period of time, enough to allow reuse of lines with bursty accesses. This fact illustrates a fundamental difference between the pollute buffer and cache bypassing based approaches for addressing cache pollution at the last-level cache [9][20]. Further discussion on cache bypassing is presented in the related work section.



a) Default Linux (no cache filtering)

b) ROCS (with cache filtering)

Figure 10. Page-level cache miss rate characterization for *art*. The histogram on the left shows the execution with no filtering, and on the right, the effects of filtering are shown.

6.4. Case study: *swim*

Out of all benchmarks evaluated, *swim* observed the highest performance improvement (34% speedup on execution runtime). Perhaps surprisingly, the performance improvement was not a result of L2 MPKI reduction of *swim*'s data; in fact, a breakdown of stall cycles shows that the performance difference comes from handling TLB misses. For *swim*, TLB miss handling contributes to 55% of the stalls when run under Linux. With ROCS, the stalls due to TLB misses decrease to 29% of the total stall cycles.

The main reason for the reduction of TLB miss-handling stalls is that the reduced L2 cache pollution with ROCS allows the hardware page-table walker to find more page-table entries in the L2, incurring fewer full main memory stalls. The PMU in the PowerPC 970FX processor does not contain support for counting the source (in the memory hierarchy) of page-table entries. However, the PMU does provide an event to count the number of cycles used by the hardware page-table walker. Our experiments show that TLB miss handling under ROCS is 69% faster than on Linux without ROCS (an average of 136 cycles per page-table walk on ROCS versus 443 cycles without it).

This illustrates another source of performance benefits from reducing last-level cache pollution. This case study shows that ROCS also reduces cache pollution effects on meta-data in the L2, mitigating the interference with infrequently accessed, but performance critical, data such as page-table entries.

7. Related Work

7.1. Cache Bypassing

Previous research most similar to ours is the study of *cache bypassing* to reduce cache pollution. With cache bypassing, selected cache lines are refrained from being installed into the cache (or, to at least one of the levels of the cache). If cache lines are chosen correctly, bypassing the cache reduces the displacement of reusable lines.

The majority of work exploring cache bypassing have focused on reducing cache pollution in the first (L1) level cache [6][11][27][31]. More recently, cache bypassing for last-level caches has also been explored [9][13][20]. All of the studied dynamic schemes require hardware support and propose non-trivial changes to the processor and cache architecture, typically in the form of non-trivial enhancements to the cache tag-array, along with constant updates of meta-data. The work we propose has the advantage of being implemented on existing commodity hardware.

Moreover, there is an important difference between traditional cache bypassing and the use of a pollute buffer in the last-level cache. Cache bypassing is an aggressive optimization; if a bypass decision is incorrect, a high price is paid, since the line must be re-fetched from a slower level cache or main memory. With the pollute buffer, however, the possibility for reuse of cache lines is still supported. Johnson *et al.* also note this and propose enhancing the L1 cache with a bypass buffer in their L1 cache bypassing proposal [11].

7.2. Software Cache Partitioning

Page coloring, the basic mechanism for software cache partitioning, has been used in previous studies, showing performance improvements due to reduced mapping conflict misses [4][12]. In this work, however, we do not focus on mapping conflict misses, as they are greatly attenuated in modern last-level caches due to the high associativity used.

The first application of software cache partitioning through page coloring was presented by Wolfe [29]. In his work, he proposed partitioning the cache to achieve predictable performance in preemptible real-time systems. By ensuring that the cache content of preempted real-time applications remains intact, partitioning minimizes the interference of preemption on the application's performance. Software cache partitioning for real-time systems has gained attention of the community in subsequent work [14][18][28].

In recent years, there has been a resurgence of software cache partitioning due to the commercial impact of chip-multiprocessors (CMPs). Many CMPs are designed with *shared* last-level caches (L2/L3), where applications can interfere with each other, potentially affecting the performance of the applications involved [5]. Software cache partitioning has been proposed as a solution to shared cache interference [7][15][25][26]. Although software-based cache partitioning is less flexible and incurs higher overhead then hardware solutions, it is attractive as it can be implemented on current, widely available CMPs. The main difference between cache partitioning for CMPs and our proposal is that CMP cache partitioning focuses on isolating cache usage of different applications. This work focuses on isolating cache usage potentially from *within* the same application, separating reusable cache lines from non-reusable ones.

7.3. Cache Replacement Policies

Several studies have proposed enhancing the LRU cache replacement policy to avoid pollution [10][16][21][30]. These studies attempt to augment LRU replacement decisions with information about locality, reuse and/or liveness. In contrast, we do not propose changing the replacement algorithm, but propose managing the competition of cache space so that the already existing LRU implementation performs better.

