Estimating Application Cache Requirement for Provisioning Caches in Virtualized Systems

Ricardo Koller[†] Akshat Verma[‡] Raju Rangaswami[†]
[†]*Florida International University* [‡] *IBM Research-India*

***Abstract*— Miss rate curves (MRCs) are a fundamental concept in determining the impact of caches on an application's performance. In our research, we use MRCs to provision caches for applications in a consolidated environment. Current techniques for building MRCs at the CPU caches level require changes to the applications and are restricted to a few processor architectures [7], [22]. In this work, we investigate two techniques to partition shared L2 and L3 caches in a server and build MRCs for the VMs. These techniques make different trade-offs across accuracy, flexibility, and intrusiveness dimensions. The first technique is based on operating system (OS) page coloring and does not require change in commodity hardware or application. We improve upon existing page-coloring based approaches by identifying and overcoming a subtle but real problem of *unequal associative cache sets loading* to implement accurate cache allocation. Our second technique called *CacheGrabber* is even less intrusive and requires no changes in hardware, OS, or application. We present a comprehensive evaluation of the relative merits of these and other techniques to estimate MRCs. Our evaluation study enables a data center administrator to select the technique most suitable to his (her) specific data center to provision caches for consolidated applications.**

I. Introduction

Server consolidation via virtualization is gaining acceptance as the solution to overcome server sprawl by enabling the replacement of large number of low utilization servers with a small number of highly utilized servers. For this model to be successful, virtual machines (VMs) must be provided the illusion of running as a dedicated physical machine: isolated from other machines and fully protected. Commodity virtualization solutions enable isolating processing cycles completely and core memory to a large extent. However, processor caches are not virtualized. These include on-chip caches (L1 and/or L2), and off-chip caches (e.g., L3 caches). Improper management of cache resources can cause unexpected performance interference in consolidated environments. For instance, Verma *et al.* have observed a performance impact of up to a factor of 4 due to cache-unaware consolidation for many HPC applications [24]. Koller *et al.* extended this observation to enterprise applications in [12]. Similarly, Lin *et al.* showed that contention in a shared cache can lead to a performance degradation up to 47% [14]. Hence, an accurate characterization of the amount of cache required for ensuring performance isolation in a virtualized system is important for the success of server consolidation.

One of the most popular ways to characterize the usage of any memory (or cache) resource by an application is the classical *working set* model [6]. The core of this model is the concept of working set which is the set of pages accessed by a process in a period of time. A refinement to this concept is the *reuse set* which identifies the set of pages accessed and reused in a period of time. Notice that the reuse set gives a better idea about the cache requirements: there is no need to allocate space for pages that are not reused.

An alternative approach to characterize memory utilization is using *miss rate curves* (MRC) which has been widely applied for processor caches. The MRC of an application models the dependency of the cache miss rate of the application on the allocated cache size. MRCs offer an advantage (over the working set) of being able to model the performance impact of arbitrary sized caches. MRCs have been derived from the working set using both analytical techniques [6] as well as an LRU stack simulator [22], [26].

In a companion paper [13], we propose a unified model called the *Generalized ERSS Tree Model* that comprehensively characterizes working sets across all phases of an application. The core of the model is a metric called *effective reuse set size* ($ERSS$) that accurately captures the amount of cache required by an application in a phase to avoid capacity misses. Estimating the cache requirement for an application would require identifying all the phases with an $ERSS$ smaller than the largest processor cache. One can then pick the phase with the largest $ERSS$ among the selected phases and provision cache equal to its working set. MRCs, which capture this behavior for all the phases for an application can be used to identify the largest cache resident phase to estimate the amount of cache that needs to be provisioned.

Most existing techniques for identifying phases of execution or to infer the working set or the MRC of an application are fairly intrusive, requiring direct access to the operating system page tables, and/or instrumentation and recompilation of applications, or binary instrumentation. Techniques that depend on specific hardware counters are only available on mid range servers (e.g. the SDAR registers on the POWER6) [22]. However, typical data centers use commodity hardware and software to minimize management costs. Furthermore, server consolidation engagements are run by data center administrators with limited access to the hardware, application, operating system or hypervisor. Consequently, intrusive changes are not permissible in such an operating environment wherein administrators do not even have kernel level access to individual VM instances. Hence, a practically usable cache provisioning technique needs to operate at user level on commodity systems in order to be viable in a data center.

