appeared in Proc. International Conference on Architectural Support for Programming Languages & Operating Systems (ASPLOS), March 2009, pp. 121-132.

RapidMRC: Approximating L2 Miss Rate Curves on Commodity Systems for Online Optimizations [*]

David K. Tam Reza Azimi Livio B. Soares Michael Stumm

University of Toronto, Dept. of Electrical & Computer Engineering, Toronto, Canada

{tamda, azimi, livio, stumm}@eecg.toronto.edu

Abstract

Miss rate curves (MRCs) are useful in a number of contexts. In our research, online L2 cache MRCs enable us to dynamically identify optimal cache sizes when cache-partitioning a shared-cache multicore processor. Obtaining L2 MRCs has generally been assumed to be expensive when done in software and consequently, their usage for online optimizations has been limited. To address these problems and opportunities, we have developed a low-overhead software technique to obtain L2 MRCs online on current processors, exploiting features available in their performance monitoring units so that no changes to the application source code or binaries are required. Our technique, called RapidMRC, requires a single probing period of roughly 221 million processor cycles (147 ms), and subsequently 124 million cycles (83 ms) to process the data. We demonstrate its accuracy by comparing the obtained MRCs to the actual L2 MRCs of 30 applications taken from SPECcpu2006, SPECcpu2000, and SPECjbb2000. We show that RapidMRC can be applied to sizing cache partitions, helping to achieve performance improvements of up to 27%.

Categories and Subject Descriptors C.4 [*Computer Systems Organization*]: Performance of Systems—measurement techniques, modeling techniques; I.6.4 [*Simulation and Modeling*]: Model Validation and Analysis; D.4.8 [*Operating Systems*]: Performance—measurements, modeling and prediction

General Terms Experimentation, Measurement, Performance, Management

[*] This work was supported by the Director, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

1. Introduction

The ever-increasing speed disparity between processors and disks in computer systems is mitigated, in part, by the memory hierarchy. Even though the size of processor caches and memory is constantly increasing, they will continue to be critical resources that need to be managed well if the memory hierarchy is to reach its performance potential. More recently there has also been an increased interest in managing this hierarchy with the objective of reducing energy consumption. In either case, the operating system is typically responsible for the management of the memory hierarchy.

Numerous researchers have proposed using Miss Rate Curves (MRCs) for the purpose of improving management of the memory hierarchy, including file buffer management [18, 28, 48], page management [3, 45, 47], and L2 cache management [29, 40, 41]. MRCs capture the miss rate as a function of memory size for a process or a workload consisting of a set of processes (e.g., virtual machine) at a particular point in time. MRCs thus identify the memory needs of processes.

MRCs can be obtained *offline* in a relatively straightforward way by running the target application or workload multiple times, each time using a different memory size or cache size. While *online* capturing of MRCs for file systems is also relatively easy, say using ghost buffers [28], the *online* capture of MRCs for main memory or for caches is significantly more challenging without hardware support.

In this paper, we target L2 caches and introduce **RapidMRC**, a software-based, *online* technique that approximates L2 MRCs on commodity systems with low overhead for the purpose of managing L2 caches. The challenge in this new context, compared to main memory and disks, is that (1) the potential gains from reducing cache misses is relatively small compared to the gains from reducing main memory misses, and (2) the cost of tracking misses and generating MRCs is large compared to the cost of the miss event. A main memory miss allows plenty of time for the processor to perform tracking and calculations before receiving the data from disk. In contrast, the act of recording the cache miss can be several times more expensive than the cache miss itself.

We make three contributions in this paper. First, we present RapidMRC, a software-based online method to characterize the cache requirements of processes on a commodity processor by generating L2 MRCs in a low overhead, low latency manner. We demonstrate how to exploit the available architectural support in modern commodity processors, in the form of performance monitoring units (PMUs), to extract information and process it in order to enable online optimizations at various software levels, such as at the operating system, virtual machine monitor, and programming language run-time system level. We compare the accuracy of online RapidMRC to the real MRCs for 30 applications taken from standard benchmarks.

As a second contribution, we examine the multitude of factors in modern processors that can impact the accuracy of the calculated MRC. We examine existing architectural support as well as barriers in developing the RapidMRC technique.

As a third contribution, we show how RapidMRC can be applied to L2 cache partitioning by determining the best partition size to allocate to each application running in a co-scheduled manner on a shared-cache multicore processor. Shared on-chip L2 caches are a popular cache organization for multicore chips mainly due to the advantage of higher aggregate utilization when compared to private cache organizations. However, a shared cache can cause performance problems because it lacks isolation properties — concurrent processes compete for space in the shared cache and may interfere with each other by evicting each others cache lines. Consequently, researchers have proposed partitioning this shared cache [12, 16, 29, 30, 41, 42]. RapidMRC can be used online to help determine the best partition sizes.

In addition to cache partitioning, RapidMRC could be used in several other ways online: (i) to reduce energy consumption by reducing the cache to the minimal size at which the running process/workload can still run effectively [1, 5, 26]; (ii) to manage bus bandwidth contention to main memory due to cache misses [2, 17]; (iii) to guide co-scheduling algorithms in selecting processes that fit within the available L2 cache space [14, 32, 36, 43]; (iv) to predict the global MRC of N applications in an uncontrolled cache-sharing configuration [8, 11]; and (v) to identify applications with low cache reuse so that they can all be placed into a single, shared *pollute buffer* cache [37].

2. Background and Related Work

In this section, we review miss rate curves, describe the specific requirements for generating L2 cache miss rate curves, and describe related work on L2 cache partitioning.

2.1 Miss Rate Curves

The Miss Rate Curve (MRC) of a memory access sequence identifies the miss rate as a function of the amount of memory allocated to the sequence at a particular point in time. The key advantage of the MRC model over the traditional working-set model is that the MRC model presents an entire trade-off spectrum between allocated memory size and resulting miss rate. In contrast, the working-set model only indicates the amount of memory that a process must have for acceptable performance, and it does not identify how performance is affected if the amount of memory allocated is less than its working-set size.

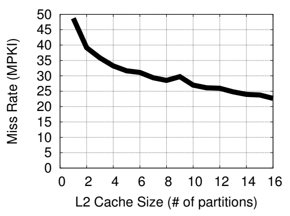


Figure 1. Offline L2 MRC of `mcf`.

In our study, we focus on generating MRCs for on-chip shared L2 caches. Figure 1 shows an example of the offline L2 MRC of `mcf`, from the SPECcpu2000 benchmark suite, where the L2 cache is divided into 16 partitions, using software-based cache-partitioning [42], and the measured L2 miss rate is plotted as a function of the number of partitions allocated to the application when it is run. The general trend in nearly all MRCs is that as more space is allocated, the miss rate decreases.

In addition to the specific patterns in memory access sequences, the MRC is affected by the replacement policy of the cache. That is, the MRC of a Least Recently Used (LRU) policy may be significantly different from that of a Most Recently Used (MRU) policy for the same memory access sequence. Throughout this paper we assume that the default replacement policy is Least Recently Used (LRU) since it is the most commonly used replacement policy for processor caches.