The dynamic insertion policy, proposed by Qureshi *et al.* [22], focuses on adapting the initial *placement* of caches lines in the LRU stack of each cache set, depending on the application access pattern. Similar to our work, the proposed dynamic insertion policy (DIP) reduces competition between caches lines by reducing the time to eviction of cache lines with thrashing access patterns. The main differences between DIP and our work are (1) DIP is applied at the way granularity of a set, while we study coarser-grain partitioning, allocating entire sets to the pollute buffer, (2) we manage cache competition per memory region of the application, while DIP adapts the entire cache based on application behavior, and (3) DIP can adapt to application phases in significantly less time than a software-based approach.

8. Conclusion

The *memory wall* problem has been studied intensively in the computer architecture and software communities and has resulted in a wide range of proposals for reducing the effects of memory latency on performance. Chip makers, for example, are dedicating increasing number of transistors for larger on and off-chip caches. However, not all workloads have responded to this increase with corresponding performance or hit ratio improvements.

We argue that proper management of the memory hierarchy is becoming more critical to achieve good performance and that software can play a significant and fruitful role in managing this hierarchy. We believe there are new opportunities to be explored with tighter cooperation between run-time software systems and the underlying hardware. This work presents a concrete example of this type of cooperation.

In this paper, we focused on attacking the specific problem of cache pollution in last-level caches. We observed that cache behavior, and pollution in particular, is uniform within a memory region, typically spanning multiple memory pages of application address space. We described the use of hardware performance counters, present on current hardware, to classify memory pages with respect to pollution.

We introduced the concept of a *pollute buffer* to host cache lines of pages with little or no reuse before eviction. We demonstrated how the last-level cache can be partitioned with operating system page coloring, to provide a pollute buffer within the cache. This technique requires no additional hardware support and no modifications to application code or binary.

Using these concepts, we described a complete implementation of a run-time operating system cache-filtering service (ROCS). We evaluated the performance of our system on 7 memory intensive SPEC CPU 2000 and NAS benchmarks, showing performance improvements of up to 34% on run-time execution, with 16% on average.

The system we implemented makes extensive use of processor performance monitoring units (PMU). Unfortunately, the architecture and interfaces of PMUs are substantially different for each processor family and in fact different across different processors within the same family. Standardizing the key PMU components and interfaces would, in our opinion, greatly accelerate the development and ubiquity of additional software optimizations, similar to the one we described in this paper. The impact of the IEEE 754 floating-point standardization efforts of 30 years ago should provide good motivation.

In conclusion, this work explored the use of rudimentary processor interfaces for monitoring and managing last-level caches. We believe that with better mechanisms for cooperation between hardware and software layers, further opportunities for improving performance would arise. We hope that this work serves as encouragement to hardware designers to include and expose more flexibility in processor components to the software layer.

Acknowledgments

Special thanks to Ioana Burcea for invaluable discussion, feedback and support throughout this work. We also thank Allan Kielstra for his enthusiastic response to our initial results and Reza Azimi for getting us interested in hardware performance counters.

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?" *ACM Transactions on Computer Systems*, vol. 15, no. 4, pp. 357–390, 1997.

[2] D. Bailey, T. Harris, W. Saphir, R. van der Wijingaart, A. Woo, and M. Yarrow, "The NAS parallel benchmarks 2.0," NASA, Tech. Rep. NAS-95-020, 1995.

[3] E. Berg and E. Hagersten, "Fast data-locality profiling of native execution," in *Intl. Conf. on Measurement and Modelling of Computer Systems (SIGMETRICS)*, 2005, pp. 169–180.

[4] E. Bugnion, J. M. Anderson, T. C. Mowry, M. Rosenblum, and M. S. Lam, "Compiler-directed page coloring for multiprocessors," in *7th Intl. Conf. on Arch. Support for Programming Languages and Operating Systems (ASPLOS)*, 1996, pp. 244–255.

[5] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting inter-thread cache contention on a chip multi-processor architecture," in *11th Intl. Symp. on High-Performance Computer Architecture (HPCA)*, 2005, pp. 340–351.

[6] C.-H. Chi and H. Dietz, "Improving cache performance by selective cache bypass," in *Twenty-Second Annual Hawaii International Conference on System Sciences*, vol. 1, Architecture Track, 1989, pp. 277–285.

[7] S. Cho and L. Jin, "Managing distributed, shared L2 caches through OS-level page allocation," in *39th Intl. Symp. on Microarchitecture (MICRO)*, 2006, pp. 455–468.

[8] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos, "ProfileMe: hardware support for instruction-level profiling on out-of-order processors," in *30th Intl. Symp. on Microarchitecture (MICRO)*, 1997, pp. 292–302.