Paper Contributions.
The first contribution of this work is the creation of two new techniques for improved estimation of cache requirements for virtualized workloads. In this regard, we first improve upon existing OS page-coloring based approaches by identifying and overcoming a subtle but real problem of *unequal associative cache sets loading* to implement accurate cache allocation. We also design and implement a *CacheGrabber* technique to infer the MRC and an estimate of the cache requirement of an application at the software level without access to the application or the operating system. This second technique can be implemented on commodity hardware for any application without requiring any hardware, OS, or application level changes. This technique is valuable in data centers where applications run as a black box and application provisioning is done at a middleware level without direct access to application code or kernel.

The second contribution of this work is a comprehensive evaluation of various techniques for estimating application cache requirement. We discuss their relative strengths and weaknesses and identify the operating conditions under which one of the techniques should be used. We also show that our techniques can be used to infer MRCs for enterprise applications whose throughputs vary over time. We show that cache required for such applications can also be estimated using prior characterization at different throughput values. Our work allows server consolidation to be practically employed in data centers without suffering from performance impact due to cache contention.

II. Related Work

A. Memory and Cache Usage Models

1) Working Sets: The most popular model for modeling memory resource usage is the classical Working Set model, made popular by Denning in his seminal work and refinements [5], [6]. The Working Set model is based on the concept of resident set, or the set of active memory pages during an execution window. Once the resident set for an application is identified, memory or cache may be allocated to the application accordingly.

2) Miss Rate Curves: A second popular model for modeling memory or cache usage is the Miss Rate Curve (MRC) model. This has been a very popular model to allocate processor caches to applications [20], [22]. A Miss Rate Curve is composed of multiple points, where each captures the number of misses for an application for a fixed amount of memory or cache. The curve captures the behavior of the application as cache or memory is varied. The model is built for the lifetime of the application and provides an aggregate view across all the phases of an application. MRCs have also been used for memory allocation by Zhou *et al.* [26] and Rajkumar *et al.* [19] by transforming the processes MRCs into convex functions (through convex hulls) and then using a gradient descent algorithm to minimize the sum of misses for all processes.

3) Phase Hierarchy: The phase behavior of applications is a third important characteristic of memory or cache usage of the application. Batson and Madison [4] define a phase as a maximal interval during which a given set of segments, each referenced at least once, remain on top of the LRU stack. A phase transition indicates that the locality subset has changed, leading to a fault rate that is 100 to 1000 times higher than during a phase. They observe that programs have marked phase behavior and there is little correlation between locality sets before and after a transition. Another important observation made by Batson and Madison and corroborated by others [9], [20] is that phases typically form a hierarchy.

In a companion work, we present the Generalized ERSS Tree Model that captures the real cache requirement for each phase of the application [13]. The real cache requirement for an application in each phase is estimated using a metric called $ERSS$ and the $ERSS$ of the largest phase is used for provisioning caches. In this work, we present technique to estimate the MRC of an application and use it to estimate the $ERSS$ or the cache required for each phase.

B. Characterization Techniques

There are two popular approaches to infer the Miss Rate Curve for an application. The first approach creates the MRC statically by running the application multiple times, typically in a simulator with different memory or cache allocations [9], [20], [25]. A second approach uses a memory trace and a LRU stack simulator [11] to infer the MRC. MRCs in filesystems can be estimated using ghost buffers [10], [17]. Estimating the MRC dynamically for caches is a much more challenging problem but has been proposed for specific platforms using fairly intrusive techniques [22], [26].

Detecting phase transitions for taking reconfiguration actions has been another popular area of research. The Dynamo system [3] optimizes traces of the program to generate fragments, stored in a fragment cache, and a change in the rate of fragment formation increases is used as an indicator of phase transition. Balasubramonian *et al.* [18] use the dynamic count of conditional branches to measure working set changes. Dhodapkar *et al.* compute a working set signature and detect a phase change when the signature changes significantly [7].

Existing mechanisms to infer the memory resource usage or identify phases operate at the hardware or operating system level and make intrusive changes. In our work, we develop new techniques that can infer the cache usage of an application for various phases at a user level without requiring any intrusive changes to the application or the operating system. These user-level techniques can be easily employed in application staging environments before deployment.

