Merlin: Application- and Platform-aware Resource Allocation in Consolidated Server Systems

Priyanka Tembey
Qualcomm Research Silicon Valley
ptembey@qti.qualcomm.com

Ada Gavrilovska Karsten Schwan
Georgia Institute of Technology
ada,karsten.schwan@cc.gatech.edu

Abstract

Workload consolidation, whether via use of virtualization or with lightweight, container-based methods, is critically important for current and future datacenter and cloud computing systems. Yet such consolidation challenges the ability of current systems to meet application resource needs and isolate their resource shares, particularly for high core count or 'scaleup' servers. This paper presents the 'Merlin' approach to managing the resources of multicore platforms, which satisfies an application's resource requirements efficiently – using low cost allocations – and improves isolation – measured as increased predictability of application execution. Merlin (i) creates a *virtual platform* (VP) as a system-level resource commitment to an application's resource shares, (ii) enforces its isolation, and (iii) operates with low runtime overhead. Further, Merlin's resource (re)-allocation and isolation methods operate by constructing online models that capture the resource 'sensitivities' of the currently running applications along all of their resource dimensions. Elevating isolation into a first-class management principle, these sensitivity- and cost-based allocation and sharing methods lead to efficient methods for shared resource use on scaleup server systems. Experimental evaluations on a large core-count machine demonstrate improved performance with reduced performance variation and increased system throughput and efficiency, for a wide range of popular datacenter workloads, compared with the methods used in prior work and with the state-of-art Xen hypervisor.

Categories and Subject Descriptors D.4.2 [*Operating Systems*]: Performance

Keywords Virtualization, Performance isolation

SoCC '14, 3-5 Nov. 2014, Seattle, Washington, USA.

ACM 978-1-4503-3252-1.
http://dx.doi.org/10.1145/2670979.2670993

1. Introduction

Datacenter servers are routinely used to run multiple disparate application workloads. Analysis performed by Google show, for instance, components (i.e., processes) of up to 19 distinct applications to be co-deployed on a single multicore node [13]. Virtualization and container technologies both enable and further encourage this trend, increasing platform utilization via higher levels of workload consolidation.

Multicore platforms and their current hypervisor- or OS-level management methods continue to be challenged in their ability to meet the performance needs of multiple consolidated workloads. This is because an application's performance is determined not only by its use of CPU and memory capacities, which can be carefully allocated and partitioned [1, 10, 35], but also by its use of shared platform resources, which are neither easily assessed nor controlled, including caches, memory bandwidth, and I/O resources. Further complications arise from application *elasticity* concerning their multi-dimensional resource needs [23], made more complex by the fact that changes in one resource dimension can also indirectly perturb resource use along other dimensions. For instance, a change in compute and/or memory resource shares for a web-server to service a volume spike [33] may lead to an associated increase in its cache and memory bandwidth usage. This may in turn *cause arbitrary hurt* to other applications sharing cache and memory bandwidth resources with the web-server.

With phase transitions [11], load fluctuations in both short and long time-scales [6, 23], or changes in relative importance compared to its co-runners being commonplace in cloud workloads, it is critical for system-level resource management, therefore, to continuously and efficiently deal with isolation and sharing. It must be aware of *(i) inter-resource dependencies within the platform architecture, (ii) how these resources are shared across applications that have certain levels of 'sensitivities' to each of the shared resource types*, and *(iii) it must be efficient, by effectively using resources to meet different applications' needs and by incurring low costs in reconfiguring their 'elastic' resource shares*. Consider, for instance, NUMA-aware scheduling [2, 16], which because it attempts to *always* allocate application memory

local to its CPU placement, causes dynamic thread migration to be inevitably followed by page memory migrations to keep accesses local. Since page migration is costly, as it consumes both memory bandwidth and CPU resources, unnecessary migrations lead (i) to inefficient, costly platform management, and (ii) may even cause hurt to other applications that are sharing the resources consumed during migration.

In summary, resource management in operating systems or hypervisors must manage resource shares for individual, elastic applications along 'all' of their resource dimensions, and they must do so in ways that consider the runtime costs of meeting their resource needs and understand the performance implications on other running applications due to indirect inter-resource dependencies.

Prior research on server resource management has principally focused on isolation, particularly for memory resources [2, 17, 18, 25, 39]. Such work aims to increase overall system throughput, but it does not consider individual application needs, nor does it take into account application-specific sensitivities to the resources being shared. Recent work addressing foreground/background processing seeks to prioritize a latency-sensitive workload over other applications [21, 37], but has no methods for explicitly dealing with per application sensitivities and inter-resource dependencies. Yet it is precisely those methods that are needed to ensure that say, multiple Hadoop jobs of differing importance are co-run efficiently, along with additional web server-based applications, and others.