A common method to calculate MRC is the *Stack Algorithm*, originally developed by Mattson *et al.* [25] and independently by Kim *et al.* [20], both intended for offline analysis of main memory access patterns at page-level granularity. In this algorithm, an *LRU stack* is maintained, consisting of memory addresses generated from the sequence of memory accesses so that the top element is the most recently accessed memory address and the bottom of the stack is the least recently accessed memory address. On each access, the *distance* of the current location of the accessed address from the top of the LRU stack (i.e., *Stack Distance*) is determined before moving the address to the top of the stack. The stack distance of the memory access can be used to speculate whether the access would result in a miss or a hit given a certain memory size. That is, for any memory size larger than the stack distance, the access would be a regarded as a hit since it is expected that the memory element has, at the time of the access, not yet been replaced by the LRU algorithm. On the other hand, for any memory size smaller than the stack distance, the memory access would be regarded as a miss. In order to generate the MRC, a histogram, $Hist$, is calculated where $Hist(dist)$ shows the total number of memory accesses with a stack distance of $dist$. Therefore, the number of misses for memory of size $size$, $Miss(size)$ can be calculated as follows:

$$Miss(size) = \sum_{dist=size+1}^{\infty} Hist(dist)$$

Miss rate curves have been used to manage main memory pages [3, 45, 47]. In this context, misses to the page are trapped into the kernel and are thus easily seen, and any page replacement policy can be used. Miss rate curves have also been used to manage disk buffer caches [18, 28, 44, 48] and database application buffer caches [38]. In this paper, we apply miss rate curves to L2 caches, obtaining L2 cache access traces with the help of hardware performance counters.

In our application of RapidMRC to sizing cache partitions, we must compare current miss rates across memory access sequences of concurrently executing applications. For this purpose, we normalize the value of $Miss(size)$ over a fixed probing period, using the number of Misses Per Kilo Instructions (MPKI):

$$MPKI(size) = 1000 \times \frac{Miss(size)}{CPUInstructions}$$

where $CPUInstructions$ is the length of the probing period as measured by instructions executed.

2.2 L2 MRC Generation

A basic requirement for building a precise LRU stack for computing an MRC is to have an accurate trace of the application's memory accesses, which can be obtained in several ways. One way to capture memory traces is to run the application in a simulation environment, where the simulator is able to monitor the execution of individual instructions of the application. This method is extensively used in computer systems research for offline analysis of memory access patterns. However, simulation is not suitable for online use because of its high constant overhead.

Another method of capturing memory traces is to *instrument* all memory access instructions of the application so that the accessed addresses are recorded into a *trace log*. Instrumentation tools such as Pin [24], DynInst [10], DynamoRIO [9], and JIFL [27] can be used for this method. While being simple and straightforward to implement, this approach is too expensive for online use when all memory accesses are instrumented. It substantially slows down the execution of applications (in some cases by a factor of 10 [33]) because of the additional instructions that must be executed and poorer instruction cache performance due to the increased instruction footprint.

One way to reduce the overhead is to dynamically enable or disable instrumentation in an on-demand basis using a dynamic code modification system such as DynamoRIO. However, there still exists a fixed overhead with a such a system, as Zhao *et al.* [46] have reported an average minimum runtime overhead of 13% for DynamoRIO when instrumentation (for purposes such as memory access tracing) is disabled. In contrast, our approach has no overhead when memory access tracing is disabled. In addition, a fundamental advantage of our approach over a dynamic instrumentation approach is that we can capture only the hardware events that are of interest, which is a much smaller fraction of all hardware events that occur, potentially leading to lower costs and simpler designs.

Our method for collecting memory access traces uses features available in modern PMUs in a way that requires no changes to applications and has sufficiently low overhead so as to be useful for online purposes. Our software-based solution is in contrast to hardware-based solutions that have been proposed in the past. For example, Patt and Qureshi [29] and Suh *et al.* [41] propose hardware additions to future processors to obtain L2 MRCs online. Their general strategy is to monitor several cache sets in an N-way set-associative cache by attaching access counters to each LRU position within a set. Within each set, these access counters serve the role of tracking the stack distance in Mattson's algorithm.

Several researchers have presented various analytical models to calculate L2 cache miss rates based on memory access traces, such as [6, 15, 34]. These techniques were mainly targeted for offline analysis of memory access traces obtained using a simulator. For Solihin *et al.*'s model [15] to be used online, additional hardware support would be required. Shen *et al.*'s model [34] does not require additional hardware support to be used online but they have yet to show its use in an online environment. Berg and Hagersten's model [6] can also be used online without additional hardware support, and they have subsequently shown how to obtain the reuse distance using watchpoints on commodity processors with an average overhead of 39 % throughout the entire execution of an application [7]. In contrast, our work takes an approach opposite to Berg and Hagersten's sampling-based approach over the entire execution of the application because we capture every access for a short window of accesses for online optimization purposes.

2.3 L2 Cache Partitioning

Many researchers have shown that shared L2 caches can suffer from performance problems, such as [11, 14, 36]. In a multiprogrammed environment, proper provisioning of the shared L2 cache among multiple cores or applications by partitioning the cache appropriately, can result in performance improvements over uncontrolled use of this shared resource. Many hardware-based cache partitioning mechanisms have been proposed [16, 23, 29, 30, 39, 41], but they are based on additional hardware components that future processors may or may not implement. Several software-based cache partitioning mechanisms have also been proposed [12, 21, 22, 42]. These software-based mechanisms can be implemented in the operating system or virtual machine monitor on existing processors.

In general, hardware proposals have the inherent advantage of lower runtime overheads and better accuracy than software implementations, but it remains to be seen if and/or when these proposals will appear in real processors, thus giving software implementations the practical advantage of being deployable today.

As for software-based solutions, an important missing piece of the puzzle in software-based mechanisms is how to determine the *optimal* cache partition size to allocate to a process or workload in an online manner with low overhead. For software-based mechanisms, only trial and error techniques have been employed so far, although they typically use a form of binary search to reduce the number of trials [19, 22]. With these approaches, determining the best sizes for more than 2 applications or cores is non-scalable because the number of possible size combinations grows exponentially with the number of applications or cores. Using dynamically obtained MRCs, on the other hand, we can eliminate this trial and error approach. A convenient property of MRCs is that they are unaffected by, and independent of, the currently configured cache partition size. This is possible because MRCs are generated from a trace of load/store memory operations, regardless of whether these accesses result in a hit or a miss in the L2 cache. In Section 4, we apply RapidMRC to provide a practical analytical approach to determining the optimal cache partition size, capable of running on commodity processors.

3. RapidMRC

In this section, we describe the design and implementation of RapidMRC. We describe how we collect memory access traces and how we generate MRCs from the collected traces. We also discuss important details about the implementation of RapidMRC on the IBM POWER5 processor.

3.1 Collecting Memory Access Traces

Our method for collecting memory access traces is based on using data sampling features available in some of the performance monitoring units (PMUs) of today's processors. The key advantage of this method is that it is completely transparent to the application, requiring no code instrumentation, since the process of recording data addresses is done entirely in hardware. The basic PMU feature required is the capability of recording the data address of memory accesses to a data address register (DAR), or to a designated memory buffer. Systems software can then be notified with an exception when the DAR is updated or when the designated memory buffer overflows.

In theory, we need to capture *all* data accesses of the target process, at least for a short period of time. However, given the fact that roughly one in three instructions is either a load or store instruction, recording all memory accesses is expensive. As a performance optimization, given that we wish to compute MRCs for the L2 cache, we record only the data accesses to the L2 cache. In our case study environment, the events that access the L2 cache are (1) L1 instruction and data cache misses, (2) L1 data write-through accesses, and (3) hardware prefetches. Due to limitations in our hardware, we only track L1 data cache misses, which are generally much less frequent that L1 cache hits. Nevertheless, we show in our results that the accuracy of RapidMRC remains high.

It is important to ensure that the time interval over which memory accesses are traced is long enough to identify the patterns in the *reuse distance* of individual cache lines. The size of the access trace must be several times as large as the number of the L2 cache lines so that the reuse distance of each cache line can be sampled several times.