C. Other Related Work

There have been efforts to estimate the memory usage of applications at compile time. Malkawi *et al.* present a compiler-driven memory management framework, where memory estimation is also done at compile time [15]. Shared Cache partitioning, both at software and hardware level, has been explored before to demonstrate that contention can lead to a performance degradation up to 47% [14]. A popular technique to partition caches employed by the operating system is page coloring. In page coloring, different applications are assigned cache-aligned memory to ensure that they can use only a fixed number of cache segments [14]. We employ similar ideas at the application level for cache partitioning and use it to derive the MRC for the application.

III. Framework to infer MRCs and Cache Requirement

We first discuss various alternative approaches to infer the cache usage of an application. We later use insights from these approaches to explore new techniques that are faster and applicable on commodity systems.

A. Cache Simulator Approach

Cache simulators can be used to build the MRCs for an application and use the MRCs to estimate the cache requirement for the application. Popular cache simulators like Valgrind [16] capture various aspects of the cache (e.g., eviction policy, associativity) and provide detailed cache statistics like hit and miss rate. In order to build the MRC, one can run the simulator with various cache sizes and use the corresponding miss rates to compute the MRC. Once the MRC is created, we can identify the phases that fit in the actual processor cache and pick the phase with the largest cache requirement. In a consolidated server environment, the estimates for all the virtual machines placed on the server can be used to estimate the amount of cache required on the physical server to ensure that application performance is not impacted post consolidation.

B. Full System Simulator Approach

Full System Simulators provide a similar and more accurate approach to estimate the cache requirement for an application. In this approach, we use a full system simulator configured with minimal amount of cache. The application is run on the simulator and all main memory accesses are tracked to obtain a memory trace. Once the memory trace is available, it is fed to an LRU cache simulator. We can obtain the MRC by changing the size of the simulated cache and use it to estimate the cache requirement for the application.

C. PMU Sampling approach

A third technique to estimate the cache requirement is to sample Performance Monitoring Units (PMU) and obtain memory traces from these samples. RapidMRC is a technique to dynamically create the miss-rate-curves for an application and is used for online optimizations [22]. The key idea behind RapidMRC is to use data sampling features available in the performance monitoring units (PMUs) of some modern servers (e.g., Sampled Data Address Register (SDAR) in IBM Power5, IBM Power6).

D. Need for a New Approach

All the existing techniques are based on enforcing resource limited execution for caches. To elaborate, the techniques use either a full fledged cache simulator (e.g., Valgrind) or obtain a memory trace and feed it through an LRU stack simulator. In either case, the goal is to ensure that the application has only a fixed amount of cache available for its use. Multiple runs of the simulator with varied amount of resource limit provides us the behavior of the application with varying sizes of cache or the MRC of the application.

Any technique that is based on simulation suffers from the problem of speed (accurate simulators are very slow) and accuracy. Further, typical simulators do not

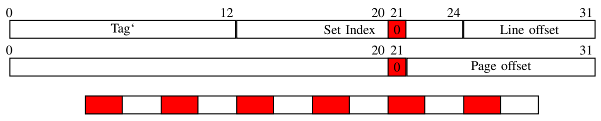

Fig. 1: The first array is the address line as interpreted for cache line translation in a $2MB$ cache (4-way associative cache with $4K$ cache sets and a cache line of 128 bytes). The second array is the address line as interpreted for translating $1KB$ size pages. We restrict the number of cache sets used by the application by generating only even-numbered pages.

provide an easy way to characterize multi-component applications. We will also show that simulation based approaches suffer from a phenomenon called *Unequal Associative Cache Sets Loading* that impacts the accuracy of the MRC obtained. PMU based techniques solve the problem of speed but are feasible only on a select set of hardware and suffer from the problem of accuracy.

IV. Resource Limited Execution for Caches

Hardware-based cache partitioning can be achieved through cache partitioning in simultaneous multithreaded (SMT) processors [8], [23]. Software partitioning techniques often employ page coloring. State of the art implementations of the above techniques are typically not currently available on commodity systems and implementing them is intrusive. In fact, static partitioning of cache resources using SMT (e.g., SMT implementations of IBM POWER5 and Intel Pentium4) was the only technique available in our hardware and kernel. Hence, we implement basic versions of cache reservation that can be driven flexibly.

In this work, we explore two different software-based techniques for cache limited execution which have different levels of intrusiveness: *(i)* utilizing cache associativity properties and page coloring [14], [21], and *(ii)* a *CacheGrabber probe* designed to continuously consume cache lines. We implemented these methods on the POWER6 processor and discuss the details of each below.