The `Merlin` resource allocator presented in this paper meets the requirements articulated above. For multicore server platforms, Merlin (1) manages application resource shares for all resource dimensions – compute, memory, and memory bandwidth (last-level caches (LLC), memory controllers (MC), and interconnects (IC)). (2) Aware of each application's resource mappings, when resource shares must be reconfigured, Merlin offers arbitration methods that are guided by online models that capture both (i) the cost of reconfiguration and (ii) the multi-dimensional sensitivities of other applications to the resource types affected by reconfigurations. Stated intuitively, when reconfiguring an application, Merlin's management methods seek improved platform efficiency through 'effective' actions that choose the 'lowest cost' allocations that also cause the 'least amount of hurt' to all other applications. (3) Merlin can also accommodate differences in application importance and priority.

Key elements of Merlin and its approach are as follows.

1. *Efficiency*: a multi-resource dimensional manager for multicore servers that controls application resource shares, takes into account their resource sensitivities, and the costs of management actions.

2. *Virtual Platforms* (VPs): a per-application resource share abstraction for its multiple resource dimensions: CPU, memory, and memory bandwidth resource types. Merlin manages and isolates application VPs.

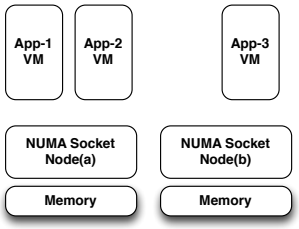

Figure 1: Two applications contained within individual VMs running on a 2-socket NUMA platform

3. *Model-guided management*: as the theoretical underpinnings for online resource management, observation-based models and metrics capture the dynamic resource needs of application VPs at system level, namely, for compute, LLC, MC and IC resource types. Using per-application performance counters, they also capture the costs of management, by assessing runtime-varying application sensitivities along with knowledge of the innate costs of various management actions.

4. A *hypervisor-level implementation*: a Xen-based realization of Merlin and VPs is validated experimentally for an Intel x86-based multicore platform. Evaluations use representative server workloads, including a commercially used key-value store, a streaming server, and multiple MapReduce application tasks. Performance results demonstrate that with Merlin, multicore resources in consolidated systems can be shared with low levels of performance degradation across all applications hosted (up to 25%), while efficiently using platform resources. This, then, directly impacts the approach's ability to support high degrees of workload consolidation.

In the remainder of this paper, Section 2 demonstrates how both cost and sensitivity driven management approaches are needed to manage platforms more efficiently and meet application performance needs. This is followed by a description of Merlin's management architecture and its online models in Section 3. Section 4 describes the system implementation, followed by experimental evaluations in Section 5. Related work and conclusions appear at the end.

2. Motivating the Merlin Approach

Consider the two-node multicore server shown in Figure 1. As seen in the figure, two applications, App-1 and App-2, are allocated CPU and memory resources on Node-(a), while App-3 is hosted on Node-(b). App-1 and App-2 also share the LLC and memory subsystem of Node-(a). If App-1 needs more CPU resources, we compare Merlin's approach against three other widely used allocation policies for NUMA mul-

ticore systems (henceforth, we refer to the NUMA node that hosts all of an application VM's VCPUs as its 'home' node, and all other nodes as 'remote'; Node-(a) is hence App-1's home node):

(1) Local policy: always allocates memory pages on the application's home node. In order to avoid remote access latencies, if a CPU is allocated on a remote node, this policy also migrates memory pages to the remote node. Representative of previous NUMA-aware allocation methods [2, 16], with this policy, when App-1 requests an increased CPU allocation and if a remote CPU is allocated on Node-(b), then its memory is also migrated to Node-(b). The policy's default memory migration actions, therefore, may incur wasteful and costly migrations if application performance is not actually sensitive to remote access latencies.

(2) Random policy: allocates compute and memory resources randomly amongst nodes based on availability. This policy is unaware of platform topology and application sensitivities to different resource types. It represents the one used by the current Xen CPU scheduler balancing CPU queues based on system load. In the example of Figure 1, the random policy may choose any CPU available on Node-(a) or Node-(b).

(3) Low cost policy: always allocates compute and memory resources using the 'lowest cost' method first. In this policy, the software methods used to allocate and/or reconfigure an application's resource shares are ranked in order of their increasing resource consumption cost as follows: (i) increasing the CPU caps of an application's VCPUs; (ii) allocating a local CPU on a 'home' node; (iii) allocating a remote CPU and further migrating the application VCPUs to a remote node; and (iv) migrating an application's memory pages to a remote node. This policy is cost-efficient, but it may not always be effective, as it disregards applications' and their co-runners' sensitivities to shared resource types. For example, in Figure 1, App-1 will be allocated a local CPU, which may increase its cache usage and/or memory subsystem usage on Node-(a), thereby potentially hurting App-2 if it is sensitive to either of these resource types.