[9] H. Dybdahl and P. Stenström, "Enhancing last-level cache performance by block bypassing and early miss determination," in *Asia-Pacific Computer Systems Arch. Conf.*, 2006, pp. 52–66.

[10] A. González, C. Aliagas, and M. Valero, "A data cache with multiple caching strategies tuned to different types of locality," in *Intl. Conf. in Supercomputing (ICS)*, 1995, pp. 338–347.

[11] T. L. Johnson, D. A. Connors, M. C. Merten, and W. mei W. Hwu, "Run-time cache bypassing," *IEEE Transactions on Computers*, vol. 48, no. 12, pp. 1338–1354, 1999.

[12] R. E. Kessler and M. D. Hill, "Page placement algorithms for large real-indexed caches," *ACM Transactions on Computer Systems*, vol. 10, no. 4, pp. 338–359, 1992.

[13] M. Kharbutli and Y. Solihin, "Counter-based cache replacement and bypassing algorithms," *IEEE Transactions on Computers*, vol. 57, no. 4, pp. 433–447, 2008.

[14] J. Liedtke, H. Hartig, and M. Hohmuth, "OS-controlled cache predictability for real-time systems," in *Real-Time Technology and Applications Symposium*, 1997, pp. 213–227.

[15] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan, "Gaining insights into multi-core cache partitioning: Bridging the gap between simulation and real systems," in *14th Intl. Symp. on High-Performance Comp. Arch. (HPCA)*, 2008, pp. 367–378.

[16] W. Lin and S. Reinhardt, "Predicting last-touch references under optimal replacement," University of Michigan, Tech. Rep. CSE-TR-447-02, 2002.

[17] W. L. Lynch, B. K. Bray, and M. J. Flynn, "The effect of page allocation on caches," in *25th Intl. Symp. on Microarchitecture (MICRO)*, 1992, pp. 222–225.

[18] F. Mueller, "Compiler support for software-based cache partitioning," in *Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 1995, pp. 125–133.

[19] P. Petoumenos, G. Keramidas, H. Zeffer, S. Kaxiras, and E. Hagersten, "Modeling cache sharing on chip multiprocessor architectures," in *Intl. Symp. on Workload Characterization (IISWC)*, 2006, pp. 160–171.

[20] T. Piquet, O. Rochecouste, and A. Seznec, "Exploiting single-usage for effective memory management," in *Asia-Pacific Computer Systems Architecture Conference*, 2007, pp. 90–101.

[21] L. R. Prabhat Jain, Srini Devadas, "Controlling cache pollution in prefetching with software-assisted cache replacement," MIT, Tech. Rep. CSG-462, 2001.

[22] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *Intl. Symp. on Comp. Arch. (ISCA)*, 2007, pp. 381–391.

[23] T. Sherwood, B. Calder, and J. Emer, "Reducing cache misses using hardware and software page placement," in *International Conference on Supercomputing (ICS)*, 1999, pp. 155–164.

[24] R. L. Sites and A. Agarwal, "Multiprocessor cache analysis using ATUM," in *International Symposium on Computer Architecture (ISCA)*, 1988, pp. 186–195.

[25] G. E. Suh, L. Rudolph, and S. Devadas, "Dynamic partitioning of shared cache memory," *Journal of Supercomputing*, vol. 28, no. 1, pp. 7–26, 2004.

[26] D. Tam, R. Azimi, L. Soares, and M. Stumm, "Managing shared L2 caches on multicore systems in software," in *Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA)*, 2007.

[27] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun, "A modified approach to data cache management," in *28th Intl. Symp. on Microarchitecture (MICRO)*, 1995, pp. 93–103.

[28] X. Vera, B. Lisper, and J. Xue, "Data caches in multitasking hard real-time systems," in *24th IEEE International Real-Time Systems Symposium (RTSS)*, 2003, pp. 154–165.

[29] A. Wolfe, "Software-based cache partitioning for real-time applications," *Journal of Computer and Software Engineering*, vol. 2, no. 3, pp. 315–327, 1994.

[30] W. A. Wong and J.-L. Baer, "Modified LRU policies for improving second-level cache behavior," in *6th Intl. Symp. on High-Performance Comp. Arch. (HPCA)*, 2000, pp. 49–60.

[31] Y. Wu, R. Rakvic, L.-L. Chen, C.-C. Miao, G. Chrysos, and J. Fang, "Compiler managed micro-cache bypassing for high performance EPIC processors," in *Intl. Symp. on Microarchitecture (MICRO)*, 2002, pp. 134–145.

[32] J. Zebchuk, E. Safi, and A. Moshovos, "A framework for coarse-grain optimizations in the on-chip memory hierarchy," in *40th Intl. Symp. on Microarchitecture (MICRO)*, 2007, pp. 314–327.