A. Associativity-based Partitioning

Page coloring has been traditionally used to partition CPU caches within the operating system. We use a similar idea to statically restrict the cache available to an application at compile time. Caches in modern systems are designed to be N-way set associative, i.e., a memory line can only be loaded into a set of N available cache lines. The set to which a page is mapped is determined based on a portion of address bits called the index (bits 13 to 24 in POWER6). Partitioning the cache for a specific application can be achieved by controlling this index for all the pages allocated to an application. For

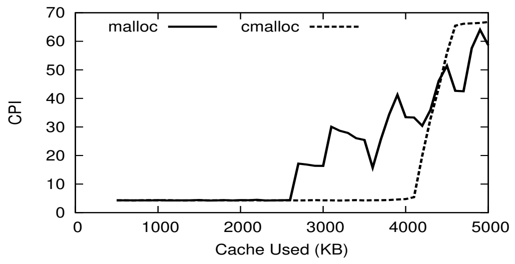

Fig. 2: Sets from the 4MB associative cache are not loaded equally due to fragmentation and associtivity. *malloc is the regular malloc and cmalloc allocates contiguous physical memory.*

instance, allocating pages physically aligned to exactly double the size of a page can effectively halve the number of cache lines available to an application. As shown in Figure 1, this will ensure that bit 21 of physical memory address references issued by the application will always get value 0. Pinning a bit of the index to 0 ensures that the index can no longer take all the $4K$ values, reducing the available cache sets to $2K$ only (using the other 11 bits). This directly leads to the application using at most $1MB$ of the available $2MB$ cache.

While the page-coloring based technique is seemingly straightforward, some subtleties introduce additional complexity. Most CPU caches are addressed physically. Therefore, to have control over the index, the memory allocator must have control over physical frame allocation. Moreover, partitioning by controlling index bits in the physical address is necessary but not sufficient to create an equally capable *subset cache*. Particularly, the lines allocated might not be mapped into the pertinent sets of an N-way associate cache with the same probability leading to wasted space because some sets of the subset cache are full while others have free lines. We call this problem *unequal associative cache sets loading* which we validated empirically. Figure 2 shows an application's performance change as its working-set changes on a processor with cache 4MB (the solid *malloc* line). Performance degrades even when the cache requirement of the application is 3MB, indicating that the application is unable to use the full 4MB.

The above problem can be addressed by first allocating frames contiguously in physical memory and then allocating pages from this physically contiguous space. We implemented a user-level allocator (*cmalloc*) that allocates memory from a ramdisk using *mmap*. The new allocator allocates memory from a contiguous space ensuring that all the cache lines of the subset cache are equal candidates for loading physical frames allocated to the application. As shown in Figure **??**, the *cmalloc* version of the memory allocator leads to an accurately partitioned subset cache of size 4MB.

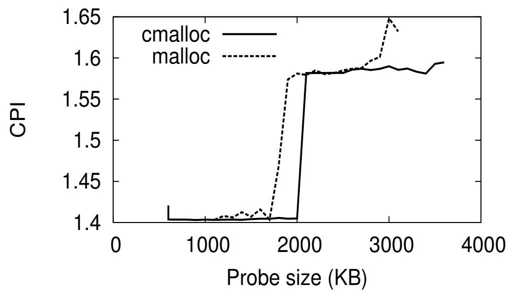

Fig. 3: CPI of application Vs Probe size. The profiled application uses 2 MB of L2 cache.

B. CacheGrabber: A Cache Partitioning Probe

We now present *CacheGrabber*, a probe that "grabs" a user-specified cache space, called the *probe size*, ensuring that the application(s) under study can access only the remaining amount of cache. *CacheGrabber* creates a vector of the required size and continuously accesses its elements. We increase the size of the vector till we observe a performance drop for both the probe and the application, indicating full utilization of the cache, and use it to infer the cache requirement of the application under study. We can compute the MRC of an application by varying the size of the vector and monitoring the corresponding impact on the hit and miss rate of the application.

The key to *CacheGrabber* is to ensure that the cache lines used by the probe should never be used by the application under study. Since cache replacement policies are variants of LRU, the above requirement can be met by touching the data set of the probe at a faster rate than any other application in the system. In other words, this would ensure that the vector used by the probe would always be resident in cache. In order to ensure a really high reuse rate for the probe, we used the $DCBT$ (Data Cache Block Touch) instruction present in the $PowerPC$ processors ($POWER6$ in this case). This instruction is available on most platforms and is typically used for prefetching – touch a line before it is going to be used so it can be loaded into the cache without incurring the cache latency. The $DCBT$ instruction is non-blocking and hence the probe does not incur any access latencies (L1 or L2). With the above design, *CacheGrabber* is able to achieve a CPI of 1.4, leading to a very high reuse rate (i.e. 1 cache access every 1.4 cycles).