A concrete experiment demonstrating the effects of the different policies listed above runs a Matrix-Multiplication MapReduce code [27] (App-1), collocated with the Voldemort [20] key-value store (App-2), with each application hosted in a separate VM, and in a configuration as the one shown in Figure 1 (but eliding App-3). We derive app sensitivities in Section 3 using online metrics, but it is useful to note here that both applications are LLC-sensitive, and Voldemort is also sensitive to the MC resource type. Figure 2 depicts the performance levels seen for the two applications when given resources using the above policies, normalized to when running alone (the first bar in Figure 2). The error bars denote performance variability across 20 runs.

When Matrix-Multiplication needs more CPU during its execution, we first use the 'local' policy to allocate an additional local core on Node-1. However this leads to an increase in response time latencies for Voldemort, by almost 22%(second bar); and its response time variability increases to about 10%. This is caused by Matrix-Multiplication's indirect increase in LLC and MC usage on Node-1, to which Voldemort's performance is sensitive. In other words, the 'local' policy's sole consideration of Matrix-Multiplication's sensitivity ends up being hurtful to other applications. The same holds for the 'low cost' allocation policy (fourth bar in Figure 2), which by allocating the local CPU, causes similar performance degradation for Voldemort: it also ignores other applications and their sensitivities. Finally, the 'random' allocation policy (the third bars in the figure) may allocate a local or a remote CPU from Node-(a) or (b) with equal probability. When allocated a local CPU, Voldemort response times are hurt due to its sensitivity to increased memory intensity at Node-(a); when a remote CPU is allocated, however, this policy ends up hurting Matrix-Multiplication, as its VCPUs end up being divided between two the LLCs of Nodes- (a) and (b), thus hurting Matrix-Multiplication's cache sensitivity.

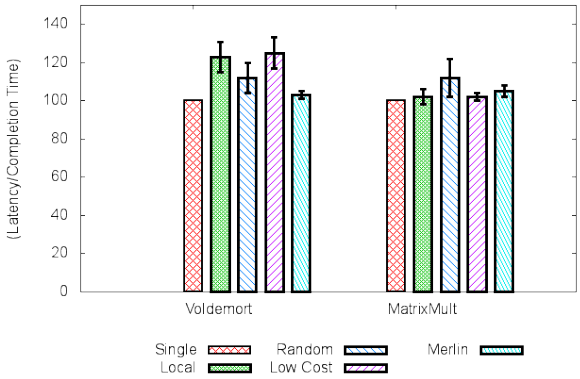

Figure 2: Reconfiguration of CPU resource shares for Matrix-Multiplication hurt Voldemort due to consequent increase in LLC and MC usage.

In contrast to the methods shown above, Merlin's allocation are much superior: their use results in the least performance degradation (∼3-5%) and variability (fifth bar). This is because (1) Merlin considers the multi-dimensional sensitivities of *both* applications, so avoids causing 'hurt' to Voldemort's LLC shares by allocating it a remote CPU and by migrating *all* VCPUs of Matrix-multiplication to Node-b. Further, (2) while Merlin incurs the cost of VCPU migration to a remote node, it avoids the costly subsequent migration of memory pages, by considering Matrix-multiplication's insensitivity to the MC resource type. Generalizing from this example, Merlin's resource management succeeds for multiple reasons. First, it operates across multiple resource dimensions, being aware of all inter-resource dependencies between compute, LLC, and MC/IC resources. Second, it accounts for collocated applications and their sensitivities

to shared resource types. Third, the costs of reconfiguration are considered, along with sensitivity information, so as to choose the reconfiguration that is 'right' for all applications. In the above example, for instance, choosing a remote CPU for matrix-multiplication is ideal: it uses the low-cost VCPU migration and considers both applications' sensitivities (thus 'fairly' degrading each). The remainder of this paper explains the methods used by Merlin for making desirable decisions like those above.

3. Merlin Resource Manager

Merlin represents the resources it maintains for specific applications – sets of VMs – as *virtual platforms* (VPs). Each application has its own VP, and it is Merlin's obligation to allocate and manage the resources of all current VPs. Specifically, arbitrating across different applications' resource requirements, Merlin (i) deals with their multiple resource dimensions of compute, memory, and memory bandwidth, (ii) accounts for the cost of management actions, and (iii) considers application sensitivities to different resource types. Generalizing on previous work [22, 36], Merlin's arbitration goal is: *to allocate compute, memory, and memory bandwidth resource shares, i.e., VP shares, to all applications, while minimizing the maximum difference between potential levels of performance degradation, i.e., 'unfairness', imposed on all applications.* When resources are oversubscribed in any or all dimensions, Merlin can also accommodate application priorities or weights when making allocation decisions, thus making it easy to specify the relative importance of some application vs. others. Stated more formally, for all applications 'i' and 'j' in the system, the following linear program captures this goal:

$$max \sum w_i u_i$$

$$w_i(1-(u_i)) - w_j(1-(u_j)) < \varepsilon, \forall i, j ...(1)$$

where, w_i represents the relative application weight, and u_i represents the normalized performance of $application_i$ relative to its ideal performance when it is not collocated with other applications ($\hat{u_i}$). u_i is a function of the application's VP, i.e., its resource shares at $time-epoch_k$ [25, 38]:

$$u_i = \frac{1}{\hat{u_i}} \gamma[CPU[k], Memory[k], Memory\_BW[k]] ...(2)$$

The term '(1 - u_i)' in Equation-(1) represents the applications degradation from its ideal performance $\hat{u_i}$. To maximize applications' performance and minimize their performance degradation, Merlin must assess (i) applications' resource usage, as a measure of their performance needs, (ii) their resource sensitivity, as a measure of the potential hurt that can be caused by resource reconfigurations, and (iii) further consider the cost of VP configurations, for efficiency. Merlin performs all such actions and carries out arbitration periodically: at the end of every time epoch 'k', Merlin arbitrates VP resource shares (Equation-2) for the next epoch, always minimizing the maximum weighted difference in degradation across all applications.

3.1 Tracking VP Resource Shares

Defining an application's performance in terms of its shares of all resources, including caches, memory, and interconnect bandwidth ideally requires these quantities to be precisely measured and controlled [12]. Since contemporary hardware does not offer such functionality, a technical contribution of our work are observation-based techniques that approximate an application's use of these resources. In our experimental Westmere NUMA platform, for instance, each physical core has an independent L1 and L2 cache, while the last-level L3 cache is shared among all cores in the same socket. These caches are inclusive, so an L2 miss will always access the L3. Merlin can therefore, approximate the L3 usage of an application as being equivalent to *L2misses - L3misses* (where cache miss values are all evaluated per 1000 instructions – i.e., MPKI). Further, using L3 MPKI as an indicator of bandwidth intensity (as also done in [2, 32]), Merlin then expresses the local and remote memory bandwidth intensity in a NUMA platform as *local L3 MPKI* and *remote L3 MPKI*.

Merlin also tries to understand the relative importance of one shared resource vs. another, by measuring the *ratio* of each application's cache vs. memory (MC/IC) usage. In other words, by considering this ratio, we can assess an application's relative *'sensitivity' to cache vs. to MC and IC contention* (as will be shown in more detail below). For each application, we measure its *Memory Factor* (MF), which is determined by the fraction of L2misses that end up as memory accesses, represented as *L3/L2 MPKI*. This metric can be generalized to other platforms as *LLC/(LLC-1) MPKI*, where 'LLC-1' denotes the previous cache in the hierarchy. Intuitively, a low value of MF denotes higher cache reuse, while a higher value denotes higher usage of MC and IC vs. less cache reuse, since a higher fraction of its misses are served as memory accesses.

Summarizing, an application's use of shared resources can be approximated by assessing the following two metrics: (i) L3 MPKI a measure of absolute MC/IC usage, and (ii) MF: a relative measure of cache vs. MC/IC usage. Also important (but not novel) are Merlin's use of CPU and memory utilization to denote the application VP's aggregate CPU and memory resource usage.

3.2 Assessing VP Sensitivities to Resource Types

An application can be termed 'sensitive' to its share of some resource type if a reduction in that share causes 'significant' performance degradation. Typically, such performance 'hurt' ((1-u_i)) is caused by other applications imposing on its sensitive resource shares, implying a less than ideal alloca-

tion of platform resources to these applications. Specifically, we use both (i) the MF value of an application and (ii) its LLC MPKI to ascertain an application's sensitivity to each of the shared resource points (cache, MC, IC) as follows:

- *Sensitivity to cache resource.* Applications with low MF value and relatively high memory intensity exhibit *high cache footprint reuse*. They are more susceptible to losing their cache-shares in the presence of co-running cache-intensive applications.
- *Sensitivity to MC/IC resources.* Applications with high MF values, e.g., close to 1, and large L3 MPKI values for their memory intensity, are *RL, MC, and IC-contention sensitive*. For SMP platforms, applications sensitive to MC contention will also be sensitive to FSB contention.
- *Insensitive applications* are those with low MF and L3 MPKI values: they use less of both caches and MC, IC and therefore, are relatively immune to hurt from these resource types.

To assess sensitivity to CPU usage, Merlin uses the following metric: *CPU_utilization / Allocated CPU*. To illustrate, consider a web-serving application allocated two physical cores, but with utilization equivalent to only one physical core; this equates its sensitivity to *0.5* toward the CPU resource. For the memory resource type, we devise a similar metric: *Memory_utilization / Memory pages initially allocated*; it is used to assess memory-related sensitivity.

Application classification. Given the experimental platform used in our work, there are three classes of memory intensity, based on (local/remote) L3 MPKI: 'low' having <2, 'medium' having > 2 and < 15, and 'high' having >15 L3 MPKI. MF values are also factored into three classes: low MF < 0.25, medium MF > 0.25 but < 0.6, and high MF > 0.6. We determine these platform-dependent thresholds by profiling them with well-understood benchmarks like the 'Stream' memory benchmark. Based on the runtime values of MF and L3 MPKI metrics and these a priori determined thresholds, Merlin measures application resource usage and sensitivities in every epoch. For instance, an application with MF$>$0.6 and having 'high' memory intensity is determined as 'highly MC/IC sensitive'.