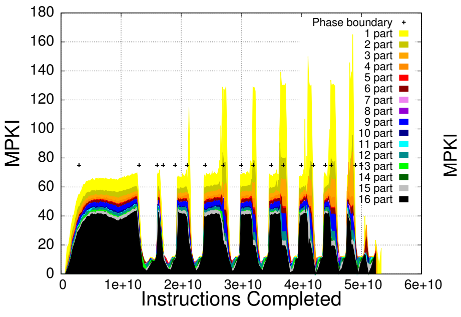
3.1.1 PowerPC-Specific Issues in Gathering Traces

The PMU in the IBM POWER5 processor, a member of the PowerPC family, can perform data sampling *continuously*, where the *Sampled Data Address Register* (SDAR) is continuously updated by the PMU as memory instructions with operands that match a selection criterion arrive in the pipeline. Systems software can sample SDAR values by periodically reading its value, which identifies the data address specified by the last memory operation that matched the given selection criterion. With this method, *all* address operands of memory instructions have a fairly equal chance of being captured.

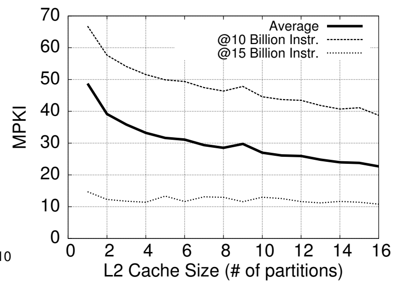
Although other processors, such as the Intel Itanium 2, AMD Opteron, and IBM POWER4, can perform data address sampling, they cannot do so continuously in order to capture a trace. As for Intel IA-32 processors, we have experimented with the Precise Event-Based Sampling (PEBS) mechanism and found that the lack of *data address* information made address collection challenging.

We exploit the current implementation of the PMU in the POWER5, where one can set the selection criterion for updating the SDAR to be a miss in the L1 data cache, thus capturing accesses to the L2 cache. We then use a separate PMU counter to count the number of L1 data cache misses, and assign an overflow threshold of *one* so that an interrupt is raised on every L1 data cache miss. Raising an exception on each L1 miss is costly considering each exception flushes the processor pipeline and switches the execution context from user-space to kernel-space and back. One can envision a hardware PMU that automatically records the data address trace into a small pre-designated buffer, either within the processor core or in main memory, raising an exception only when the buffer overflows so that the cost of overflow exception is amortized over a larger number of data samples. To the best of our knowledge, the PMU in none of today's mainstream processors provide such a feature for data samples. To deal with this lack of hardware support, we limit the period of time over which addresses are gathered.

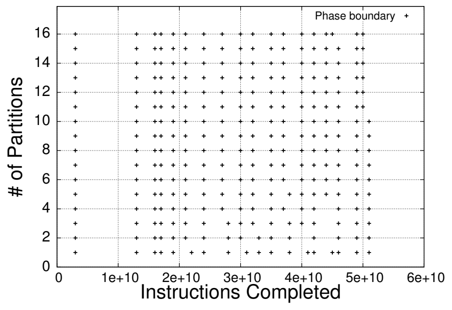
Intricacies of the POWER5 introduce two sources of inaccuracy in our method. The first is the fact that in a superscalar processor, there may be more than one L1 data cache miss-



a. Phases in terms of L2 miss rate.



b. MRCs at various execution points.



c. Detected phase boundaries (see Section 5.2.2).

Figure 2. Phase transitions in `mcf` and their impact on the MRC, measured on an IBM POWER5.

inflicting load-store in flight, due to multiple instruction issue and out-of-order execution. With two neighboring L1 data cache misses being serviced in parallel by the two load-store units of the processor, it is possible that one of them does not cause the SDAR to be updated. When the first L1 data cache miss raises an exception, the entire pipeline is flushed, which includes the second in-flight memory instruction. However, since the memory access request for the second access has already been sent to the lower levels of the memory hierarchy, when the memory instruction is re-issued after the exception is handled, it may not miss in the L1 data cache anymore, and therefore, the SDAR would not be updated by the second memory access. Fortunately, our results in Section 5.2 show that the collected trace is sufficiently accurate for the purpose of computing MRCs. For problematic applications, we show the impact of disabling multiple instruction issue and out-of-order execution.

The second source of inaccuracy is due to hardware prefetch requests to the L1 data cache that do not cause the SDAR register to be updated with the address of the prefetch target. As a result, a stale SDAR value is recorded into the access trace, leading to trace segments containing consecutive entries all with the same value. We handle this problem by converting these repetitions into a series of ascending cache lines accesses, thus emulating the value that should have been recorded into the SDAR. Fortunately, our results Section 5.2 show that the corrected trace is sufficiently accurate for the purpose of computing MRCs.

3.2 L2 MRC Generation

In order to generate MRCs, we record the L2 accesses due to L1 data cache misses by appending them to an *access trace log* located in main memory. This can be done either automatically by the hardware, or through an exception-handler in software. We then feed the access trace into an LRU stack simulator which builds the LRU stack and generates the MRC using the Mattson stack algorithm [25]. In our targeted use of RapidMRC, since the L2 MRCs are used to size the partitions of the L2, we limit the size of the LRU stack to the size of the L2 cache. The LRU stack simulator implementation is based on our design described in [3], which uses the *range list* optimization proposed by Kim *et al.* [20].

As we will show in Section 5.2, cache prefetching and missed events on address trace collection have the effect of causing the calculated MRC to be vertically offset from the real MRC. To adjust for this, we vertically shift (transpose) the calculated MRC so that it matches at least one point of the real MRC. Since any point can be used, in practice, this point can be the currently configured cache partition size, since its miss rate can be easily obtained from the processor PMU.

MRCs predict the miss rate for a fully associative cache. While today's L2 caches are not fully associative, they usually have high associativity (e.g., 16-way). This makes the behavior of the cache to be similar to that of a fully associative cache. We have found this approximation to be adequate for computing MRCs, as we will show in Section 5.2.

Many applications go through several *phases* in their execution. In each phase, the performance of an application, often characterized in terms of key performance metrics such as Instruction Per Cycle (IPC), is fairly stable. However, the performance characteristics of two phases of a single application may be substantially different. As a result, we need to take into account the potential changes in an application's MRC, caused by phase transitions. While the number of unique phases in an application is often quite small, there may be many transitions back and forth between these phases.

Figure 2 shows the impact of phase transitions on the measured MRC of `mcf`. These measurements were taken by using our software-based cache partitioning mechanism [42], running the application 16 times, each time with a different L2 cache size, and using the PMUs to measure the cache miss rate [4]. Figure 2a shows how `mcf` alternates between two phases repeatedly. This graph also indicates how the L2 miss rate diminishes as more L2 partitions (parts) are allocated.

The graph in Figure 2b shows the measured MRCs for these two phases, compared to the average MRC over the entire execution of the application. The MRCs for the two phases imply substantially different L2 cache requirements within a single application.

4. Example Usage of RapidMRC: Cache Partitioning

In this section we describe how RapidMRC can be applied to the cache partitioning problem, especially in the context of multicore chips. We utilize our software-based cache partitioning mechanism, described in [42], to divide the L2 cache into a number of *colors*. Each application is allocated a number of colors and as a result can only populate a fraction of the cache. A key issue is how to determine the number of colors to allocate to each application.

In Section 5.3, we show the effectiveness of using RapidMRC in determining partition sizes to allocate to applications with the overall goal of improving performance. However, the operating system could pursue other performance objectives, such as providing quality-of-service.

For deciding optimal performance partitioning between two co-scheduled applications, we use a simple function which minimizes overall misses. Given two miss rate curves MRC_a and MRC_b, for two processes, we apply:

$$\min_{x \in [1, C-1]} \left[(MRC_a(x) + MRC_b(C-x) \right]$$

where C is the total number of colors into which the cache can be divided. While, for typical C values (e.g., 16), this utility function is sufficiently lightweight to be re-computed dynamically (online) for different phases of applications, in our current implementation we compute this utility function for any pair of applications statically (offline).