We study the partitioning ability of *CacheGrabber* in Figure 3, where we examine the number of cache misses for an application under study as the size of the probe is reduced. The application has an $ERSS$ of $2MB$ and we observe a drastic drop in the performance of the application with a probe of size $2MB$ that uses $cmalloc$. Further, there is no drop in performance with probe sizes smaller than $2MB$. This allows us to correctly validate that the application under study has a L2 resident phase with a $2MB$ $ERSS$. We also observe that even if the probe does not use $cmalloc$, it is able to infer an $ERSS$ of 2.1MB, which is very close to the actual $ERSS$ value. The probe using $cmalloc$ allows it to effectively use the 2MB and no less (as shown in Figure 2). This implementation is non-intrusive, completely general, can be deployed easily in production environment and has good accuracy.

Cache partitioning techniques make an implicit assumption that the context switches between applications are reasonably frequent. If there is a significant delay between context switches, one application may drive out popular data for all other applications from the cache. In our experiments, we observed that the context switches were fairly quick and the applications used only the required cache. It is interesting to note that simulator based approaches do not take into account context switch times. Hence, even if a hypervisor allows large scheduling slots, a simulator based approach would compute the same MRC. However, *CacheGrabber* has the advantage of running on the real hardware and its estimates would be impacted accordingly. This ability of *CacheGrabber* to run in a real environment enables it to capture all such operating factors and accurately estimate the MRC for an application.

V. Experimental Validation

We performed a large number of experiments to evaluate how well the various techniques can identify the cache usage of real applications. We first describe the experimental setup used in our experiments.

A. Experimental Setup

We used the NAS Parallel Benchmark [1] to evaluate the efficacy of various cache partitioning techniques. We identified a set of applications from the benchmark that capture the spectrum of applications from small footprint to large footprint applications. We used the NPB 3 Serial version in out study. Some of the larger workload sizes of the benchmark took an inordinately long time to complete on the simulators and hence, we used the Class W workload size in our study.

We evaluate a suite of techniques on their ability to infer the cache resident working set sizes. Some of these techniques are implemented on real hardware and some use a simulator. We use the QEMU simulator [2] for obtaining memory traces using full system simulation. We used the open source Valgrind simulator [16] as representative of cache simulators. To evaluate