3.3 Costs of Resource Reconfiguration

Reconfiguring the resource shares of an application's VP, incurs costs in terms of the resources consumed by reconfiguration actions. As stated in Section 2, Merlin ranks reconfiguration methods in terms of their increasing cost of resource consumption, which are (from lowest to highest): (1) increase or decrease CPU shares, by using CPU capping; (2) migrate the VCPUs of VMs within and across NUMA nodes, by allocating local vs. (3) remote node CPUs; and (4) migrate memory pages across nodes. This ranking is because of the absolute costs associated with each reconfiguration action: increasing or decreasing the CPU shares of an application costs only a few CPU cycles; migrating an application's VCPUs to another node requires performing VCPU state transfers and later moving cache lines to the remote node over the off-chip interconnect [19]; migrating an application's memory pages consumes memory bandwidth and memory subsystem resources of the source and destination memory controllers, memory buses, and interconnects (and the compute capacity used for memory-copying). In addition to such direct costs, Merlin also considers the fact that certain reconfigurations, like those in which memory pages are migrated, can impose hurt on the sensitive resource shares of other applications.

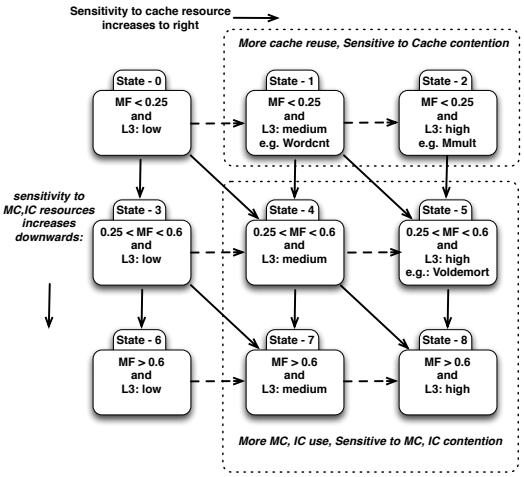

Figure 3: State Transition model capturing possible interference at shared resource points: Cache, MC, IC.

In summary, to maintain the goal stated in Equation-1, Merlin factors in both the cost of reconfiguration and the possible hurt caused by reconfiguration actions, the latter depending on the sensitivities of all applications in the system, We next describe how Merlin uses observation-based methods to determine the need for VP reconfiguration.

3.4 Triggers for Resource Reconfiguration

Sections 3.1 and 3.2 describe Merlin's use of system-level metrics to assess an application's current resource usage and resource sensitivities. On this basis, for all resource types, Merlin detects the dynamic, elastic nature of an application VP's resource requirements: via runtime tracking of its transitions in resource usage and sensitivities. For instance, for the CPU, when utilization nears the VP's actual allocated CPU shares (CPU_sensitivity > 0.8), the application is deemed as being highly sensitive to the CPU resource type. This hightened sensitivity also indicates to Merlin the need to allocate additional CPU capacity. To assess the need for additional cache and MC/IC resources, Merlin builds for each application a two-dimensional characterization of its MF metrics and absolute L3 MPKI values. This yields the 9-state transition matrix shown in Figure 3, based on the profile-based classification thresholds seen in Section 3.2.

Transitions between two states in Figure 3 denote an application's change in resource usage as well as a change in its resource needs. In particular, when an application transitions towards the 'right' (i.e., towards State-2,5,8), this signifies an increase in its absolute L3 MPKI values, i.e., increased MC/IC usage. When an application transitions 'along the downward dimension' of MF, this signifies an increase in MC/IC accesses, without a corresponding change in L2miss counts, thus indicating contention at the LLC and the need for more LLC resources. Upon detecting such a transition, Merlin attempts to re-allocate cache shares for cache-sensitive applications.

Transitions to the left and upward signify that cache demands have been met. Finally, 'downward' transitions cannot be detected for State-7,8 applications, as they are completely cache-insensitive. These states correspond to being MC/IC- and RL-sensitive. Merlin detects possible MC/IC resource needs by periodically aggregating the memory load information (L3 MPKI) of all VPs sharing each NUMA node's MC and IC link. For our experimental server system, if the total L3 MPKI of a NUMA node is greater than 100 (this value is derived from the Stream benchmark's L3miss counts), Merlin takes that as an indication of the application's VP requiring additional MC/IC resources.

Upon identifying these triggers when an application is highly sensitive to a certain required resource type, Merlin is responsible for determining the right reconfiguration action that meets the application VP's resource needs. By considering both the cost-related metrics (Section 3.3) and the sensitivities (Sections 3.1 and 3.2) of all applications to all resource types affected by the reconfiguration, Merlin chooses *the lowest cost reconfiguration option that also causes the least hurt to all applications affected by the reconfiguration*, hence maintaining its arbitration goal seen in Equation-(1).