Our simple method for obtaining the optimal partitioning is effective only when two applications run simultaneously. In configurations where there are more than two applications at a time, one can use more sophisticated methods such as the approximation presented by Qureshi *et al.* [29] to address the NP-Hard complexity of the problem [31].

5. Experimental Results

5.1 Setup

The experimental results we present here were obtained on an IBM POWER5 system, as specified in Table 1, and on a similarly configured POWER5+ system for some experiments. Each POWER5 chip contains an L2 cache that is shared between 2 cores. Each core contains a private L1 data cache and L1 instruction cache. Connected to each chip is an off-chip L3 victim cache, which is also shared between the 2 cores.

RapidMRC was implemented in the Linux Operating System, kernel version 2.6.15 on POWER5 and 2.6.24 on POWER5+, and evaluated with 19 applications from SPECcpu2000, 10 applications from SPECcpu2006, and SPECjbb2000. The IBM J2SE 5.0 JVM was used to run SPECjbb2000 (1 warehouse configuration). For SPECcpu2000 and SPECcpu2006, the applications were run using the standard *reference* input set. Thread migration between cores was disabled in the operating system to provide a more controlled execution environment.

Item	Specification
# of Cores per Chip	2
Frequency	1.5 GHz
L1 ICache (Private)	64 KB, 128-byte lines, 2-way associative
L1 DCache (Private)	32 KB, 128-byte lines, 4-way associative
L2 Cache (Shared)	1.875 MB, 128-byte lines, 10-way associative
L3 Victim Cache	36 MB, 256-byte lines, 12-way associative
RAM	8 GB (4 GB on POWER5+)

Table 1. IBM POWER5 specifications.

5.2 Results

We begin by evaluating the accuracy of RapidMRC by comparing it to the real MRC values, and then analyzing the runtime overhead. Finally, we briefly present results from applying RapidMRC to sizing cache partitions.

5.2.1 MRC Accuracy

There are two components to MRC accuracy: curve shape and vertical offset (v-offset). Matching the shape is the challenging component, whereas matching the v-offset is relatively easy, as described in Section 3.2. Factors influencing the v-offset will be examined in later subsections.

The size of the access trace log was configured to 160k entries. For each application, the percentage of the trace log used for warming up the LRU stack is shown in Table 2, column f. The number of entries used for warmup was either determined automatically by the MRC calculation engine or it was statically set to 80k entries (one half of the trace log). For automatic warmup determination, we waited until all entries in the LRU stack were occupied before switching out of warm up mode. For some applications, the trace log was not long enough to warm up the LRU stack under this criteria. These applications had very small working set sizes and seldom spilled to memory, as evidenced by the LRU Stack Hit Rate shown in Table 2, column g. Therefore, the statically set warmup length was, in fact, adequate for them.

Figure 3 illustrates the online calculated MRC compared to the real MRC for each of the 30 applications. To obtain the real MRCs, we used an exhaustive offline method combined with our software-based cache partitioning mechanism described in [42]. For each of the possible 16 cache sizes of our L2 cache, the application was executed in its entirety while using the processor PMU to measure the L2 cache miss rate every 1 billion processor cycles. Both real and calculated MRCs are taken from a brief slice of execution, at the 10-billion completed instruction mark. For the real MRCs, the length of the slice is 1-billion completed instructions, whereas the slice length of the calculated MRCs varies and is shown in Table 2, column c, averaging to 54-million instructions. To verify that the offline real MRC generated from the 1-billion instruction slice of a phase was indeed representative of the entire phase, we also experimented with thicker, 10-billion instruction slices and obtained the same results. For each application, v-offset matching was done, as described in Section 3.2, using the 8-color point of the real

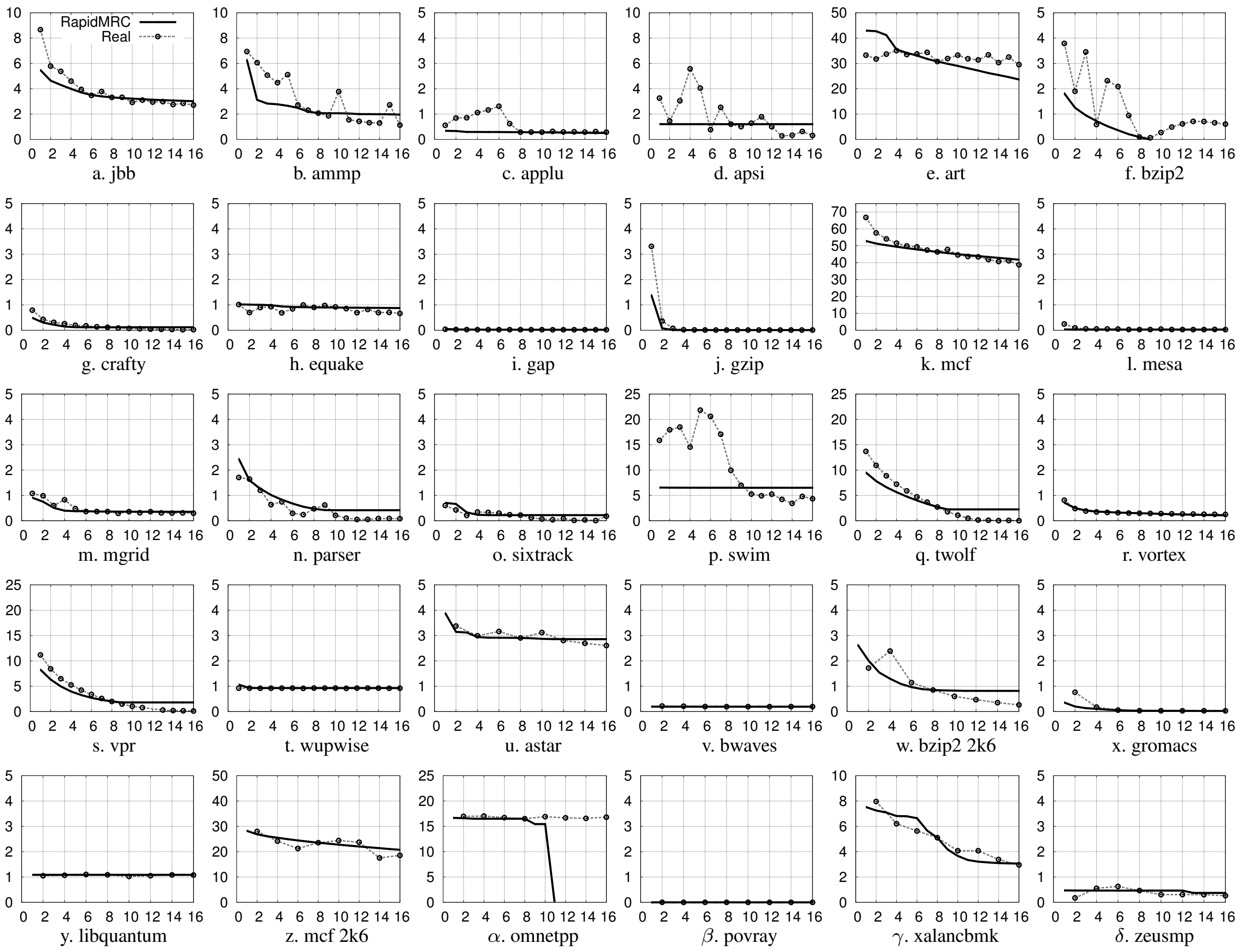


Figure 3. Online RapidMRC vs real MRCs. x-axis = allocated L2 cache partition size (# of colors), y-axis = resulting L2 cache miss rate (MPKI).