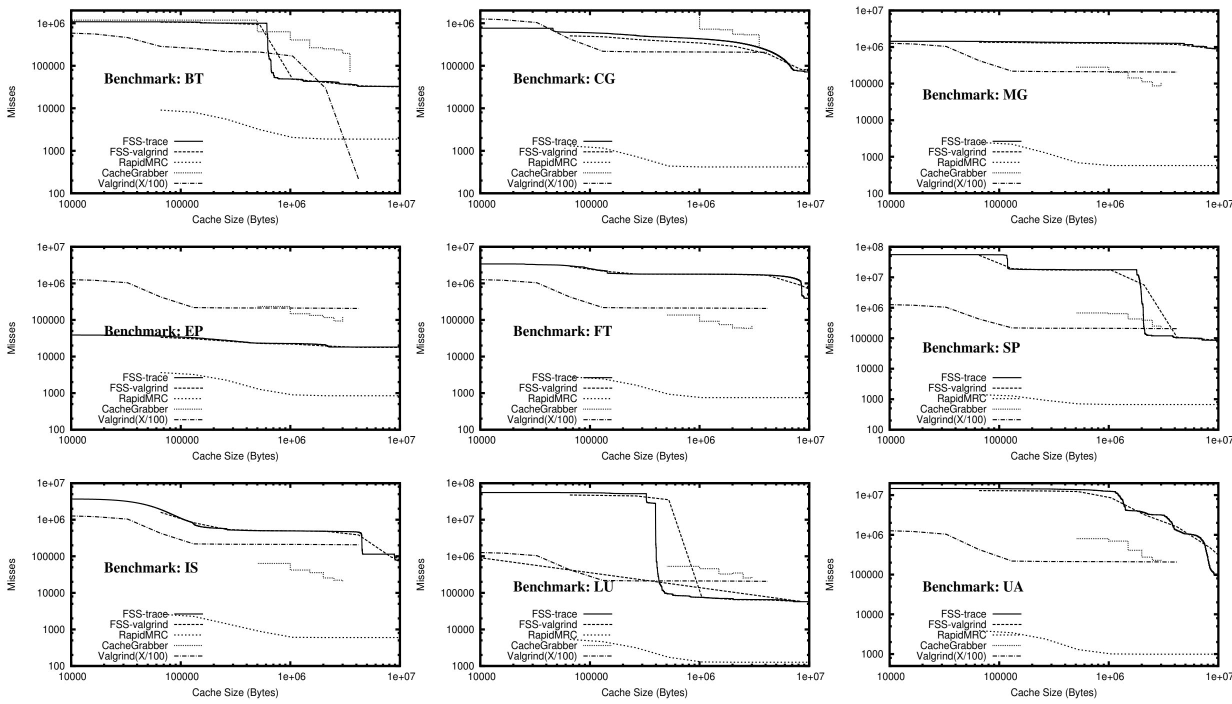

Fig. 4: MRCs for the benchmarks using the different techniques

the CacheGrabber and RapidMRC techniques, we implemented them on an IBM Power6 JS22 Bladecenter cluster. A blade in the cluster has $8GB$ RAM, $4MB$ of L2 cache and 4 IBM Power6 $4.2GHz$ processors. Each LPAR (VM) was entitled to $2GB$ RAM and 2 physical processors for our experiments.

We evaluated the following 5 techniques in our study. The techniques include QEMU-based full system simulation memory trace driven simulation using LRU (FSS-LRU) and Valgrind (FSS-VG), RapidMRC generated Power6 processor memory traces fed to Valgrind (RapidMRC), the CacheGrabber (CacheGrabber) technique, and the Valgrind (Valgrind) simulator directly. We could not use the associativity-based technique due to lack of $malloc$ primitive in the Fortran-based applications, indicating a portability limitation.

B. Comparative Evaluation

In order to compare the various cache partitioning strategies, we implemented all of them and obtained the MRCs of the NAS applications using them.

Figure 4 captures the MRCs using the different strategies. The first observation we make is that the CacheGrabber technique can plot the MRC only for the real range of the cache. This is a direct consequence of the fact that it operates on the real hardware. Further, the resolution of the CacheGrabber technique is 512K, which is the granularity at which the probe size is varied.

The resolution of the Valgrind cache-simulation techniques are limited to power of two values of cache size. This does not impact small sized caches where the granularity is good. However, as the cache sizes increase, it is unable to provide a fine resolution MRC. To take an example, the technique does not provide any estimates between $2MB$ and $4MB$ cache sizes. Hence, cache simulation is useful for L1 caches but not very useful for larger caches (L2 or L3).

We also observe that the MRCs from various techniques are not vertically aligned. This is a direct consequence of the fact that the simulation-based techniques are oblivious of prefetching and provide a higher estimate of misses (all prefetched data is also accounted as misses). On the other hand, PMU-based technique (e.g., RapidMRC) provides a lower estimate of the misses as it is sampling-based. CacheGrabber is the only technique that is based on real runs and provides an accurate estimate of the misses. However, an estimate of the misses is not necessarily required for estimating the cache required for an application in a phase. This estimate can be obtained using a change in the number of misses (or slope of the MRC curve).

The MRC for ua application is a good example to study the relative strengths and weaknesses of individual techniques. There are few dominant phases at $100KB$, $1.4MB$ and $9MB$ and a few less prominent phases. The CacheGrabber technique has a valid range between $0.5MB$ to $4MB$ and is able to clearly identify the phase

App	FSS-LRU	RapidMRC	CacheGrabber
bt	770K, 4.1M	1M	500K,1M,1.5M
cg	610K, 7.25M	512K	1M, 4M
ep	425K, 2.2M	512K	1M, 2.5M
ft	145K, 9M	512K	None
is	390K, 4.5M	512K	1MB
lu	530K, 7.25M	1M	1.5M
mg	580K,4.8M,5.8M,7.8M	512K	500K, 1M
sp	153K,2.1M,2.5M,3.9M	512K	1.5M, 2.5M
ua	1.4M, 2.4M, 9M	1M	1.5M, 2M

Fig. 5: Required Cache Estimates for the NAS benchmark with different techniques. RapidMRC and CacheGrabber run on real system with 4MB cache.

at $1.4MB$ (as $1.5MB$) as well as the less prominent phases in this range. The Valgrind based technique is able to identify the smallest phase but unable to identify any of the larger phases because of its granularity issues. The Full System Simulator based approaches are able to identify the 2 larger phases but does not identify the less prominent phases. The RapidMRC technique works at a coarse resolution due to sampling and does not identify many phases.