3.5 Choosing the Right Reconfiguration

Figure 4 illustrates arbitration steps and their timeline. We use the term 'application-set (S)' to denote the set of applications whose resource demands must be met in the current arbitration epoch (t). At time instant (t), Merlin considers a single requesting VP's (App-1's) resource re-allocation need (as determined using methods described in Section 3.4). Hence, the application set (S) contains App-1 at time (t). Merlin first assesses the availability of free resource capacity on the application's local node, as local resource allocation is low cost and causes least hurt to applications in set (S), in this case - App-1. For the CPU resource type, Merlin allocates an additional local CPU to App-1's VP. For the cache and MC/IC resource types, Merlin applies the lowest cost allocation, by 'freeing up' local cache and MC/IC resources by capping the CPU usage of other corunning applications, as an indirect method of limiting their cache and MC/IC usage.

Unfortunately and as seen in Section 2, choosing local free resources may be hurtful to co-running applications. For example, an additional core allocated to App-1's

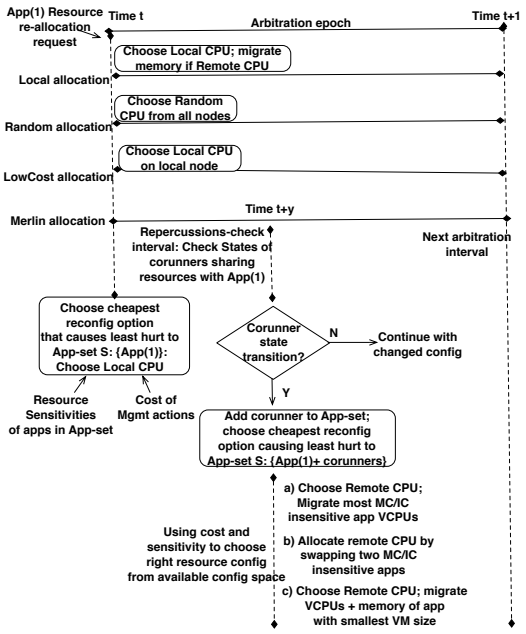

Figure 4: Merlin arbitration steps and timeline

VP may lead to increased LLC/MC/IC usage, thus hurting co-running applications' resource shares. Capping the CPU shares of applications sensitive to the CPU resource type may hurt their performance. To monitor for potential 'hurt' caused to co-running applications, after a reconfiguration in an epoch, Merlin denotes a time-interval (y) within its next arbitration epoch (t+1) as a 'repercussion interval' phase. During this phase, any sensitivity state transitions (see Figure 3 and Section 3.4) detected for any corunning applications are attributed by Merlin to the reconfiguration actions performed at the beginning of the current epoch. Only if there are no transitions, the reconfiguration action is deemed successful, and Merlin proceeds to the next arbitration epoch (t+1). Else, when there are state transitions that denote 'hurt' caused to other corunners, Merlin adds those applications to the set (S) of the current epoch and reassesses its reconfiguration, as explained next.

Consider Figure 1. If App-2 is 'hurt' due to resource-reallocation for App-1, the set of applications (S) to be considered for reconfiguration becomes (App-1, App-2). For this set, Merlin attempts to apply the next cheaper reconfiguration option: allocating remote CPU on a remote node, if available (Node-b in the figure), and this reconfiguration is applied to the more MC/IC insensitive application from set (S), so as to minimize hurt caused to the migrated application, while also keeping the cost of reconfiguration low (for MC/IC insensitive applications, memory migration can be avoided). If however, there is no remote CPU available, Merlin next attempts to find an MC/IC-insensitive application on a remote node (e.g., App-3 in Figure 1). If App-3 is also MC/IC insensitive, Merlin 'swaps' App-3's CPU

resources with the chosen MC/IC insensitive application's CPUs in set (S), to create the next 'ideal' reconfiguration minimizing 'hurt' and 'cost' for all applications in (S). LLC re-allocation for an application is performed similarly, using the above steps as allocating CPUs on a remote node is equivalent to allocating LLC shares on that node. However, if none of the applications in set (S) are MC/IC insensitive, and hence, unsuitable for VCPU migration alone, Merlin uses the final reconfiguration option: migrating memory pages. Using further migration-related optimizations, such as choosing the application in set (S) having fewer memory pages to migrate, and 'phased-out migration' (as seen next), Merlin again attempts to reduce the overall 'hurt' and 'cost' incurred by page migration.

MC/IC Re-allocation. The need for memory-reallocation may arise when an application's VP has 'remote' memory that must be reconfigured to 'local', because of the application's high MC/IC sensitivity, or when the total MC/IC miss counts at a NUMA node are > 100 (see Section 3.4). Memory re-allocation is the most expensive management option, so Merlin uses certain cost-based optimizations to attempt to minimize the 'hurt' caused to other applications due to the operation's high memory intensity and sensitivity.

