Available online at www.sciencedirect.com
ScienceDirect
Performance Evaluation 00 (2010) 1–??
Performance Evaluation

Generalized ERSS Tree Model: Revisiting Working Sets

Ricardo Koller[a], Akshat Verma[b], Raju Rangaswami[a]

[a]*School of Computing and Information Sciences, Florida International University*
[b]*IBM Research, India*

Abstract

Accurately characterizing the resource usage of an application at various levels in the memory hierarchy has been a long-standing research problem. Existing characterization studies are either motivated by specific allocation problems (e.g., memory page allocation) or they characterize a specific memory resource (e.g., L2 cache). The studies thus far have also implicitly assumed that there is no contention for the resource under consideration. The inevitable future of virtualization driven consolidation necessitates the sharing of physical resources at all levels of the memory hierarchy by multiple virtual machines (VMs). Given the lack of resource isolation mechanisms at several levels of the memory hierarchy within current commodity systems, provisioning resources for a virtualized application not only requires a precise characterization of its resource usage but must also account for the impact of resource contention due to other co-located applications during its lifetime. In this paper, we present a unifying Generalized ERSS Tree Model that characterizes the resource usage at all levels of the memory hierarchy during the entire lifetime of an application. Our model characterizes capacity requirements, the rate of use, and the impact of resource contention, at each level of memory. We present a methodology to build the model and demonstrate how it can be used for the accurate provisioning of the memory hierarchy in a consolidated environment. Empirical results suggest that the Generalized ERSS Tree Model is effective at characterizing applications with a wide variety of resource usage behaviors.

Keywords: Memory Model, Cache Contention, Performance Isolation, Server Consolidation

1. Introduction

The ever-increasing gap between processing speed and disk bandwidth ensures that the allocation of resources at all levels of the memory hierarchy (henceforth also referred to simply as *memory levels*) significantly influence the performance of applications. The memory levels used by an application include on-chip caches (L1 and L2), off-chip caches (e.g., L3), DRAM, and other external memory caches (e.g., a flash-based disk cache [1]). Characterizing the resource utilization of an application at various memory levels has been an active research area over the years. While the consolidation of multiple applications on a shared hardware is not new, the

rise of virtualization has made such systems more the norm than the exception. In virtualized systems, multiple applications run on the same physical server and compete for resources at all memory levels. For instance, it has been shown that contention in a shared cache can lead to performance degradation — as much as 47% [2] and 75% [3] depending on the workload. An accurate characterization of the resource requirement of resident applications within each virtual machine (VM), while carefully taking into account the impact of resource contention due to other applications, is a prerequisite to ensuring that all applications meet their performance goals [2, 3, 4, 5].

Research in memory characterization can be broadly classified into models of memory usage and techniques that instantiate these models. The modeling work can be summarized using three related but disparate concepts. One of the most popular ways to characterize the main memory usage of an application is the classical working set model [6]. The core of this model is the concept of a resident set that defines the set of memory-resident pages of an application at any given point in time. An alternative modeling approach uses the Miss Rate Curve (MRC) [7] of an application to model the influence of the allocated memory cache size on the performance-influencing cache miss rate of the application. MRCs offer an advantage over the working set of being able to model the performance impact of arbitrarily sized caches. Finally, the concepts of phases and phase transitions have been proposed to model the memory resource usage behavior of an application as it changes over time [8]. A phase denotes stability of the active data set used by the application and phase transitions indicate a change in application behavior resulting in a change in the set of active data used.

In this work, motivated by the multi-tenancy of applications within virtualized enterprise data centers and clouds, we investigate the problem of characterizing the resource requirement of an application at all levels of the memory hierarchy for accurately provisioning resources in a shared environment. We identify the following properties for a memory model to be relevant in such shared environments.

1. Address all memory levels: The degree of sharing in a consolidated environment determines the amount of resources available to an application at each memory level. As opposed to a dedicated system, the resources available to an application at various memory levels in a shared environment do not have clearly demarcated values (e.g, 4KB L1, 2MB L2, 16MB L3 cache). A holistic view of resource consumption across all memory levels during an application's entire lifetime is thus necessary to provision memory resources for the application.

2. Identify dominant phases: The phases that are long-running have the greatest impact on application performance. On the other hand, reserving memory resources for short-lived phases, even though they may constitute larger working sets, may not be required. Therefore, model should adequately inform about the lifetime of the phases.

3. Address impact of contention: In spite of the copious work on partitioning caches at the hardware or the OS kernel level, solutions for cache partitioning are available only in high-end systems and that too only for certain memory levels. Today's commodity systems do not support flexible cache partitioning at any level. Hence, the model should address the impact of resource contention in a shared environment as a first class concern.

4. A commodity solution: Typical virtualized data centers run on commodity systems wherein administrators have little or no control on the internal operations of the guest VMs. Hence, a practically viable model should be built primarily from high level system parameters that do not require intrusive changes to the systems under consideration.

1.1. Gaps in Existing Models and Characterization

Existing memory models are insufficient to adequately model memory usage in a shared environment. First, existing techniques do not adequately capture the distinct phases of memory resource usage during an application's lifetime; working sets typically model the resource requirement for a fixed phase in the application while MRCs unify multiple phases into one, losing important distinctions between their individual resource consumption characteristics. Second, existing models focus on a specific level of memory; working sets are used for main memory [6] and MRCs for caches [9]. With the increasingly diverse levels of memory resources (L1, L2, L3, main memory, flash, and disks), a unified view of all memory resources is critically important for memory provisioning. For instance, if the working-set of a phase cannot be accommodated in L2, it may be possible to provision for it in L3, and provisioning a greater amount of L3 cache can reduce memory bandwidth requirement. Third, these models were not designed to model the impact of contention between applications which is important for ensuring performance isolation. An application's actual memory requirement may be very small, i.e., *capacity misses* may be close to 0, but it may have a large number of *compulsory misses* (e.g., streaming data access) which would effectively pollute the cache and thus impact other co-located applications significantly. Finally, most existing techniques for identifying phases or to infer the working set or the MRC are fairly intrusive, requiring direct access to the operating system page tables or fragment cache [10] that are only available within the kernel. Typical data centers use commodity software and administrators do not have kernel level access to individual virtual machine instances.

1.2. Contributions

We present the Generalized ERSS Tree Model, a new model for memory usage which both refines and unifies existing models. The Generalized ERSS Tree Model is based on a novel characterization that we term *Effective Reuse Set Size* (ERSS), which refines the working set by accounting for the reuse of data and architectural advances like prefetching. We define ERSS relative to the miss rate allowing it to model MRC concepts. The ERSS tree additionally captures hierarchical phases and their transitions. Additionally, we introduce two new parameters that intuitively model resource contention. The *Reuse Rate* captures the average rate at which an application reuses its *Effective Reuse Set* and the *Flood Rate* captures the rate at which an application floods a resource with non reusable data. We show that the former captures the vulnerability of an application's performance to competition from other applications, whereas the latter captures the adverse impact of an application on the performance of co-located applications. We overcome significant technical challenges to design a practical methodology for instantiating the model on commodity hardware without access to the target application or operating system. Our methodology uses (i) existing as well as new memory resource partitioning techniques to get the hit/miss rates for applications and (ii) a new phase detection technique that can be built solely from hit and miss rates for all levels of the memory hierarchy. Finally, we demonstrate the use of the model to characterize the amount of memory required to ensure performance isolation for applications in a consolidated environment.

1.3. Impact of Contributions

The Generalized ERSS Tree Model can be utilized for multiple purposes:

Resource Provisioning. An application requires resources at various memory levels in order to meet its performance objective. The provisioning problem differs from the well-explored allocation problem that addresses short-term allocation of memory resources, (e.g., by predicting

the working set size of the application in the short term). Resource provisioning, on the other hand, is a longer term decision and requires information about working sets for all phases and whose sizes may overlap across multiple memory levels. A unified model that identifies the phases across all memory levels, their durations, and the memory requirement in each phase, is therefore essential for accurate memory provisioning decisions.

Server Consolidation. Server consolidation via virtualization is gaining acceptance as the solution to overcome server sprawl by replacing many low utilization servers with a few highly utilized servers. Virtualization enables strict isolation of processing cycles and main memory. However, since caches are not virtualized, they can be a source of unexpected impact to an application's performance due to consolidation. Verma *et al.* have observed a performance impact of up to a factor of 4 due to cache-unaware consolidation for many HPC applications [3]. Similarly, Lin *et al.* show that contention in a shared cache can lead to a performance degradation up to 47% [2]. Our proposed unified model allows accurately characterizing the amount of cache resources required by each application including the over-provisioning required to address contention and ensure performance isolation between the applications.