We next report an estimate of the cache required for the dominant phases of the applications in Figure 5 using different techniques for cache limited execution. We note that the cache requirement for dominant phases are usually found by all techniques. The RapidMRC technique does not find large phases well due to sampling, while the CacheGrabber techniques is unable to identify small phases. CacheGrabber requires the probe to have a reuse rate greater than the reuse rate of the application being profiled. Since phases with very small cache requirement (relative to cache size) tend to have a high reuse rate, CacheGrabber may fail to identify them. While not explored in this work, this can potentially be addressed by running multiple instances of CacheGrabber simultaneously to ensure sufficient CPU time for achieving the necessary *combined* reuse rate relative to the profiled application. However, CacheGrabber is very effective in identifying phases in the range between $0.5MB$ and the size of the cache.

CacheGrabber also has the advantage of very fast modeling time as the MRC is built using few runs (8 in our experiments) in real time. On the other hand, the simulator based approaches suffer a slowdown by a factor upto 100. RapidMRC suffers from a slowdown of 30% but does not produce fine-grained MRC due to sampling issues. Associativity-based technique can also built MRCs quickly but may not be feasible due to porting issues. Hence, our recommended profiling strategy is to use CacheGrabber to identify large phases, and then use either the cache simulator or the Full System Simulator based technique to identify smaller phases. Since smaller phases may also be short-lived, the

Technique	Accuracy	Intrusiveness	Overhead
Associativity	V. high	re-compilation kernel module	none
CacheGrabber	High	none	spare CPU
RapidMRC	Medium	Access to HW counters	1.3x
Full System Simulator	Medium	none	10 to 100x
Cache Simul.	Medium	none	30 to 100x

TABLE I: Comparison of Cache Limited Execution Techniques

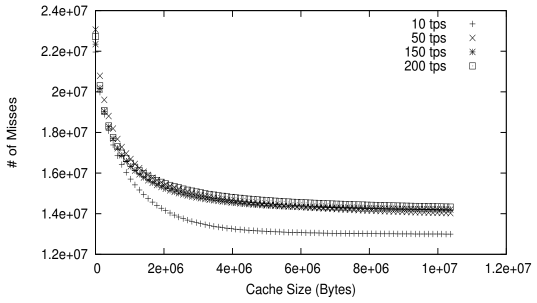

Fig. 6: MRCs for RUBiS at multiple throughput

simulator may be run only for a small part in the lifetime of the application. This would also help overcome the speed issues with this approach and provide an accurate estimation of the cache requirement for an application.

C. MRCs for Enterprise Applications

We have presented methods to estimate the MRC and cache requirement for batch scientific applications. However, a large variety of enterprise applications do not process batch requests. Instead, the applications are request-driven, where the overall throughput or request rate of the application varies with time. We now investigate if it is possible to estimate an MRC for such applications efficiently. Figure 6 capture the MRCs for Rice University Bidding System (RuBIS), a system that models e-bay. The application has a web-based front end where users can submit requests and a database to store the details about the items. We estimated the L2 hit and miss rates for the application at different throughput values and use them to plot the MRC.

We observe that the MRCs of an application retain the same nature with change in throughput. We do see that the actual number of misses may vary with throughput but the cache requirement of the applications does not change with time. This is a direct consequence of the fact that the throughput only impacts the amount of disk data that is streamed by the application. The streaming data is too large to be cached by processor caches. The working set cached in processor caches is typical the local state maintained by worker threads, which is constant for a given request type (e.g., browsing request type). Hence, the amount of cache required by the application does not change with increase or decrease in the number of requests. Hence, our techniques can be used to estimate the cache required by enterprise applications as well.

D. Hybrid Strategy

We summarize the relative merits of all the techniques in Table I. The associativity-based technique, if applicable, is very accurate with no overheads. RapidMRC and simulator-based techniques, if feasible on the platform and application under study, are ideally suited to identify small-sized phases as the overhead for smaller phases may be acceptable. CacheGrabber is probably the most flexible technique and works well for provisioning L2 caches. However, it can not identify very small phases and is not usable for L1 caches. Hence, we propose a *hybrid* strategy for fine-grained estimation of the cache requirement for all phases of an application. We recommend using CacheGrabber to identify coarse phases that are L2 resident, and then use one of the other approaches to identify fine-grained phases within the larger phases. Since the smaller phases are embedded in a larger phase, simulators can choose to only execute specific portions of the application, instead of the entire application. Thus, the *hybrid* strategy can identify phases for all levels of cache in an efficient manner.