First, if a VM's memory must be migrated from one memory node to another, in order to resolve contention at the memory controller resource between two or more applications, Merlin will migrate the application VM with the smaller memory footprint. Second, Merlin executes page migration amongst NUMA nodes in cycles of migrating and copying 1024 pages per iteration. These iterations may be further phased out in time if applications at the source and/or the destination memory controllers are highly memory intensive and are also sensitive to the memory subsystem resources (MC/IC). Because this technique may hurt the performance of the application being migrated, Merlin resorts to using application priorities to decide which of the applications can be hurt. Finally, if such hurt is not acceptable, and memory migration being the final reconfiguration option available, Merlin may resort to notifying higher-level schedulers to migrate an application to other servers in the datacenter.

Summarizing, Merlin's resource allocation and arbitration methods factor in both the cost of management operations and the sensitivity of applications to all resource types, to make reconfiguration decisions to maintain its goal defined in Equation-(1). Finally, considering the scalability of Merlin to high core count platforms, consider an experimental platform of 'n' NUMA nodes. Here, the number of different CPU and memory configurations for an application VP are on the order of $O(n^2)$. Merlin prunes this decision space by discarding the wasteful configurations (based on cost) and the hurtful ones (based on sensitivity). This helps it choose more effective configurations for its workloads and more efficiently manage application VPs.

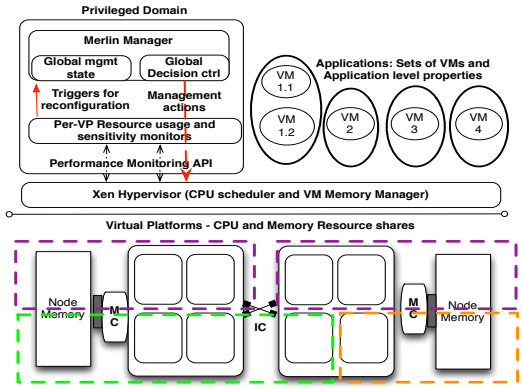

Figure 5: Merlin and VP Implementation in Xen

4. Implementation

Figure 5 depicts the different implementation components of Merlin and its virtual platforms, realized in the Xen hypervisor. The Merlin Manager is a user-level multi-threaded process in the privileged management domain, Dom0. It carries out the following management functions.

Virtual Platform Creation. Applications are deployed within sets of VMs, illustrated in Figure 5 for 1 multi-VM and 3 single-VM applications. Each application's initial allocation requirements are specified in a configuration file that states its *CPU and memory requirements* as numbers of VCPUs and memory size, respectively. Merlin uses this data to allocate requested CPU and memory resource shares to the application VM(s), always attempting to allocate local CPU cores and memory, by sequentially traversing sockets. CPU sharing is controlled by pinning VCPUs to independent physical cores within a socket. A modified NUMA-aware version of the Xen memory manager [28] *confines* VM memory pages to particular NUMA nodes. Our memory allocator also supports a *percentage-split* of a VM's memory footprint across a configurable number of multiple NUMA nodes. The initial mapping of application VPs to shared resources such as LLCs and MC/IC resource types in the platform is hence known to Merlin at the time of VP creation. After creating the application VMs and finalizing their resource allocations, Merlin spawns a per-VP thread in Dom0 to periodically inspect hardware performance counters for monitoring VPs' resource shares.

Monitoring application resource shares and sensitivities. Per-VP user-level threads in Dom0 are responsible for monitoring periodically the application resource usage and sensitivities, using black-box monitoring techniques. Per-VP monitor threads periodically (every 1 second) collect relevant performance counters, including unhalted CPU cycles, instructions retired, L2 cache misses, local L3 cache misses resulting in local node memory accesses, and remote L3 cache misses resulting in remote node memory accesses. Per-VCPU monitoring history of up to one second is stored in Xen kernel data structures that are then exported to the user-level (Dom0) monitors via newly introduced Xen hypercalls. The per-VP monitor also calculates L3 MPKI and

MF values and uses the state model described in Section 3.4 to detect state transitions that indicate potential VP need for specific resource types. These are then conveyed to the Merlin manager process using shared memory message queues.

Merlin management. After allocating initial VP resource shares and their monitors, the Merlin Manager polls for inputs from VP-monitors in case resource need is detected. This configurable polling period is currently set it to 1 second. Next, using the arbitration techniques described in Section 3.5, Merlin attempts to choose the lowest cost management method for reconfiguration that will also cause the least amount of hurt to 'all' applications. The management mechanisms are implemented as follows. Merlin caps CPU shares by adjusting Xen CPU scheduler parameters for the application VM's VCPUs: this parameter increases or decreases the percentage value of running time for the VCPUs in each Xen scheduling epoch, hence affecting the VP's CPU usage. The default capping percent is 10% per epoch. In order to migrate VCPUs, Merlin changes the CPU assignment of the application VM's VCPUs using Xen hypercalls. Finally, we have implemented new functionality in Xen with which Merlin can live-migrate and copy VM memory pages in batches of 1024 pages between NUMA nodes. Each such migration round can be delayed by up to 1 second in order to account for MC/IC sensitivities of other application VPs.