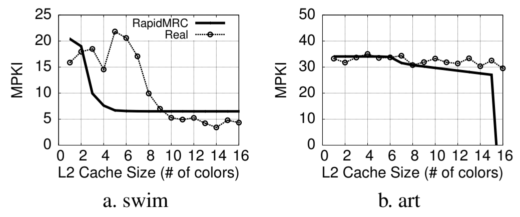


Figure 4. Improved RapidMRC.

MRC. This shift amount was uniformly applied to all other points of the calculated MRC, resulting in a uniform vertical shift without any distortion to its shape. Table 2, column h shows the amount of vertical shifting applied to each application.[1]

For 25 out of the 30 applications, the calculated MRCs match closely to the real MRCs. The general trend is that RapidMRC is capable of tracking a variety of shapes from real MRCs. However, there are five problematic applications: *swim*, *art*, *apsi*, *omnetpp*, and *ammp*. Using a longer, 1600k-entry trace log improved *swim*, as shown in Figure 4a, but it remains problematic. Some improvements to *art* were achieved on the POWER5+ configured with hardware data prefetching disabled, in-order execution, and single instruction issue, as shown in Figure 4b. However, a problem remains with the 15-color point. In general, the sources of inaccuracy in these five applications are subject to further research, since they are not caused by the factors examined in the subsequent subsections.

For a quantitative evaluation of MRC similarity, we propose using the metric of average MPKI distance between the each real and corresponding calculated point, over the 16 possible cache sizes. The formula is shown below, and the calculated values are shown in Table 2, column i.

$$Distance = \frac{1}{16}\sum_{i=1}^{16} |MPKI_{real}(i) - MPKI_{calc}(i)|$$

[1] The average is calculated using absolute values.

Column	(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)	(j)
Workload	Trace Logging Time (x10^6cycles)	MRC Calculation Time (x10^6cycles)	Application Instructions (x10^6)	Average Phase Length instrs:cycles (x10^9)	Prefetch Conversion (% Log)	% Log Used for Warmup	LRU Stack Hit Rate	Vertical Shift (MPKI)	Distance (MPKI) 160k Log	Distance (MPKI) 1600k Log
jbb	189	86	17	60 : 101	15 %	42 %	80 %	1.3	0.51	0.51
ammp	192	72	22	46 : 65	14 %	83 %	95 %	1.6	1.02	1.02
applu	201	83	27	400 : 483	7 %	29 %	70 %	-1.6	0.28	0.26
apsi	462	59	351	5 : 6	39 %	60 %	88 %	1.1	1.09	1.09
art	177	146	6	100 : 246	18 %	20 %	76 %	17.5	4.54	4.06
bzip2	200	81	26	16 : 17	4 %	81 %	97 %	-0.8	1.02	0.94
crafty	191	48	24	250 : 249	5 %	50 %	98 %	0.0	0.08	0.07
equake	252	128	57	120 : 150	42 %	12 %	48 %	-0.6	0.12	0.12
gap	599	98	301	175 : 224	76 %	27 %	65 %	-0.2	0.00	0.00
gzip	191	51	21	325 : 446	30 %	50 %	99 %	-0.1	0.14	0.22
mcf	185	155	5	3 : 11	2 %	13 %	50 %	25.0	2.57	2.64
mesa	284	47	91	275 : 356	7 %	50 %	98 %	0.0	0.03	0.03
mgrid	192	69	30	550 : 509	54 %	38 %	72 %	-1.2	0.08	0.07
parser	203	59	24	104 : 144	5 %	50 %	98 %	0.3	0.28	0.21
sixtrack	207	48	36	500 : 474	8 %	50 %	99 %	0.2	0.13	0.12
swim	204	113	20	11 : 28	62 %	15 %	51 %	2.1	6.12	4.88
twolf	191	77	16	300 : 518	4 %	50 %	100 %	2.2	1.72	1.71
vortex	251	74	97	450 : 400	11 %	54 %	88 %	0.0	0.02	0.03
vpr	189	69	16	16 : 25	5 %	50 %	99 %	1.7	1.03	1.01
wupwise	291	137	129	310 : 314	48 %	15 %	37 %	0.1	0.01	0.01
astar	185	158	16	152 : 355	3 %	30 %	69 %	-0.3	0.20	0.19
bwaves	150	62	14	15,000 : 16,088	0 %	50 %	91 %	-0.8	0.00	0.00
bzip2 2k6	161	81	27	42 : 38	11 %	50 %	92 %	0.4	0.43	0.47
gromacs	243	90	71	5,000 : 7,230	11 %	62 %	89 %	-0.2	0.06	0.02
libquantum	153	404	11	2,250 : 1,753	96 %	9 %	0 %	-14.0	0.02	0.02
mcf 2k6	161	282	6	25 : 104	2 %	20 %	53 %	30.1	1.95	1.96
omnetpp	167	323	7	650 : 1,704	0 %	24 %	86 %	-15.8	6.57	3.82
povray	161	324	24	14,000 : 14,362	6 %	50 %	100 %	0.0	0.00	0.00
xalancbmk	176	177	21	324 : 551	4 %	66 %	88 %	2.1	0.53	0.53
zeusmp	224	113	102	12,000 : 12,650	5 %	47 %	83 %	0.1	0.13	0.15
Average	221	124	54	1,782 : 1,987	20 %	42 %	79 %	3.9	1.02	0.87

Table 2. RapidMRC statistics.

5.2.2 Overheads

Table 2, columns a and b show the overheads involved in calculating the MRC. The trace logging time measures the wall clock time required to capture 160k entries into the trace log during application execution. On average, it takes 221 million cycles to obtain the trace log, which is 147 ms on our POWER5 system. During this trace log period, the application is still making progress, although much slower, at 24% of the original IPC on average. The MRC calculation time is the time required to process the trace log and generate the calculated curve. This time was acquired assuming that the application is not running during the calculation. The average time required is 124 million cycles (83 ms). Given the two columns of trace logging time and MRC calculation time, we can see that the average time required to perform online MRC calculation is 345 million cycles (230 ms).

The actual runtime overhead incurred by RapidMRC depends on the frequency of phase transitions, which require recomputation of the MRC. Due to limitations in our current implementation of RapidMRC, we currently do not automatically track program phase transitions and re-trigger RapidMRC. However, we have done post-mortem analysis on the collected PMU data to calculate the average length of application phases. Column d in Table 2 indicates the average phase length of each application, both in terms of the number of instructions and the number of processor cycles. In all but two cases (`apsi` and `mcf`) the total runtime overhead of trace logging and online MRC calculation is below 2%. In many cases, due to very long phases, the overhead is negligible.

To locate phase transitions in our collected PMU data, we used the following simple heuristic. We used changes in the L2 cache miss rate as the indicator of phase transitions because it directly reflects the changes in the application's cache usage, rather than IPC as suggested by Sherwood *et al.* [35]. The L2 cache miss rate of an application can be monitored online with negligible overhead. In order to identify *significant* changes in the miss rate, we used the following simple heuristic. We divided the collected PMU data into intervals containing a fixed number of instructions. At the end of each interval, we compared the miss rate of the current interval against the average miss rate of the past w intervals, and a phase transition was declared if the two miss rates differed more than a specified threshold. In addition, since phase transitions can span several intervals, this threshold is also used as the minimum/maximum miss rate difference threshold to signify the beginning/end of a lengthy phase transition.

The numbers shown in Table 2, column d were obtained using the following parameter values for the above heuristic: (1) the L2 miss rate of the 8-color cache size configuration, (2) an interval length of 1 billion instructions, (3) a history size of $w = 3$, (4) a miss rate difference threshold of 3 MPKI, and (5) a start/end of phase transition threshold of 50%.