Troubleshooting and Optimization. A unified model can also be used during performance troubleshooting. The model instance for the application can reveal gaps in the memory provisioned and required and pin-point the bottleneck memory level and the phases that are impacted, the amount of additional resources required, and the overall performance impact. Our model can also be used to debug an application by identifying anomalous memory usage in a phase. For instance, a gap between the expected memory use and the real memory use in a phase may indicate memory leaks. Modern processors are now also capable of switching ranks of idle memory to a low power self-refresh mode. Accurate memory characterization could help power-aware virtual memory systems [11] better estimate the minimum number of ranks required.

2. The Generalized ERSS Tree Model

We now present the Generalized ERSS Tree Model that unifies and refines the classical concepts of working set, miss rate curves, and phases. In line with previous work [8], we define a phase as a maximal interval during which a given set of memory lines, each referenced at least once, remain on top of an LRU stack. The model is based on a characterization of the core parameters that determine the memory capacity requirement of an application within a single phase of its execution. We extend this core concept to a tree-based structural model that characterizes the memory requirements of an application across all of its phases. We finally enrich this model with parameters based on access rates at various memory levels to model resource contention in shared environments. For the rest of the paper, we use the term *memory line* to denote the basic unit of access, thus denoting both cache lines and memory pages based on the context.

2.1. Capacity Related Parameters

A key characteristic of an application's memory usage is the number of memory lines the application requires to avoid capacity misses within a single phase of its execution. This metric assumes that there is no contention from any other application for the resource.

Definition 1. In Use Resident Set (IS): The In Use Resident Set for a phase is the set of all distinct virtual memory lines that are accessed during the phase. This notion is the same as the classical working set [6], but restricted to a single phase of execution.

Parameter	Phase i	Duration	InUse Resident Set	Reuse Set	Effective Reuse Set Size	Miss/Hit Ratio	Reuse Rate	Flood Rate
Notation	P_i	θ	IS	RS	$ERSS$	Δ_M	ρ_R	ρ_F

Table 1: Phase centric parameters of the model.

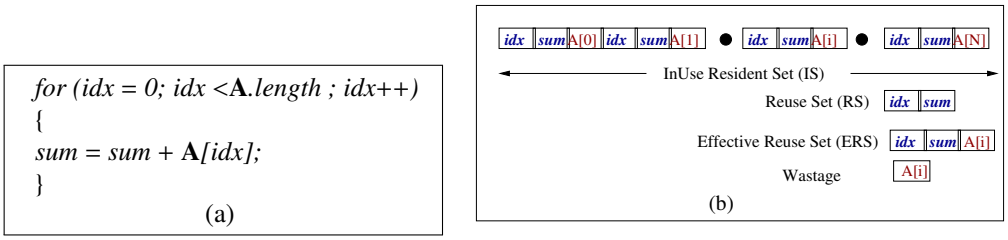

Figure 1: An illustration of the model concepts.

While the In Use Resident Set describes the virtual memory lines in use by an application during a phase, we are interested in the physical memory lines that need to be provisioned. A virtual memory line that is not reused may not need a physical memory line to be provisioned for the entire phase which can then be used to host other lines. Hence, the In Use Resident Set does not adequately capture the amount of physical memory needed by the application.

Definition 2. Reuse Set (RS): The Reuse Set is the subset of In Use Resident Set that is accessed more than once during a phase.

While the *Reuse Set* is a better approximation for the memory requirement of an application, it lacks certain key properties. First, the *Reuse Set* does not capture any capacity that is required to prevent the non-reusable data from evicting the reusable data. The actual memory requirement of the application during a phase may thus be larger than the *Reuse Set*. Second, the *Reuse Set* may contain some virtual memory lines that are reused very infrequently. The performance degradation of the application by not provisioning physical memory for these lines may be negligible and thus these lines do not contribute in a significant way to the effective memory requirement of the application during the phase. Finally, prefetching may hide latencies for accesses that are part of a stream of accesses if the rate of data access is low. This may further reduce the effective number of memory lines required by an application below the *Reuse Set* size. Thus, we define a metric that more accurately captures the amount of memory required by an application within a phase.

Definition 3. Effective Reuse Set Size (ERSS): The Effective Reuse Set Size is the minimum number of physical memory lines that need to be assigned to the program during a phase for it to not exhibit thrashing behavior [12]. The ERSS is defined with respect to a miss/hit ratio Δ_M and is denoted by $ERSS(\Delta_M)$.

Definition 4. Miss/Hit Ratio (Δ_M): This parameter defines the ratio between the number of misses and the number of hits during a phase for a given memory resource allocation.

The new phase centric metrics introduced above are summarized in Table. 1. We illustrate the above concepts with an example in Figure 1. Consider a phase where a program computes the sum of all the elements in an array in the variable *sum* (Figure 1(a)). The Inuse Resident Set

(IS) of the program in this phase is the array $\boldsymbol{A}$ and the variables *idx* and *sum*. Since all accesses to $\boldsymbol{A}$ are compulsory misses and only the two variables *idx* and *sum* get reused, the size of the reuse set is 2. The minimum Miss/Hit Ratio (Δ_M) is $1/2$ which can be achieved by provisioning physical memory for 3 integers — 2 integers to hold the reuse set and 1 more as a buffer for the current array element. Hence, the $ERSS(1/2)$ is 3 and different from the size of both IS or RS.

Phase transitions in typical programs are abrupt, i.e, the miss/hit ratio is constant for a large range of memory size allocations and increases/decreases significantly with small change in memory resource at a few memory resource allocation sizes [7]. We validate this observation for applications in the NAS benchmark suite [13] in Figure 2 which reveals sharp knees at a few $ERSS$ values. Since the $ERSS$ for different values of Δ_M around the phase transition (or knee) are similar, one can represent the $ERSS$ for each phase by a single $ERSS$ value corresponding to a default Δ_M; in this work, we used the minimum $ERSS$ at which the derivative of Δ_M w.r.t. memory resource size is less than 0.1 as the default value for that phase.

2.2. Generalized ERSS Tree

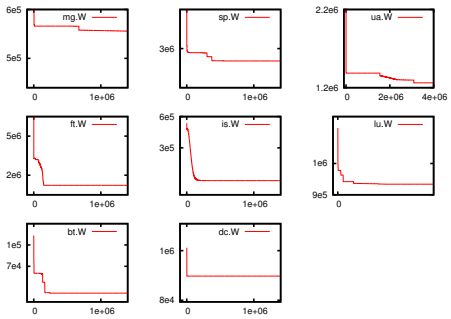

Figure 2: MRCs for the NAS benchmark applications.

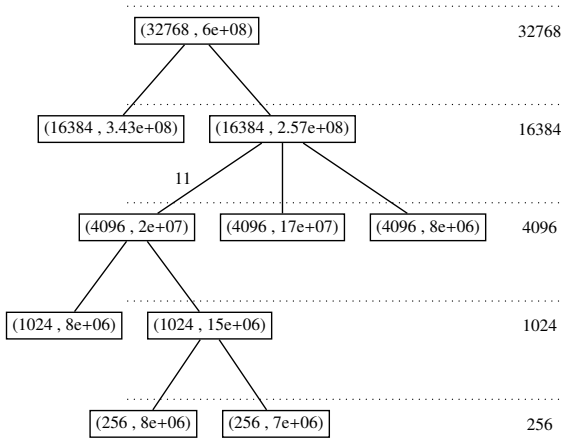

Figure 3: A sample Generalized ERSS Tree.

So far, we have introduced the parameters that describe the memory requirement of an application in a single phase. We now present the Generalized ERSS Tree Model that characterizes all the phases of an application. The Generalized $ERSS$ Tree of an application is a tree structure, where each phase is represented as a node specified by its duration (θ) and $ERSS(\Delta_M)$ function. The phase duration is defined in terms of number of virtual memory accesses and is thus platform-independent. If the $ERSS(\Delta_M)$ function has a sharp knee, we replace the function with a default $ERSS$ value. Smaller phases contained within a larger phase are captured by a parent-child relationship in the tree. Further, if a small phase occurs multiple times within a larger phase, the edge weight between the two nodes represent the number of small phases. Finally, since a single phase may contain multiple phases with different characteristics, a node may have multiple children.