VI. Conclusion

We have presented two new techniques to estimate the MRC and the cache requirement for an application. In order to design the techniques, we overcome significant challenges to accurate memory characterization such as *unequal associate cache sets loading*. Our first technique is based on page coloring and cleanly partitions cache for an application. However, the technique requires changes in memory allocation primitive (e.g., $malloc$ in C applications). Our second technique $CacheGrabber$ works at user level and operates on commodity hardware and software with no changes. We evaluate a variety of cache partitioning techniques which make different trade-offs along accuracy, flexibility, and intrusiveness dimensions. We distill our experimental study to recommend a hybrid technique that leverages the strengths of various techniques to identify the cache requirement for all the phases of an application. Finally, our observations on unequal associate cache sets loading could be of independent interest as well as aid in understanding classical memory resource usage models.

References

[1] National Aeronautics and Space Administration. NAS parallel benchmark. Online, http://www.nas.nasa.gov/Resources/Software/npb.html.

[2] Fabrice B. Qemu, a fast and portable dynamic translator. In *Proc. of the USENIX Annual Technical Conference*, April 2005.

[3] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *ACM SIGPLAN*, 2000.

[4] A. P. Batson and A. W. Madison. Measurements of major locality phases in symbolic reference strings. In *Proc. of ACM SIGMETRICS*, 1976.

[5] P. J. Denning. The working set model for program behavior. In *Communications of the ACM*, 1968.

[6] P. J. Denning. Working sets past and present. In *IEEE Tran. on Software Engineerng*, 1980.

[7] A. S. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *Proc. of ISCA*, 2002.

[8] A. El-Moursy, R. Garg, D. H. Albonesi, and S. Dwarkadas. Partitioning multi-threaded processors with a large number of threads. In *Proc. of ISPASS*, 2005.

[9] M. Karlsson and P. Stenstrom. An analytical model of the working-set sizes in decision-support systems. In *Proc. of ACM SIGMETRICS*, 2000.

[10] 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 *Proc. of USENIX OSDI*, 2000.

[11] Y. Kim, M. Hill, and D. Wood. Implementing stack simulation for highlyassociative memories. In *Proc. of ACM SIGMETRICS*, 1991.

[12] R. Koller, A. Verma, and A. Neogi. Wattapp: An application aware power meter for shared data centers. In *Proc. of IEEE ICAC*, 2010.

[13] R. Koller, A. Verma, and R. Rangaswami. Generalized erss tree model: Revisiting working sets. In *Proc. of IFIP Performance*, 2010.

[14] 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*, 2008.

[15] M. Malkawi and J. Patel. Compiler directed memory management policy for numerical programs. In *Proc. of ACM SOSP*, 1985.

[16] Open Source. Valgrind. Online, http://valgrind.org/.

[17] R. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *SOSP*, 1995.

[18] A. Buyuktosunoglu R. Balasubramonian, D. Albonesi and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general purpose architectures. In *IEEE MICRO*, 2000.

[19] R. Rajkumar, C. Lee, J. P. Lehoczky, and D. P. Siewiorek. Practical solutions for qos-based resource allocation problems. In *Proc. of IEEE Real-Time Systems Symposium*, 1998.

[20] E. Rothberg, J. P. Singh, and A. Gupta. Working sets, cache sizes and node granularity issues for large-scale multiprocessors. In *Proc. of ISCA*, 1993.

[21] 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 *IEEE MICRO*, 2008.

[22] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm. Rapidmrc: Approximating l2 miss rate curves on commodity systems for online optimizations. In *Proc. of ACM ASPLOS*, 2009.

[23] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *Proc. of ISCA*, 1995.

[24] A. Verma, P. Ahuja, and A. Neogi. Power-aware dynamic placement of hpc applications. In *Proc. of ACM ICS*, 2008.

[25] S. Woo, M. Ohara, E. Torrie, J.P.Singh, and A. Gupta. Methodological considerations and characterization of the splash-2 parallel application suite. In *Proc. of ISCA*, 1996.

[26] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. In *Proc. of ACM ASPLOS*, 2004.