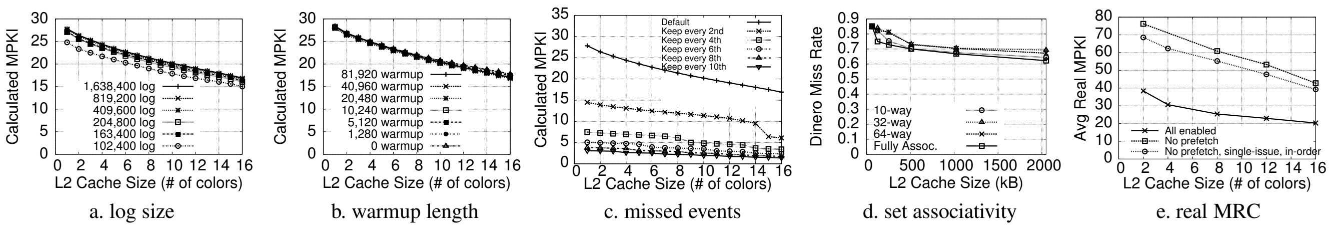


Figure 5. Impact of various factors on the calculated and real MRC of `mcf`.

An indication of the accuracy of this heuristic can be seen in the phase boundary markings shown for *mcf* in Figure 2a. These boundary locations coincide with the actual phase transitions visually depicted in the graph.

Figure 2c provides an example to demonstrate that these boundary locations are insensitive to an application's currently configured L2 cache size. The graph indicates the phase boundaries of `mcf` detected by monitoring the L2 MPKI for each possible L2 cache size. It can be clearly seen that the vast majority of phase transitions are detected at the same points of execution for all L2 cache sizes.

Figure 2c also shows that changes to the MRC as a whole, can be detected by monitoring changes to just a single point of the MRC. If a single point on the MRC changes significantly, then all points of the MRC change significantly too. Conversely, if a single point on the MRC does not change, then all points do not change significantly either.

5.2.3 Impact of Trace Log Size

We chose a trace log that was long enough so that the bottom stack position, the furthest from the top of the LRU stack, had a chance of being incremented several times. Since our LRU stack is 15,360 in length, in the worst-case cache-hit scenario, it would require a trace log of at least 15,360+1 in length in order for the first trace log entry to end up at the bottom of the stack and then be accessed on the 15,361st access, registering a stack hit. To be conservative, we chose a trace log of approximately 10 times the length of the LRU stack, resulting in our trace log length of 160k entries.

Although the trace log itself can pollute the L2 cache, this impact has been automatically incorporated into the RapidMRC curves of Figure 3.

In addition to the 160k-entry trace log size, we also tried using a 1600k-entry trace log size. Although *swim* benefited greatly, as shown in Figure 4a, the other applications did not show benefits. Figure 5a shows how *mcf* is largely unaffected by the log size. The warmup period is 50% of the trace log size. For the remaining applications, we show the average MPKI distances for a 1600k-entry log in Table 2, column j.

5.2.4 Impact of Warmup Period

As with any structure that contains state information, the LRU stack requires a warmup period before it begins recording statistics. This warmup period prevents the stack distance counters from initially reporting wrong stack position hits, as well as false cold misses. For automatic warmup determination, we waited until all entries in the LRU stack were occupied before switching out of warm up mode. The impact of varying the warmup period for *mcf* is shown on Figure 5b. Similar trends were seen for the other applications and are not shown in this paper. From these results, we can see that our chosen criteria for warmup is adequate for MRC accuracy.

5.2.5 Impact of Missed Events

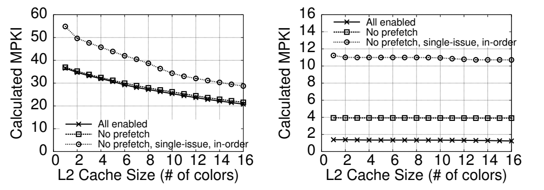
The POWER5 PMU does not guarantee that it will capture every single L1 data cache miss event. In this section, we examine the impact on the calculated MRC of losing more and more of these events. Since we are unable to obtain the number of lost events from the PMU, we examine the impact on the calculated MRC by artificially further dropping more and more entries from the trace log. By working forwards to capture the trend, we can extrapolate the trend backwards.

Figure 5c shows the impact on the calculated MRC of *mcf* as a larger and larger percentage of its trace log entries are ignored. These artificial degradations to the trace log are indicated by labels such as "keep every 4th", which simulates the impact of dropping 3 events and keeping the next event. The larger 1600k-entry trace log was used to ensure adequate trace log lengths. Similar trends were seen for the other applications and are not shown here.

From these results, we can see that the v-offset of the calculated MRC is affected. As the number of events missed increases, the MRC is shifted further down. There is also a potential impact on the MRC shape, affecting the smaller cache sizes. The precise magnitude of the shifting or shape distortion varies across applications and shows no predictable pattern. By extrapolating these trends backwards, we can conclude that missed events are a potential source of the v-offset mismatch between the real and calculated MRCs.

5.2.6 Impact of Set Associativity

To examine the impact of using a fully associative cache model compared against the 10-way set-associative cache used by the POWER5, we fed our trace log into the Dinero cache simulator [13]. We configured it to simulate only the POWER5 L2 cache. The associativity was varied from 10-way to full, and the impact on the miss rate was extracted. The results for *mcf* are shown in Figure 5d. Similar trends were seen for the other applications. The graph indicates that our



a. calculated MRC (`mcf`) b. calculated MRC (`equake`)

Figure 6. Impact of various factors on the calculated MRC.

fully associative cache model simplification does not have a material impact on miss rate.

5.2.7 Impact of Hardware Prefetching

Hardware prefetchers, located in the L1 data cache and the L2 cache can have an impact on the real MRC. For some applications, the prefetchers can pollute the L2 cache and lead to a higher miss rate than indicated from the memory access pattern or predicted by RapidMRC. On the other hand, for other applications, the prefetchers can be beneficial and help overcome the problems of smaller cache sizes. Figure 5e shows the impact on the real MRC of *mcf* from disabling the hardware prefetchers. Similar trends were seen for 8 other applications that we tried[2] indicating that the POWER5 hardware prefetchers are beneficial and help to reduce the miss rate, vertically shifting the real MRC downwards.

Hardware prefetchers can also have an impact on the captured trace log because prefetching is occurring during the capture period, thus also affecting the calculated MRC. On the POWER5, these prefetched addresses appear in the trace log as a series of consecutive repeated data address values, as described in Section 3.1.1, but they do not show the actual values. As described in Section 3.1.1, we converted these repetitions into consecutive adjacent cache line addresses. Table 2, column e shows the percentage of the trace log that required this conversion. In contrast, the POWER5+ omits this prefetch activity from the trace log. Therefore, both processors cannot provide adequate information to accurately model the prefetcher impact on the calculated MRC. In effect, this problem causes an increase in the number of missed events in our trace log, leading to the problems described in Section 5.2.5. To model prefetcher activity, we need to capture all L1 data cache accesses (both hits and misses). However, this approach incurs extremely high overhead which makes it impractical for online use.

In an attempt to determine the general impact of hardware prefetching on the calculated MRCs, we compare the calculated MRC from a trace log obtained with prefetching against a trace log obtained without prefetching. These experiments were run on the POWER5+. Figure 6a and Figure 6b show two examples of how these MRCs are affected for *mcf* and *equake*, respectively. The other applications show similar results. In general, the calculate MRCs are vertically shifted by various, currently unpredictable amounts, perhaps dependent on the application access pattern.