An example $ERSS$ tree is shown in Figure 3 that describes the resource usage of the *bt* application in the NAS benchmark suite. Each node represents a phase with two parameters ($ERSS$,θ). The tree contains 5 levels of phases with the largest phase having a length of 6×10^8 memory accesses and containing two smaller phases, each with an $ERSS$ of $16MB$. The first

phase has a length of 3.43×10^8 memory accesses and the second phase has a length of 2.57×10^8. The second phase has three embedded phases of $4MB$ each, where the first child phase repeats 11 times. Given such a tree, one can easily identify the phases that would be resident in any level of memory. A typical example of resident locations of the phases at various levels of the memory hierarchy is shown using dotted lines in the figure.

2.3. Rate Parameters

Sharing introduces another dimension of complexity to the resource usage behavior of applications. Since commodity systems do not support strict isolation of various cache resources across applications, the effective cache size (at any level of the cache hierarchy) available to an application is directly influenced by the resident sizes and rates of accesses of co-located applications. We now extend our model to estimate application memory resource usage with multiple co-located applications, explicitly accounting for the resource competition.

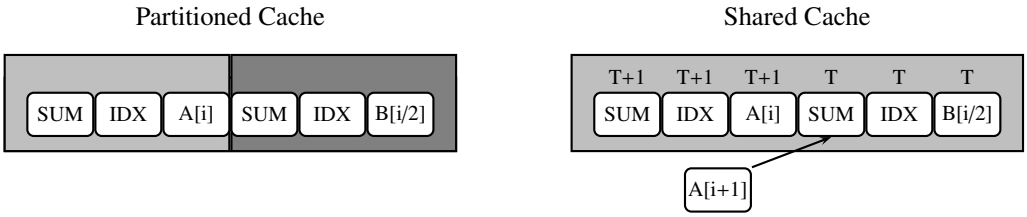

Figure 4: Contention in a shared cache.

Consider an application A_1 executing the example code in Figure 1. A co-located application A_2 computes the sum for another array B in a VM that is assigned a smaller fraction of the CPU than that assigned to A_1. Let us assume that due to its smaller CPU share, A_2 accesses the cache at half the rate of A_1. Figure 4 contrasts the cache allocation when the cache is partitioned and the cache is shared. For a partitioned cache, a cache size equal to the sum of the $ERSS$ of two applications is sufficient to ensure no performance impact due to consolidation. For a shared cache, however, the application with a higher rate of access floods the cache and evicts elements of the reuse set of the application with a lower rate of access (e.g., $A[i+1]$ evicts B's *sum* variable instead of $A[i]$ since $A[i]$ was most recently used at time $T+1$ whereas B's *sum* was last used at an earlier time T in Figure 4).

In order to understand this, note that the memory resource required in a phase consists of two parts: (i) the data items in its Effective Reuse Set for the phase and (ii) a *buffer* to handle misses during the phase without evicting items in (i). Intuitively, the amount of *buffer* that an application requires is determined by its miss rate during the phase. Furthermore, the cache flooding caused due to these misses would impact other applications using the same memory resource. With this motivation, we formalize the effect of resource contention between applications and refine the Generalized ERSS Tree Model with a set of *rate* parameters. These parameters are based on platform-specific parameters like clock speed, cache and memory bandwidth, and Cycles Per Instruction (CPI) for various instructions.

Definition 5. Flood Rate (ρ_F): The Flood Rate of an application for a given phase is the average rate at which the application floods the cache with memory lines that are not part of its Effective Reuse Set.

The Flood Rate of an application determines the amount of buffer space required in a memory resource in order to prevent the reusable data of the application from being evicted. In other words, the flood rate is the cache miss rate of the application, restricted to a single phase.

Flooding due to an applications introduces competition not just for the items in the effective reuse set of the application but also for the other co-located applications. A memory line that is part of the Effective Reuse Set would remain in an LRU cache if it is accessed frequently enough to get preference over the transient items that flood the cache. We formalize this observation using the Reuse Rate. Formally, the Reuse Rate of a memory line is the average rate at which the line is reused. If lines in the Effective Reuse Set have widely varying reuse rates, it may be possible that provisioning a slightly lesser amount of memory and evicting data that is used less frequently leads to no noticeable performance impact. This would imply a smooth $ERSS(\Delta_M)$ function. However, sharp knees in typical application MRCs imply that most of the data in the Effective Reuse Set are equally reused. Thus, we define the Reuse Rate of an application in a phase as the average reuse rate across the memory lines in the reuse set.

Definition 6. Reuse Rate (ρ_R): The Reuse Rate of an application for a given phase is the average rate at which each memory line in the Effective Reuse Set is reused.

If the rate at which an application floods the cache with non-reusable data is less than the reuse rate, the non-reusable data gets evicted quickly and it is adequate to provision the cache using the Effective Reuse Set of the application. On the other hand, if the reuse rate is less than flood rate, items in the Effective Reuse Set are evicted ahead of the non-reusable items and the cache provisioning must also includes any buffer required to hold the non-reusable data. We term this additional buffer requirement as *Wastage*.

Definition 7. Wastage (W): The Wastage for an application in a phase is defined as the buffer required to hold non-reusable data and prevent transient eviction of reusable data by the non-reusable data.

It is interesting to note that the *Reuse Rate* and *Wastage* concepts are naturally related — at a constant flood rate, a higher reuse rate leads to lower Wastage and vice-versa. Further, since computing the *Effective Reuse Set* is not practically feasible, we compute these parameters using alternate means. We use the following simultaneous equation to compute the *Reuse Rate* and *Wastage* using the measurable parameters of *Flood Rate*, $ERSS$, and $HitRate$.

$$\boxed{W = \frac{\rho_F}{\rho_R}, \text{ if } \rho_F \geq \rho_R \quad \& \quad W = 0, \text{ otherwise}} \qquad \boxed{\rho_R = \frac{Hit\ Rate}{ERSS - W}} \tag{1}$$

In the example in Figure 1, if the time taken to compute the sum of the vector of size N is T units, the flood rate (ρ_F) of the application is N/T. Similarly, the $HitRate$ for the above example is $2N/T$. Since the $ERSS$ is 3, using the above equations, one can compute the Reuse Rate (ρ_R) for each element in the reuse set as N/T and Wastage W as 1.

2.4. Using the Generalized ERSS Tree Model for Provisioning

We wrap up this section with a discussion of how to utilize the model described so far when provisioning resources for a set of consolidated applications. The cache provisioning approaches in the literature today all suggest allocating an amount equal to the sum of the working sets of each application. The proposed $ERSS$ metric leads to a more accurate estimate of the cache

requirements of an application but yet assumes that the application is running in isolation. To address this gap, we modeled the impact of contention due to co-located applications with the additional parameters of *ReuseRate*, $FloodRate$, and $Wastage$ that enable us to perform cache provisioning more reliably. Every application has a fixed *Reuse Rate* and an LRU-based eviction policy (typically employed in caches today) implies that the application with the smallest *Reuse Rate* is the most likely to face eviction during contention. We formalize this notion in the form of the *cache critical application* in a set of consolidated applications and determine the condition for ensuring performance isolation across this set.

Definition 8. Cache critical application: For a set of N applications A_i, we define the cache critical application A_c as the application with the smallest reuse rate, i.e., $A_c = \arg\min_i \rho_{Ri}$. For a set of N applications A_i, we define Wastage w.r.t. to the cache critical application A_c as

$$W_c = \begin{cases} 1 & \text{if } \sum_{i=1}^{N} \rho_{Fi} \leq \rho_{Rc} \\ \sum_{i=1}^{N} (\frac{\rho_{Fi}}{\rho_{Rc}} - W_i) & \text{otherwise} \end{cases} \quad (2)$$

Definition 9. Isolation Condition: A set of N applications A_i sharing a common resource of capacity C, bandwidth B are said to satisfy the isolation condition iff the following conditions hold:

$$\boxed{\sum_{i=1}^{N} ERSS_i + W_c \leq C,} \qquad \boxed{\sum_{i=1}^{N} (\rho_{Ri} ERSS_i + \rho_{Fi}) \leq B} \quad (3)$$

Before consolidating, it is critical that the isolation condition holds at various levels of the memory hierarchy to ensure performance isolation. To determine if this is true, the $ERSS$ tree for each application must first be created. All phases that were resident in a specific level of the memory hierarchy when the application ran stand-alone is determined. Consequently, the phase, termed P_{big}, with the largest $ERSS$ for each application and whose duration is above a pre-determined threshold, is identified. These are the phases for which we would need to ensure the isolation condition (Equations 3) must be met. This process is then applied to each level of the cache hierarchy for a given system to conclusively establish if the applications can be consolidated on the given hardware. Finally, in applications with hierarchical phases (smaller phases embedded within larger ones), the hit rate and $ERSS$ for larger phase includes access to memory lines of the smaller phase as well. However, an accurate characterization of the larger phase should be made independent of the accesses to any embedded smaller phases. Hence, the *ERSS* and *Hit Rate* is calculated as the marginal (or additional) $ERSS$ and *Hit Rate* respectively over the smaller phase.

3. Constructing the ERSS Tree

We now present a methodology to construct a generalized ERSS tree (henceforth referred to as ERSS tree), representing an instantiation of the Generalized ERSS Tree Model for the application under consideration.

3.1. Methodology Overview

The Generalized ERSS Tree Model is a recursive model with a subtree composed of smaller phases embedded within a larger phase. Our methodology to create the ERSS tree is based on the observation that one can create a coarse tree (with few levels) and increase the resolution of

the tree by identifying phases for additional resource sizes iteratively. We start with the root node of the tree which corresponds to the largest unit in the memory hierarchy (typically disk-based storage) and use a three-step *node generation process* to identify and incorporate phases of the longest duration. We then recursively refine it to include phases at lower levels of the tree by iteratively applying the node generation process for decreasing sizes of memory. To deal with noise or data-dependent execution, the process is repeated for multiple runs and only the phases that are identified in all runs are included in the model. The *node generation process* consists of
1. Refinement Level Identification. In this step, we identify the next size of memory resource, $Avail_{mem}$, (e.g, 2 MB) using which the tree should be refined further.
2. Resource Limited Execution. In this step, we identify the memory resource R that best matches $Avail_{mem}$ and ensure that the application executes with only $Avail_{mem}$ amount of R available. The techniques to ensure this reservation depends on the memory resource R and is detailed in Sec. 3.3. This step also creates an *execution trace* including the hits and misses for application requests to the specific resource, which serves as the input to *Atomic Refinement* step.
3. Atomic Refinement. This step uses the execution trace following the *Resource Limited Execution* to refine the ERSS tree. Atomic refinement is described in detail in Section 3.4.

It is straightforward to use the ERSS tree for an application generated using the above node generation process iteratively in a consolidation scenario. Prior to consolidation, we refine the tree to closely reflect the proposed allocation of various memory resources for each application to determine actual ERSS sizes at various levels of the memory hierarchy. We then use Equation 3 to determine if the planned allocations are adequate or over-provisioned for the dominant phases at each memory level. Next, we elaborate on the steps of the node generation process.

3.2. Refinement Level Identification

The *node generation process* must be applied for various levels (sizes) of memory resources in order to create the ERSS tree. A significant advantage of the three step process is the complete flexibility in refining the ERSS tree at the required granularity. The refinement level identification step allows a model that focuses on interesting parts of the tree to create higher resolution sub-trees. Selection of the refinement level in this step would ideally be based on the level of provisioning granularity required and candidate allocation sizes for each memory resource. Thus, the methodology allows easy refinement of the tree for the range of the actual memory assignments (e.g., at all candidate sizes for the L1, L2, and L3 caches and main memory).

3.3. Resource Limited Execution

Resource limited execution facilitates application execution for a specific size of the resource under consideration and records the memory hit and miss counters for the application. For example, to identify the memory phases at $ERSS = 2MB$ on a machine with 64KB L1 and 4MB L2, a user would run the application with a resource limited L2 cache size of $2MB$ and measure the L2 hit and miss rates. The hit and miss rate information is available by default on most commodity platforms for all levels of the memory hierarchy. We now present techniques for resource limited execution of the application for a user-specified size limit for various memory levels.

3.3.1. External Memory

The external memory phases of an application form the highest level of the ERSS tree. In legacy systems, the external memory data for an application is different from the other levels of memory in the sense that there are no resource miss events at this level. Consequently, phases

and ERSS descriptions in legacy systems are inconsequential for application performance. On the other hand, these considerations are relevant for systems which employ an external memory device as a cache. Examples of such systems abound in the literature including performance-improving caches [14, 15] and energy-saving caches [1, 16]. Such external memory devices are typically block devices (e.g., disk drives or solid-state drives). Controlling the size of a block device cache for resource limited execution is achieved easily with block device partitioning. Block I/O tracing tools are available in most commodity operating systems (e.g., Linux blktrace [17]) which work irrespective of the type of the underlying block device. These tools support execution tracing at the partition granularity to ensure non-interference with other block I/O operations in the system. By associating the cache partition block accesses with hits and non-cache partition block accesses to misses, hit and miss events can be recorded.

3.3.2. Main Memory

Techniques for limiting the main memory (henceforth RAM) available to the entire operating system to a specific fraction of the physical RAM exist in many systems. In AIX, we use the *rmss* command to ensure that the operating system can use only the specified size (fraction) of the total physical memory. Once we ensure memory reservation, we run the application and log the page fault rate through the lifetime of the application.

The *Atomic Refinement* step of the node generation process optionally uses the hit rate for higher accuracy. The hit rate for memory pages can be estimated using a binary rewriting API like Dyninst [18]. The Dyninst API can attach itself to an executing binary and dynamically instrument the application to get the complete memory trace. An alternative approach to obtain RAM hit rates is to use a full system simulator (e.g., Mambo, QEMU, etc.) and run the application with different RAM configurations.

3.3.3. Processor Caches (L1 / L2 / L3)

Various techniques have been proposed to partition the L2 cache at both hardware and software levels. These techniques, however, are not available in commodity systems and implementing these intrusive changes on a production server is not always feasible. The lack of flexible user-level partitioning schemes and fine-grained cache line monitoring counters creates significant challenges in further refining the models for cache resident phases in a non-intrusive fashion.

We developed and implemented two new techniques to partition the cache and record the hit and miss events. Our techniques are accurate for L2 and L3 caches and work with limited accuracy for L1 cache as well. The first technique uses ideas from page-coloring typically employed by the operating system [2, 19] to ensure that the application uses only a fixed amount of cache. However, this technique requires application re-compilation. Our second technique, named *CacheGrabber* runs a probe process on the server that continuously utilizes a fixed amount of cache resources, *reusing* the utilized cache in a manner such that it becomes unavailable to the application being characterized. In addition to these, we directly implement two previously proposed techniques for inferring the miss rate curve (MRC) for caches. The first technique uses a cache simulator to simulate caches of different sizes which directly leads to the MRC. The second technique uses sampling of performance monitoring unit (PMU) registers to create a memory trace and replays the trace through a LRU stack simulator to infer the MRC [20]. Further details on these techniques are beyond the scope of this paper.

3.4. Atomic Refinement

The *atomic refinement* step uses the hit and miss events in the execution trace to refine the ERSS tree for that level of memory. If hit rate information is not available for a memory level, only the miss rate information is used for the refinement. There are two components to atomic refinement at a high level: (i) it detects phases and phase transitions, size estimates, and the instruction time-line during which they occur and (ii) it uses these phases for ERSS *tree refinement*. This ability to detect phases with only the hit/miss information at various levels of the memory hierarchy makes model instantiation feasible in real data center environments.

3.4.1. Phase Detection

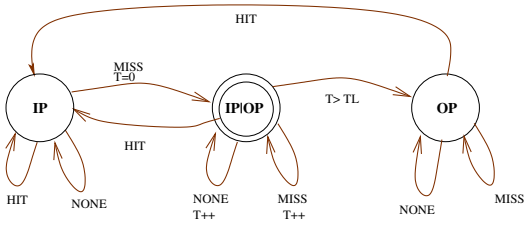

Figure 5: Phase Identification State Diagram.

We model the phase transition behavior of an application at a specific memory level using a state diagram, where transitions are triggered by the hit/miss events. The states and the transitions identify various phases and their durations. We model the following states:

1. InPhase (**IP**): The state denotes that the application is within a phase that fits in the memory resource under consideration.
2. OutPhase (**OP**): The state denotes that the application is within a phase that does not fit in the memory resource under consideration.
3. InPhase or OutPhase (**IP|OP**): The state denotes that it is unclear if the application is either in an InPhase or an OutPhase.

The HIT, MISS, and NONE events are *compound* rather than individual events. For instance, a HIT event requires the next window of events exceeds a pre-specified threshold percentage of hits, rather than a single hit event; if hit events are unavailable, then the we conversely use the requirement that the percentage of misses should be below a certain threshold. A MISS event is defined similarly. A NONE event is one which is neither a HIT nor MISS.

The thresholds used in the above definitions can be computed from the miss rate trace (typically available for all memory levels) for the application as follows. Using the histogram of the miss rates in the trace, we define the lower threshold at the 10^{th} percentile and the upper threshold at the 90^{th} percentile. Due to the steep nature of $ERSS(\Delta_M)$ (Figure 2), such a thresholding is sufficient to identify if the phase fits in the available memory resource or not.