5.2.8 Impact of Multiple Instruction Issue & Out-of-Order Execution

Allowing multiple instructions to be issued and be in-flight in the processor pipeline may potentially lead to inaccuracies in RapidMRC, as described Section 3.1.1. Figure 6a and Figure 6b show the impact of a simplified processor mode (single-issue, in-order, no prefetching) on RapidMRC as calculated on the POWER5+ for *mcf* and *equake*. The processor executes the application in complex mode (multiple-issue, out-of-order, with prefetching) except during the trace collection period when it is placed into the simplified mode. The other applications had similar trends and are therefore not shown. The trends indicate that the calculated MRCs are vertically shifted by varying amounts, dependent upon the application. There is also a potential impact on the MRC shape, affecting the smaller cache sizes. The precise magnitude of the shifting or shape distortion varies across applications and shows no predictable pattern. As described in Section 5.2.1, *art* showed significant accuracy improvement in Figure 4b with the simplified processor mode.

Finally, Figure 5e shows the impact on the real MRC running in the simplified mode. In general, the real MRC is vertically shifted upwards relative to the complex mode.

5.3 RapidMRC for Sizing Cache Partitions

We briefly evaluate the usefulness of RapidMRC by applying it to sizing cache partitions for multiprogrammed workloads running on a shared-cache multicore processor. We compare the partition size chosen using the MRC supplied by RapidMRC versus the offline real MRC. Since in our current implementation we compute RapidMRC only once, we have selected applications that have fairly stable behavior throughout the measurement period.

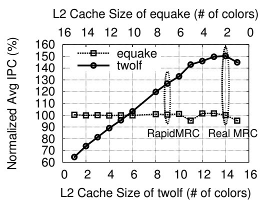
The workloads *twolf+equake* and *vpr+applu*[3] were executed on the POWER5+ but with the 36 MB L3 cache disabled. We found that the small working set size of these application pairs, combined with the abnormally large L3 cache, eliminated any shared cache performance problems: with the L3 cache enabled, the application pairs experienced a 98% hit rate to the L2 or L3 caches, leaving only 2% of accesses to main memory. Consequently, we disabled this unusually large L3 cache to re-introduce the shared cache performance problems seen by previous researchers who used commonly-found dual-core hardware configurations that do not contain L3 caches.

The *ammp+3applu* workload, in contrast, demonstrates a fully utilized hardware configuration. It was run on the POWER5, utilizing the 36 MB L3 cache and all 4 SMT

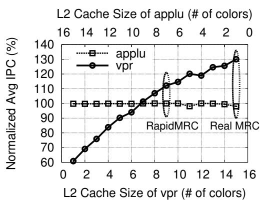
[2] *applu*, *apsi*, *art*, *equake*, *mgrid*, *swim*, *twolf*, *vpr*.

[3] Only the "place" phase of *vpr* was utilized.

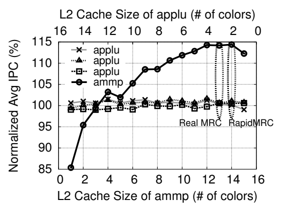
Chosen Cache Sizes		
Applications	Real MRC (#colors:#colors)	RapidMRC (#colors:#colors)
twolf : equake	14 : 2	9 : 7
vpr : applu	15 : 1	9 : 7
ammp : 3applu	13 : 3	14 : 2



a. twolf+equake



b. vpr+applu



c. ammp+3applu

Figure 7. Multiprogrammed workload performance as a function of L2 cache size.

hardware contexts. To reduce its search space, all 3 instances of *applu* were confined to sharing the same cache partition[4].

Since the chosen applications exhibit fairly stable behavior, the application MRCs shown in Figure 3 from both the RapidMRC and offline real MRCs were first fed as inputs to the partition size selection algorithm described in Section 4. Although this algorithm can be executed online with low-overhead, due to limitations in our current implementation of RapidMRC, we used this algorithm offline. The resulting chosen partition sizes are shown in the table of Figure 7.

Next, we ran the selected applications together with the L2 cache partitioned according to the suggested partition sizes. In addition to the cache configurations chosen from RapidMRC and the real MRC, all other possible partition sizes were also run to obtain an entire spectrum of multiprogrammed performance results, as shown in Figure 7. The graphs show the average IPC of each application for the entire multiprogrammed run of the application, normalized to the uncontrolled sharing configuration. The multiprogrammed combinations are terminated as soon as one of the applications ended.

Figure 7 shows that using RapidMRC, *twolf+equake* improved by 27%, *vpr+applu* by 12%, and *ammp+3applu* by 14%. In contrast, using the sizes chosen using the offline real MRCs resulted in improvements of 50%, 28%, and 14%, respectively. The cause of the performance gaps is the horizontally flat sections of the calculated MRCs in *twolf* and *vpr*, seen in Figure 3, which prevent the size selection algorithm from choosing the same sizes as with the offline real MRCs. Despite the performance gaps, these results illustrate that the MRC supplied by RapidMRC can help achieve performance gains when applied to cache partitioning.

For future work, we envision extending our current implementation to dynamically track MRC transitions and recompute optimal partition sizes accordingly. To enable dynamic L2 cache partition resizing in this vision, we have taken some initial steps and implemented a page migration mechanism with an attendant cost of 7.3 μs per 4 kB page.

6. Discussion

In developing RapidMRC, we have pushed the envelope of what is possible using today's PMUs. In particular, we have taken the data address capturing feature of the POWER5 processor, which is primarily intended for the purpose of *sampling*, to the extreme so that it can be used for *tracing*. To trace in this way, we force the processor to raise an exception at every L1 data cache miss, which obviously has substantial overhead. In addition, this method of tracing is inherently incomplete, as some data accesses are not recorded by the hardware PMU due to concurrency with other accesses. Fortunately, the results of our experimental analysis show that even with these limitations, the calculated MRCs are accurate in most cases, and the performance overhead is acceptable for our purposes. However, we believe more adequate hardware monitoring support will facilitate producing more accurate MRCs with much lower overhead.

Based on our experience, there are a few capabilities we would like to see in future PMUs. The first one is the ability of *tracing* data addresses into a small trace buffer, rather than a single data address register. This feature would allow an overflow exception to be raised only when the buffer is full, as opposed to on every data access. This would amortize the cost of exception handling over many data samples and thus greatly reduce monitoring overhead. Secondly, the trace buffer should be capable of recording all accesses, despite having several memory instructions in-flight. This seems to be feasible with a trace buffer instead of a single data address register. Thirdly, all accesses to the on-chip cache should be recordable, regardless of whether they are the result of processor memory instructions or hardware prefetchers. With these three features, for a short period of time, a complete trace of memory accesses performed by an application can be recorded. Finally, our experience with varying hardware performance counter capabilities, even within the same family of processors, leads us to believe that they should be standardized in an implementation-independent fashion so that they can be widely adopted across different platforms, perhaps analogous to the IEEE 754 floating-point standard.

7. Concluding Remarks

In this paper, we have shown a technique, called RapidMRC, to obtain the L2 miss rate curve of an application online by exploiting hardware performance counters found in modern processors. We have also shown that our transparent method produces fairly accurate MRCs with a runtime overhead that is substantially less than other software-based approaches. As an example of the utility of the calculated MRCs, we

[4] A simple heuristic is to place all cache-insensitive applications, indicated by their horizontally-flat RapidMRCs, into a single shared cache partition.

show they can be used for determining cache partition sizes in a shared cache environment, enabling up to 27% performance improvement compared to an uncontrolled cache sharing scheme. We believe that by providing a fairly accurate estimate of the cache needs of applications, RapidMRC will enable further optimization opportunities in on-chip caches.

We acknowledge the fact that we have exploited a PMU feature, i.e., continuous data address sampling, that is currently available only in IBM POWER5 processors. However, by demonstrating what can be accomplished with a simple PMU feature that is already implemented in a real processor, we hope to provide motivation to other processor vendors to adopt similar PMU features.