The starting state is $(IP|OP)$, indicating that the initial state is unknown. A HIT event in $(IP|OP)$ indicates the start of a phase that fits in the available memory resource and we transition to (IP) state. The absence of a HIT indicates either a *phase transition* or an OutPhase state. We distinguish between the two by using a threshold TL to limit the maximum length of a phase transition. If the time T spent in $(IP|OP)$ exceeds TL, an OutPhase is detected marked by a transition to (OP) state. A MISS event from (IP) indicates a phase transition $(IP|OP)$ and a HIT event in (OP) indicates the start of a phase that fits in memory and is captured by a transition to (IP). The detailed state transition diagram is described in Figure 5.

3.4.2. Tree refinement

The *Tree refinement* step uses the node generation process to add new levels in the $ERSS$ tree and/or refine the existing nodes of the tree. The choice of the level to refine is closely determined by the target architecture and the sizes of the various hardware resources in the memory hierarchy. Let $\mathcal{M}_i$ be the memory level currently being investigated and let $ERSS_{\mathcal{P}}$ be the ERSS for a parent phase $\mathcal{P}$, *s.t.* $ERSS_{\mathcal{P}} > \mathcal{M}_i$ The goal is to determine sub-phases within $\mathcal{P}$ that may potentially be impacted by the memory level $\mathcal{M}_i$. If phase detection within $\mathcal{P}$ for memory level $\mathcal{M}_i$ leads to more than one child phase (IP or OP), a child node for each child phase is added to the parent node. Each InPhase (IP) fits in $\mathcal{M}_i$ memory and is marked with an $ERSS$ of $\mathcal{M}_i$ in the child node, whereas the $ERSS$ of each OutPhase (OP) node is set to the parent memory value $\mathcal{M}_{i-1}$ (as it does not fit in the memory level $\mathcal{M}_i$). If the node generation process with memory level $\mathcal{M}_i$ does not identify more than one phase, the parent phase is refined in the following manner. If the number of misses in the parent phase is found to be equal to those observed with $\mathcal{M}_i$, we refine the ERSS value at the parent node as $\mathcal{M}_i$. However, if the number of misses are larger at the memory value $\mathcal{M}_i$, we retain the earlier $ERSS_{\mathcal{P}}$ value for the parent node.

4. Experimental Validation of the Model

We now evaluate the need and accuracy of Generalized ERSS Tree Model. In particular, we address the following questions.

1. What is the need for a unified ERSS tree model?
2. How do reuse rate and flood rate impact memory provisioning?
3. Is the model sufficiently accurate to ensure isolation for consolidated workloads?

4.1. Experimental Setup

We used three different experimental testbeds in our evaluation. Our first test-bed was an IBM JS22 Bladecenter cluster with 4 3.2 GHz processors and 8 GB RAM with $4MB$ L2 cache. The experiments conducted on this test-bed used the L2 hit/miss counters available on the system. Our second test-bed was the QEMU full system emulator. QEMU runs memory accessing instructions natively on the host via dynamic binary translation. We modified the software-MMU version of QEMU and inserted tracing code at binary translation points. Specifically, each time a *translation block* (i.e., binary blocks between jumps and jump return points) is sent to the inline compiler, we insert code for recording every load and store instruction. Since the addresses used by loads and stores may be unknown at translation block compile time, the appropriate register values are recorded at run time. Timestamp information is collected from the tsc register in $x86$ and $IA64$. We ran Linux on the modified QEMU emulator configured with 1.8GB of physical memory with several workloads. The emulator itself ran on a 2.93 GHz Intel Xeon X7350 processor. The traces were then fed into a LRU stack simulator to build the MRCs. As our final test-bed, we used Valgrind to perform fine-grained analysis on single application instances and their resource usage behavior across the memory hierarchy, primarily for generating data to support our approach leading to the Generalized ERSS Tree Model.

Two sets of workloads were used in conducting the evaluation. The first set were the $daxpy$ and $dcopy$ benchmarks from the Basic Linear Algebra Programs (BLAS) library [21], which represent building blocks for basic matrix and vector operations. We modified these benchmarks to control the size of memory used, the number of iterations, and injected appropriate idle periods

to programmatically control memory access rates. These benchmarks mimic behavior common in many high-performance, scientific computing workloads and the fine-grained control allowed us to experiment with a wide variety of memory reuse and flood rates. Our second set of workloads are from the NAS Parallel Benchmark [13]. The NAS benchmarks mimic a wide spectrum of representative application behaviors, from CPU intensive to memory intensive. It also includes benchmarks that represent computational science applications (e.g., *bt* captures the basic calculation of Computational Fluid Dynamics).

4.2. The need for the Generalized ERSS Tree Model

We start by motivating the need for a model such as the Generalized ERSS Tree Model to completely describe the resource usage characteristics of applications, including the need for the ERSS metric for characterizing phases and the hierarchical ERSS tree model for characterizing the overall resource usage of an application.

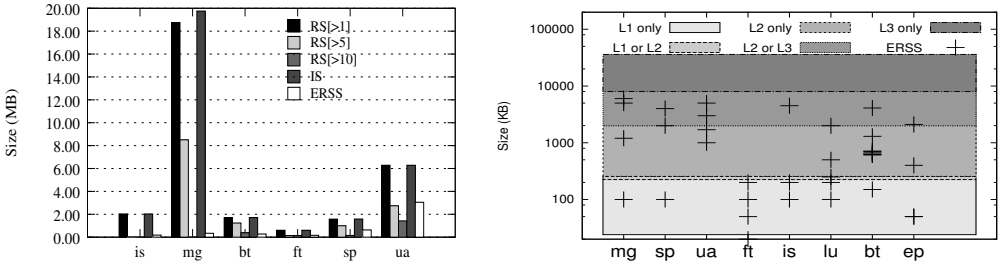

Figure 6: (a) IS, RS, ERSS for one phase. (b) ERSS of various phases for NAS applications.

Need for the ERSS metric: We first examine the need for the new ERSS metric introduced in this work for describing working set sizes within a single phase. Specifically, we study if the default ERSS for a phase can be inferred from known measures such as the Inuse Set (IS) or the Reuse Set (RS). We refine the reuse set to require at least a specified number (k) of reuses in a phase, allowing us to define multiple notions of reuse denoted as $RS[> k]$. We used the Valgrind test-bed and the NAS applications to characterize the resource usage behavior within phases in real applications. Figure 6(a) depicts the IS, RS and $ERSS$ for a randomly chosen phase within NAS benchmark applications. It is evident that the $ERSS$ metric is distinct from all the alternative measures considered, including variants of reuse set. Since the $ERSS$ characterizes application resource usage more accurately than either the IS or RS, it is evident that an accurate memory model should be based on $ERSS$ and new techniques should be designed to accurately characterize it.

Need for an ERSS tree: Resources at various levels of the memory hierarchy on stand-alone servers typically have disjoint ranges and provisioning can be handled independently for each resource. First, we show that in production, virtualized data center settings, depending on the type and number of co-located VMs, the amount of resource at individual levels that would be available to a single VM may no longer fall into pre-specified ranges. Further, we demonstrate that a single application may have multiple phases and the $ERSS$ values for these phases may span across multiple levels of the memory hierarchy.

To accomplish the above, we used the Valgrind test-bed to compute the $ERSS$ values for all distinct phases of the NAS benchmarks. Further, to get an estimate on the range of memory resources available to individual applications, we examined the configuration data sheet of a production data center with virtualized servers hosting one or more VMs. Based on the placement per the configuration data sheet, we divided the L1 and L2 cache resources to each VM in proportion to the CPU allocation. We take the maximum/minimum L1 (or L2) cache available to any VM across all servers in the data center as an upper/lower limit on the cache that a VM can use. Thus, we noted that as opposed to a fixed cache size on stand-alone servers, the cache sizes on virtualized servers changes significantly based on the number of hosted VMs.

To remove the influence of outlier configurations specific to the data center, we discarded the top 10% VMs with the largest cache sizes. We then plotted the $ERSS$ for various phases of NAS applications to identify the memory resource that they would fit into in Figure 6(b). We observed that most workloads encompass more than one resource level owing to distinct $ERSS$ values within phases of different granularities. Further, the $ERSS$ of some phases have sizes that overlap in the range for two memory resources. Hence, during memory provisioning, if a phase does not fit in a low level cache, it may be possible to provision a higher level cache for the phase. This underscores the importance of a unified model that captures resource usage across all levels for better cache and memory provisioning.