As for future directions, we would like to explore methods for further increasing the accuracy of calculated L2 MRCs. One such method is the ability to automatically track application phase changes and to dynamically calculate the L2 MRC of each phase. We would like to explore extending L2 MRCs to account for the impact of non-uniform miss latencies in addition to predicting the impact of misses on processor stall cycles. Finally, we would like to explore other online optimization opportunities that can be pursued using the online information provided by RapidMRC and its underlying trace of cache accesses.

Acknowledgments

We would like to thank our shepherd, Manish Gupta, for his guidance in helping to improve the quality of our paper. We would like to thank the following IBM folks for providing access to research equipment and technical expertise: Allan Kielstra, Toronto Software Lab; Jimi Xenidis, Watson Research Lab; and Santhosh Rao, Linux Technology Centre Ozlabs and Austin Linux Technology Center.

References

[1] D. Albonesi. Selective cache ways: on-demand cache resource allocation. In *MICRO*, pages 248–259, 1999.

[2] C. Antonopoulos, D. Nikolopoulos, and T. Papatheodorou. Scheduling algorithms with bus bandwidth considerations for SMPs. In *ICPP*, pages 547–554, 2003.

[3] R. Azimi, L. Soares, M. Stumm, T. Walsh, and A. Demke Brown. PATH: page access tracking to improve memory management. In *ISMM*, pages 31–42, 2007.

[4] R. Azimi, M. Stumm, and R. Wisniewski. Online performance analysis by statistical sampling of microprocessor performance counters. In *ICS*, pages 101–110, 2005.

[5] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *MICRO*, pages 245–257, 2000.

[6] E. Berg and E. Hagersten. StatCache: A probabilistic approach to efficient and accurate data locality analysis. In *ISPASS*, pages 20–27, 2004.

[7] E. Berg and E. Hagersten. Fast data-locality profiling of native execution. In *SIGMETRICS*, pages 169–180, 2005.

[8] E. Berg, H. Zeffer, and E. Hagersten. A statistical multiprocessor cache model. In *ISPASS*, pages 89–99, 2006.

[9] D. Bruening, E. Duesterwald, and S. Amarasinghe. Design and implementation of a dynamic optimization framework for Windows. In *FDDO*, 2001.

[10] B. Buck and J. Hollingsworth. An API for runtime code patching. *J. of High Performance Computing Applications*, 14(4):317–329, 2000.

[11] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *HPCA*, pages 340–351, 2005.

[12] S. Cho and L Jin. Managing distributed, shared L2 caches through OS-level page allocation. In *MICRO*, pages 455–468, 2006.

[13] J. Edler and M. Hill. Dinero IV trace-driven uniprocessor cache simulator. URL `http://www.cs.wisc.edu/~markhill/DineroIV`.

[14] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. Performance of multithreaded chip multiprocessors and implications for operating system design. In *USENIX ATC*, pages 26–26, 2005.

[15] F. Guo and Y. Solihin. An analytical model for cache replacement policy performance. In *SIGMETRICS*, pages 228–239, 2006.

[16] R. Iyer. CQoS: a framework for enabling QoS in shared caches of CMP platforms. In *ICS*, pages 257–266, 2004.

[17] R. Iyer, L. Zhao, F. Guo, R. Illikkal, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. QoS policies and architecture for cache/memory in CMP platforms. In *SIGMETRICS*, pages 25–36, 2007.

[18] J. Kim, J. Choi, J. Kim, S. Noh, S. Min, Y. Cho, and C. Kim. A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references. In *OSDI*, pages 119–34, 2000.

[19] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *PACT*, pages 111–122, 2004.

[20] Y. Kim, M. Hill, and D. Wood. Implementing stack simulation for highly-associative memories. In *SIGMETRICS*, pages 212–213, 1991.

[21] J. Liedtke, H. Härtig, and M. Hohmuth. OS-controlled cache predictability for real-time systems. In *RTAS*, pages 213–227, 1997.

[22] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *HPCA*, pages 367–378, 2008.

[23] C. Liu, A. Sivasubramaniam, and M. Kandemir. Organizing the last line of defense before hitting the memory wall for CMPs. In *HPCA*, pages 176–185, 2004.

[24] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, pages 190–200, 2005.

[25] R. Mattson, J. Gecsei, D. Slutz, and I. Traiger. Evaluation techniques and storage hierarchies. *IBM Systems J.*, 9(2):78–117, 1970.

[26] K. Meng, R. Joseph, R. Dick, and L. Shang. Multi-optimization power management for chip multiprocessors. In *PACT*, pages 177–186, 2008.

[27] M. Olszewski, K. Mierle, A. Czajkowski, and A. Demke Brown. JIT instrumentation: a novel approach to dynamically instrument operating systems. In *EuroSys*, pages 3–16, 2007.

[28] R. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *SOSP*, pages 79–95, 1995.

[29] M. Qureshi and Y. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO*, pages 423–432, 2006.

[30] N. Rafique, W. Lim, and M. Thottethodi. Architectural support for operating system-driven CMP cache management. In *PACT*, pages 2–12, 2006.

[31] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. A resource allocation model for QoS management. In *RTSS*, pages 298–308, 1997.

[32] A. Settle, J. Kihm, A. Janiszewski, and D. Connors. Architectural support for enhanced SMT job scheduling. In *PACT*, 2004.

[33] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *USENIX ATC*, pages 17–30, 2005.

[34] X. Shen, J. Shaw, B. Meeker, and C. Ding. Locality approximation using time. In *POPL*, pages 55–61, 2007.

[35] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *ISCA*, pages 336–349, 2003.

[36] A. Snavely and D. Tullsen. Symbiotic jobscheduling for a simultaneous multi-threading processor. In *ASPLOS*, pages 234–244, 2000.

[37] L. Soares, D. Tam, and M. Stumm. Reducing the harmful effects of last-level cache polluters with an OS-level, software-only pollute buffer. In *MICRO*, 2008.

[38] G. Soundararajan, J. Chen, M. Sharaf, and C. Amza. Dynamic partitioning of the cache hierarchy in shared data centers. In *VLDB*, pages 635–646, 2008.

[39] S. Srikantaiah, M. Kandemir, and M. Irwin. Adaptive set pinning: Managing shared caches in chip multiprocessors. In *ASPLOS*, pages 135–144, 2008.

[40] H. Stone, J. Turek, and J. Wolf. Optimal partitioning of cache memory. *IEEE TOC*, 41(9):1054–1068, 1992.

[41] E. Suh, L Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *The J. of Supercomputing*, 28(1):7–26, Apr. 2004.

[42] D. Tam, R. Azimi, L. Soares, and M. Stumm. Managing shared L2 caches on multicore systems in software. In *WIOSCA*, pages 26–33, 2007.

[43] D. Tam, R. Azimi, and M. Stumm. Thread clustering: Sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *EuroSys*, pages 47–58, 2007.

[44] D. Thiebaut, H. Stone, and J. Wolf. Improving disk cache hit-ratios through cache partitioning. *IEEE TOC*, 41(6):665–676, 1992.

[45] T. Yang, E. Berger, S. Kaplan, and J. Moss. CRAMM: virtual memory support for garbage-collected applications. In *OSDI*, pages 103–116, 2006.

[46] Q. Zhao, R. Rabbah, S. Amarasinghe, L. Rudolph, and W.-F. Wong. Ubiquitous memory introspection. In *CGO*, pages 299–311, 2007.

[47] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. In *ASPLOS*, pages 177–188, 2004.

[48] Y. Zhou, J. Philbin, and K. Li. The multi-queue replacement algorithm for second level buffer caches. In *USENIX ATC*, pages 91–104, 2001.