Application	**bt**	**cg**	**ep**	**ft**	**is**	**lu**	**mg**	**sp**	**ua**
#phases	12	4	3	4	4	5	5	4	6
$ERSS_{max}$	4.1MB	7.25MB	2.2MB	8MB	4.5MB	7.25MB	7.8MB	4.1MB	3MB
$ERSS_{min}$	150KB	50 KB	50KB	20KB	100KB	100KB	100KB	100KB	1MB
# large phases	2	2	2	2	2	2	4	4	2

Table 2: Distinct, dominant phases for NAS applications.

Need for phase duration: We use the phase duration (θ) parameter in our model to characterize the relative importance of a phase. Table 2 presents the total number of phases as well as the phases that are long-lived. We define a phase as long-lived if running the phase located within and outside of its required memory level leads to a difference of at least 10% in total number of misses at that memory level (across all phases). We noted (not shown) that even for applications with a large number of distinct sized phases, we can safely ignore a large number of the phases (e.g., 10 out of 12 phases for *bt* are very small). It is adequate to provision memory resources only for long-lived phases because doing so would lead to little or no impact to application performance.

4.3. Impact of Reuse and Flood Rate

We now evaluate the importance of the new parameters of our model, the *reuse rate* and the *flood rate*, in determining the performance of an application. In particular, we study how the *Isolation Condition* gets influenced by these parameters. The BLAS benchmarks, which allow varying the memory access rates, were ideally suited for this experiment. We used *daxpy* with a cache resident memory size and *dcopy* as the application that floods the cache with a very large array. In other words, *daxpy* controlled the *reuse rate* and *dcopy* controlled the *flood rate* in these experiments. We set the $ERSS$ of *daxpy* to $1.6MB$. Since the *dcopy* application has no reuse behavior at all, its $ERSS$ for the data cache is 0. The memory traces are fed through an LRU stack simulator.

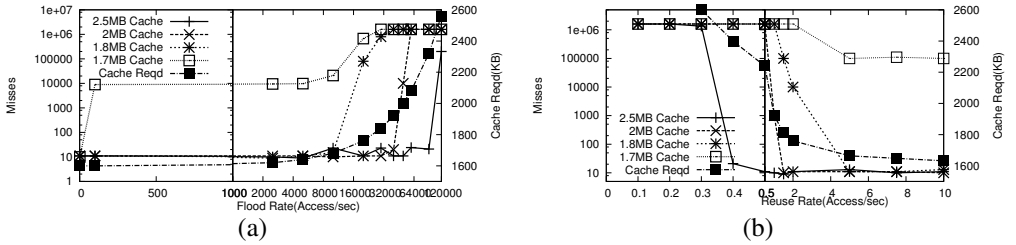

Figure 7: Misses for *daxpy* as we increase (a) the flood rate of *dcopy*. *Reuse rate for daxpy is 1/sec.* (b) the reuse rate of *daxpy*. *Flood rate for dcopy is 32KBps. ERSS for daxpy is* $1.6MB$. *X-axis has two scales*

Figure 7(a) studies the number of misses for *daxpy* as the flood rate of *dcopy* is varied. The reuse rate of *daxpy* was fixed at 1 per second. Earlier models would have predicted that the combined cache requirement for these applications as $1.6MB$ and no performance impact to either application due to consolidation with cache sizes greater than $1.6MB$. Our model (specifically, Equation 3), on the other hand, estimates a different cache requirement for each flood rate of *dcopy* (also shown in Figure 7). We observe that *daxpy* runs in two modes, either with very few misses or very high misses, for most cache sizes. We also note that the transition from very few to large number of misses happens at approximately the flood rate predicted by the *Isolation Condition*. For example, for a cache size of $2MB$, the transition happens at a flood rate of 60000. This is the flood rate at which the total cache requirement, as predicted by Equation 3 exceeds $2MB$.

The only exception to this bimodal behavior is for a cache size of $1.7MB$, which has 10000 or more misses, even when the cache requirement is $1.65MB$. To understand this behavior, we further analyzed the trace and found that flooding can sometimes be bursty. Consequently, the instantaneous flood rate may be higher than the average flood rate of an application, leading to an unanticipated increase in the number of misses. Combined with the fact that the buffer for the Wastage in this configuration is only $100KB$ ($1.7 - 1.6KB$), this buffer gets overrun. However, as long as the buffer for Wastage is $200KB$ (i.e., total cache provisioned is $1.8MB$) or more, this effect is not seen. Hence, our model is able to capture the impact of flooding and predict cache requirement with very high accuracy, as much as within $100KB$ of the actual cache space needed for isolation.

Next, we study the impact of the *reuse rate* of an application on its own performance in the presence of an additional cache flooding application. Figure 7(b) depicts the number of misses for *daxpy* as its reuse rate is varied. The flood rate of *dcopy* was fixed at $32KB$ per second. We observe that the *reuse rate* has significant impact on performance, directly influencing the amount of cache space required for ensuring isolation. Our model is able to account for the reuse rate and is once again able to predict the actual cache requirement with a high degree of accuracy.

4.4. Evaluating the model under consolidation

4.4.1. Accuracy of the Isolation Condition

To understand how the model functions when used for predicting isolation, we examine the cache isolation requirements for several pairs of applications in the NAS benchmark suite using the IBM Power6 test-bed. Later, we shall evaluate isolation for more than 2 consolidated workloads. Table 3 depicts characteristics of the NAS applications as well as the effect of contention (in the form of number of misses) when some of these applications are paired up against other

	BT	CG	EP	FT	IS	LU	MG	SP	UA
Misses	54349	188651	5358	65758	211226	257676	382629	1559421	1053185
Hit Rate	967139	-	23283	499913	3000191	79965121	9876	18010923	10914267
ERSS	1MB	-	400KB	200KB	300KB	1.5MB	200KB	2MB	3MB
BT	166238	**358642**	34723	123266	43565	**869982**	557936	**1104074**	**1934911**
EP	61275	210938	8343	87342	19173	264892	401546	1453682	1071482
UA	**1934827**	**1445027**	1192866	1248833	1115466	**3635310**	1648460	**4352033**	**4458986**

Table 3: Characteristics for stand-alone execution of the NAS benchmarks (top three rows). Number of misses for consolidated scenarios with application pairs (bottom three rows). *Combinations in* **bold** *show contention.*

applications (bottom 3 rows). For brevity, we focus our discussion here on 5 benchmark applications, namely BT, EP, UA, SP, and LU, that serve to adequately illustrate the necessity and accuracy of our model in determining performance isolation. The BT application has a moderate $ERSS$ value and moderate number of misses. EP has a small $ERSS$ and a few misses. UA has a large flood rate, as indicated by the large number of misses, as well as large $ERSS$. SP has characteristics similar to UA. Finally, LU has a moderate $ERSS$ but a high flood rate.

First, we point out that using the data in Table 3, existing models would predict performance isolation for all the consolidation scenarios except when consolidating SP and UA. Such a decision is arrived by summing up the ERSS (serving a proxy for the working set here) values, which exceeds the available test-bed cache of $4MB$ only for combining SP and UA. However, we note that more than 30% of the consolidation scenarios observe performance degradation. This points out the inadequacy of the existing state-of-the-art modeling approach when modeling performance isolation under consolidation. As we illustrate next with specific application pairs, our model is able to predict correctly whether $4MB$ of cache is sufficient for consolidation for more than 95% of all the pairwise consolidation scenarios possible with the NAS benchmark suite.

We now illustrate using several consolidation scenarios where simple models fail. First, we observe that EP, owing to its low flood rate and $ERSS$, does not affect the performance of other applications and can thus be consolidated with all the applications. BT can be consolidated with most applications barring SP and UA which have high $ERSS$ values. BT cannot be consolidated with LU because the $Wastage$ created by the high flood rate of LU cannot be accommodated in the remaining available cache after provisioning for BT. Interestingly, BT can be consolidated with MG, an application with a higher flood rate than LU but very small $ERSS$. This is because even though the $Wastage$ is high, there is approximately $3MB$ of unused cache space to accommodate it. We also observe that UA can be consolidated with small $ERSS$ applications only. We observe that EP with its low flood rate and $ERSS$ is a good consolidation candidate with any application. On the other hand, applications such as BT can be consolidated with other applications to a varying degree depending on the $ERSS$ and flood rate of the competing applications whereas applications like UA can be consolidated with very few applications.

Our mix of applications has another interesting application CG, which does not have a clearly defined knee point in the miss rate curve. We note that CG, due to the lack of a well-defined ERSS value, impacts the performance for most applications in the NAS suite because the isolation condition is not satisfiable. We can consolidate CG with EP though because of the very small amount of cache required for the phases in EP. We can hereby infer that an application like CG should not be consolidated with any other application whereas

All of the above predictions were only possible due to the availability of novel model concepts such as flood rate, reuse rate, and $Wastage$. Hence, we conclude that the model is not

only sufficient but also necessary to identify candidates for consolidation, whereas existing models that are oblivious to reuse and flood rates may allow consolidation scenarios that lead to adversely high performance impact.

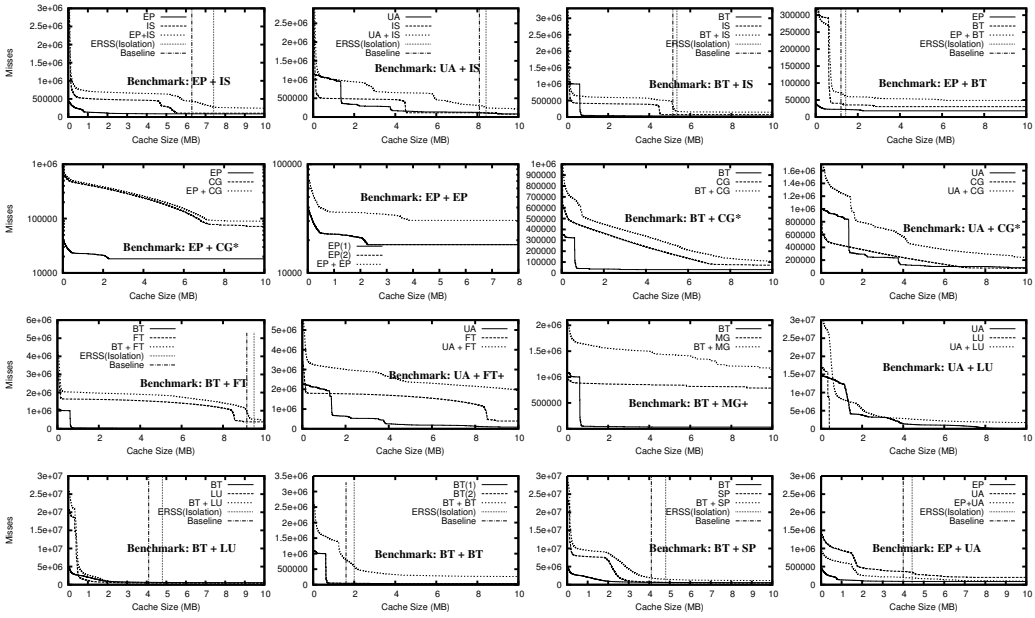

Figure 8: MRCs before and after consolidation with predicted and baseline cache requirement for isolation. (+) indicates that cache requirement exceeds $10MB$ and (*) indicates no clear knee.

Our previous experiment illustrates the ability of the Generalized ERSS Tree Model to predict whether a given amount of cache is sufficient to isolate the performance of consolidated applications. We now investigate the ability of our model to predict isolation for application pairs for a range of cache values using the $QEMU$ test-bed. In these experiments, instead of focusing on phases with $ERSS$ smaller than $4MB$, we focus on the phase with the largest $ERSS$. Further, since the two consolidated applications may run for different durations, we terminate the run as soon as one of the applications complete when obtaining MRCs.

Figure 8 presents the consolidation MRCs for a few randomly chosen NAS benchmark application pairs. Using our model, we estimate the cache required to ensure isolation of the benchmarks and report it. For comparison, we also present a baseline cache size obtained by adding the largest ERSS values that could be cache-resident. For visual clarity, we report the cache isolation prediction values only when the baseline and cache isolation prediction values differ by at least $0.2MB$ and both are less than $10MB$.

The experiment that consolidates BT and SP illustrates the importance of incorporating $Wastage$, while provisioning cache. Due to the low reuse rate of BT and high flood rate of SP, the $Wastage$ approaches $1MB$. Hence, we predict a cache requirement of $4.9MB$, which is very close to the actual cache requirement of $4.95MB$ (the last point at which the MRC curve has a slope greater than 0). Simply summing up the size of the two $ERSS$ values provides $4.05MB$ ($0.9MB$ or 18% less than the actual requirement). On the other hand, for applications with small flood rates (e.g., $EP + EP$), the $Wastage$ is small. Hence, the baseline value and the value pre-

dicted by our model are both very close to the real cache requirement. The MRCs also validate our earlier observation that applications such as *CG* without a clearly defined ERSS values are not good candidates for consolidation and we find that the isolation property is not satisfied for the corresponding experiments (e.g., $EP + CG$, $BT + CG$). We observe once again that the cache requirement for consolidating *UA* is large, exceeding practical cache capacity values ($10MB$ in our experiments) in some cases. These experiments establish the accuracy of our model to predict the actual cache requirement for a large range of cache values.

4.4.2. Evaluating Fixed Over-provisioning Solutions

Parameter	Mean	Median	Minimum	Maximum	Standard Deviation
Value	865KB	630 KB	80KB	3MB	690KB

Table 4: Aggregate ***Wastage*** statistics for pairwise consolidation runs of NAS benchmark applications.

We have shown that our model accurately characterizes cache contention and the additional cache required to buffer the space requirement due to flooding (or *Wastage*). Administrators may potentially consider provisioning a fixed amount of additional cache in anticipation of buffer requirement to deal with the excess requirement due to *Wastage*. To determine the effectiveness of such a strategy, we investigate the aggregate nature of *Wastage* for consolidated workloads. Table 4 lists the mean *Wastage* as well as other statistical measures across various pairwise consolidation runs of the NAS benchmark application suite. We observe that there is large gap between the maximum and minimum *Wastage* even across applications within the same suite. Further, we note that the mean *Wastage* is a significant fraction (20% or more) of typical L2 cache sizes (e.g., $4MB$) and that the standard deviation is significant as well (close to $700KB$). Hence, provisioning an average amount of excess cache to accommodate *Wastage* may lead to low cache usage when the *actual Wastage* is low (over-provisioning) and lead to performance impact (under-provisioning) when the actual *Wastage* is high. We thus conclude that a gross estimate of *Wastage* is not very effective for ensuring cache isolation.

4.4.3. Evaluation Using a Consolidation Benchmark

Our final set of experiments evaluate the accuracy of our model on a consolidation benchmark using the Xen hypervisor running on top of QEMU. We place 4 VMs on the hypervisor running the following: (i) Web tier of RUBiS, an auction site prototype modeled after eBay.com, at a throughput of 50 req/second, (ii) *BT* benchmark that performs CFD calculations, (iii) SpecPower benchmark [22] that evaluates power and performance characteristics of servers, built on top of SpecJBB, and (iv) an idle VM.

Parameter	*BT*	*RUBiS**	*SpecPower*
ERSS	0.8MB	2.4MB	4.3MB
Misses	500K	5M	5.1M
Hits	2.8M	19.9M	30.5M

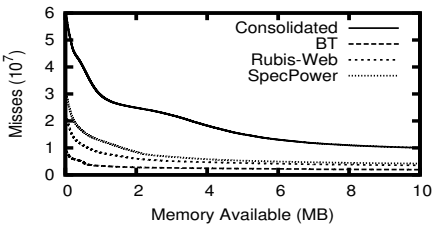

Figure 9: Model Parameters and MRCs for Consolidation Benchmark. (*) denotes cache critical application.

Figure 9 lists the model parameters for the workloads and the required cache estimated by our model and the baseline model. The baseline cache requirement is $7.5MB$ obtained as the sum of $ERSS$ values. However, our model predicted an estimated an actual cache requirement of $8.5MB$ when accounting for $Wastage$ due to each application. The consolidated MRC reveals that the real cache requirement is $8.4MB$, which is very close to the value predicted by our model. On the other hand, the baseline model has an error close to $1MB$ for this consolidation benchmark. This final experiment with a diverse set of applications conclusively establishes the accuracy of our model for realistic operating scenarios. It also highlights the inadequacy of existing memory models to characterize the resource contention in shared memory resources.

5. Related Work

5.1. Existing memory models

The most popular model for application main memory usage is the classical *working set* model and its refinements [6, 23]. The working set model is based on the concept of resident set, or the set of active memory pages within an execution window. Once the resident set for an application is identified, the operating system may allocate memory to the application accordingly. A second popular model for resource usage in the memory hierarchy is the Miss Rate Curve (MRC) model, typically used to allocate processor caches [7, 20]. These models capture the memory usage characteristics of an application during a window of execution, termed as a *phase*. Phases are a third important memory resource usage characteristic of applications. Batson and Madison [8] 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. They observe that programs have marked execution phases 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 is that phases typically form a hierarchy [7, 24]. Hence, smaller phases with locality subsets are nested inside larger ones. Rothberg *et al.* [7] present such a working set hierarchy for several parallel scientific applications. However, the working set hierarchy does not capture the frequency of each phase and its impact on performance.

In contrast to previous work, we show that the amount of memory resource required by an application is more accurately reflected by a new metric that we introduce, the Effective Reuse Set Size (ERSS). Further, memory provisioning requires the identification of all the phases of an application and their durations to ensure that dominant phases are resident in a sufficiently fast level of the memory hierarchy. An important aspect of memory usage not captured by any existing model is the rate of memory usage by an application. The *Reuse Rate* and the *Flood Rate* significantly determine the amount of memory resource required by an application. When multiple applications share a common memory resource, these rate parameters dictate the amount of memory available to each application. Hence, memory provisioning in a consolidated virtualized environment requires significant refinements to existing memory usage models for them to be applicable.

5.2. Mechanisms to build memory usage models

Several techniques have been proposed to identify the optimal working set during program execution. Carr *et al.* [25] combine the local working set with a global clock to design WS-Clock virtual memory management algorithm. Ferrari *et al.* refine the classical working set to variable interval size, where the size of the interval is controlled using three parameters – the

minimum interval size M, the maximum interval size L, and the maximum page fault Q [26]. The page-fault frequency (PFF) algorithm [27, 28] uses the PFF concept to determine the resident set size, enlarging it if the PFF is high and shrinking it otherwise. There are two popular approaches to infer the MRC for an application. The first approach creates the MRC statically by running the application multiple times, typically in a simulator with different memory resource allocations [7, 24, 29]. A second approach uses a memory trace and a LRU stack simulator [30] to infer the MRC. MRCs in filesystem caches have been estimated using ghost buffers [31, 32]. Dynamically estimating cache MRCs is much more challenging and estimation methods exist only on specific platforms [20, 33]. Detecting phase transitions for taking reconfiguration actions has been explored before. The Dynamo system [10] optimizes traces of the program to generate fragments, which are stored in a fragment cache; an increase in rate of fragment formation is used to detect phase transition. Balasubramonian *et al.* [34] 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 [9].

All of the existing mechanisms to infer the memory resource usage or identify phases operate at the hardware or operating system level and make intrusive changes. With virtualization-driven consolidation, virtual machine images running applications cannot be modified by the data center system administrators who deploy them. In our work, we develop new techniques that can infer a more accurate memory usage metric — the ERSS — for various phases at the 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.

6. Conclusion

We have presented the Generalized ERSS Tree Model that addresses two significant gaps in our current understanding of application resource usage characteristics for the multi-level memory hierarchy. First, the model characterizes the memory resource usage of an application for its entire lifetime with a high degree of accuracy. We have presented a methodology to build model instances for arbitrary workloads on commodity hardware without making intrusive changes. Second, and more significantly, this model can be utilized for highly accurate provisioning of the memory hierarchy in a shared application environment. Our model comprehensively addresses the resource usage characteristics when competing workloads introduce non-trivial isolation requirements. It does so by explicitly characterizing the complex interaction between the resource reuse rates and flood rates of various workloads at each memory level. We have demonstrated that the model can be used to predict when isolation conditions are not satisfied as well as to determine resource provisioning requirements to ensure application performance isolation in shared environments. Finally, by experimenting with a wide mix of applications, we have established the utility and accuracy of our model in practice.

Acknowledgments

This work was supported in part by the NSF grants CNS-0747038, CCF-0937964, and CNS-0821345 and by DoE grant DE-FG02-06ER25739.

References

[1] T. Kgil, D. Roberts, T. Mudge, Improving nand flash based disk caches, in: Proc. of ISCA, 2008.

[2] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, P. Sadayappan, Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems, in: Proc. of IEEE HPCA, 2008.
[3] A. Verma, P. Ahuja, A. Neogi, Power-aware dynamic placement of hpc applications., in: Proc. of ACM ICS, 2008.
[4] S. Kundu, R. Rangaswami, K. Dutta, M. Zhao, Application performance modeling in a virtualized environment, in: Proc. of the IEEE HPCA, 2010.
[5] A. Verma, P. Ahuja, A. Neogi, pmapper: Power and migration cost aware application placement in virtualized systems, in: Proc. of ACM Middleware, 2008.
[6] P. J. Denning, Working sets past and present, in: IEEE Tran. on Software Engineerng, 1980.
[7] E. Rothberg, J. P. Singh, A. Gupta, Working sets, cache sizes and node granularity issues for large-scale multiprocessors, in: Proc. of ISCA, 1993.
[8] A. P. Batson, A. W. Madison, Measurements of major locality phases in symbolic reference strings, in: Proc. of ACM Sigmetrics, 1976.
[9] A. S. Dhodapkar, J. E. Smith, Managing multi-configuration hardware via dynamic working set analysis, in: Proc. of ISCA, 2002.
[10] V. Bala, E. Duesterwald, S. Banerjia, Dynamo: A transparent dynamic optimization system, in: ACM SIGPLAN, 2000.
[11] H. Huang, P. Pillai, K. Shin, Design and implementation of power-aware virtual memory, in: Proc. of the Usenix ATC, 2003.
[12] P. J. Denning, Thrashing: Its causes and prevention, in: Proc. of AFIPS Fall Joint Computer Conference, 1968.
[13] NAS Parallel Benchmarks (NPB), http://www.nas.nasa.gov/resources/software/npb.html.
[14] S. Akyurek, K. Salem, Adaptive Block Rearrangement, Computer Systems 13 (2) (1995) 89–121.
URL `citeseer.ist.psu.edu/akyurek92adaptive.html`
[15] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, V. Hristidis, BORG: Block-reorganization for self-optimizing storage systems, in: Proc. of USENIX FAST, 2009.
[16] L. Useche, J. Guerra, M. Bhadkamkar, M. Alarcon, R. Rangaswami, Exces: External caching in energy saving storage systems, in: Proc. of the IEEE HPCA, 2008.
[17] J. Axboe, blktrace user guide (February 2007).
[18] Open Source, Dyninst: An Application Program Interface (API) for Runtime Code Generation, Online, http://www.dyninst.org/.
[19] L. Soares, D. Tam, M. Stumm, Reducing the harmful effects of last-level cache polluters with an os-level, software-only pollute buffer, in: IEEE MICRO, 2008.
[20] D. K. Tam, R. Azimi, L. B. Soares, M. Stumm, Rapidmrc: Approximating l2 miss rate curves on commodity systems for online optimizations, in: Proc. of ASPLOS, 2009.
[21] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, An updated set of basic linear algebra subprograms (blas), ACM Trans. Math. Soft. 28 (2) (2002) 135–151.
[22] Standard Performance Evaluation Corporation, Specpower_ssj2008, Online, http://www.spec.org/power_ssj2008/.
[23] P. J. Denning, The working set model for program behavior, Communications of the ACM 11 (5) (1968) 323–333.
[24] M. Karlsson, P. Stenstrom, An analytical model of the working-set sizes in decision-support systems, in: Proc. of ACM SIGMETRICS, 2000.
[25] R. Carr, J. Hennessy, WSCLOCK – A simple and efficient algorithm for virtual memory management, in: Proc. of ACM SOSP, 1981.
[26] D. Ferrari, Y.-Y. Yih, VSWS: The variable-interval sampled working set policy, in: IEEE Trans. on Software Engineering, 1983.
[27] W. Chu, H. Opderbeck, The page fault frequency replacement algorithm, in: Proc. of AFIPS Conference, 1972.
[28] R. K. Gupta, M. A. Franklin, Working set and page fault frequency replacement algorithms: A performance comparison, in: IEEE Transactions on Computers, 1978.
[29] S. Woo, M. Ohara, E. Torrie, J. P. Singh, A. Gupta, Methodological considerations and characterization of the splash-2 parallel application suite, in: Proc. of ISCA, 1996.
[30] Y. Kim, M. Hill, D. Wood, Implementing stack simulation for highlyassociative memories, in: Proc. of ACM SIGMETRICS, 1991.
[31] J. Kim, J. Choi, J. Kim, S. Noh, S. Min, Y. Cho, C. Kim., A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references, in: Proc. of USENIX OSDI, 2000.
[32] R. Patterson, G. Gibson, E. Ginting, D. Stodolsky, J. Zelenka, Informed prefetching and caching, in: Proc. of ACM SOSP, 1995.
[33] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, S. Kumar, Dynamic tracking of page miss ratio curve for memory management, in: Proc. of ASPLOS, 2004.
[34] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, S. Dwarkadas, Memory hierarchy reconfiguration for energy and performance in general purpose architectures, in: IEEE MICRO, 2000.