Recall that Merlin also keeps track of per NUMA node L3 MPKI. Currently, if this value exceeds 100 for a NUMA node, Merlin interprets this as need for additional MC/IC resource shares. The implementation of this functionality consults per-VP monitors of the relevant VPs sharing the NUMA node for their L3 MPKI counts, with subsequent reconfigurations using the MC/IC reallocation algorithms described in Section 3.5.

5. Experimental Evaluation

The evaluation of the Merlin resource allocator seeks to answer two key questions.

1) How does the use the cost and sensitivity related metrics improve state-of-art resource management, and how important are they for choosing more 'effective' reconfigurations for applications?

2) Are platforms managed with Merlin's methods 'better-managed' systems i.e., managed effeciently to obtain high application performance with good isolation and low overheads?

5.1 Experimental Methodology

Merlin's allocation methods are evaluated on the experimental platform used is as described in Section 2: a 32-core Westmere platform with 128GB physical RAM, with both its CPUs and memory divided equally amongst 4 NUMA nodes, each with a 24MB LLC. Representative cloud workloads are constructed from application mixes containing both latency-sensitive and batch applications [13]: (1) the latency-sensitive Voldemort key value store, co-running with throughput-intensive mapreduce codes [27], and (2) latency- and throughput-demanding data streaming workloads [7] co-running with throughput-oriented data-mining benchmarks [24]. With resource demands dynamically varied along multiple dimensions, Merlin is compared against the 'local', 'random', and 'low-cost' representative policies mentioned in Section 2. Merlin's benefits and 'effectiveness' are assessed by observing performance degradation for all applications in the workload mix, along with other system-level metrics like observed CPU and memory utilization, the number of VCPU and memory migrations undertaken, and the resources consumed, L3 MPKI as well as the Memory Factor (MF) values of applications. A *Platform Efficiency* metric computed for each allocation policy demonstrates the efficiency and effectiveness of Merlin's management operations. Performance degradation results are reported as *h-spread* distributions.

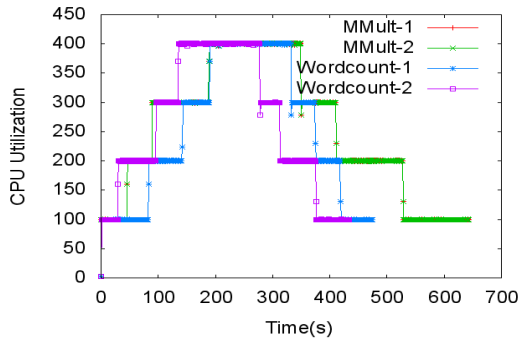

Figure 6: Variable CPU demand of Mapreduce codes

5.2 Importance of Multi-dimensional Sensitivity

This section's experiments validate that *multi-dimensional sensitivity knowledge enables resource management to choose the 'right' allocations for 'all' applications, and to do efficiently by also reducing the number of costly operations like memory migration.*

5.2.1 Considering Multi-dimensional Sensitivity when Allocating CPU Resources.

In order to demonstrate how allocation policies differ in allocating additional dynamic CPU capacity, our first experiment uses with the first workload mix: (1) Voldemort key value stores, collocated with mapreduce codes. This mix is representative of workloads with a latency sensitive application and parallel batch workloads collocated together. However, unlike previous work [21], Merlin does not assume that the latency sensitive application is the only foreground workload. Instead, Merlin makes cost and sensitivity related considerations for 'all' applications in managing their dynamic allocations along 'all' dimensions. In the first experiment, we execute Voldemort key value store within two virtual machines, each with 2 GB of physical memory, and 2 VCPUs

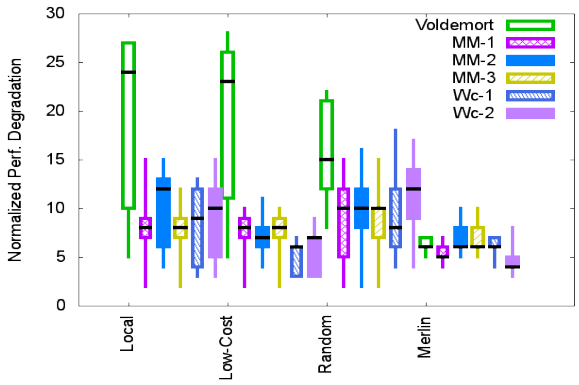

Figure 7: Considering multi-dimensional sensitivity is important when reallocating CPU shares

each. An external machine hosts a Voldemort client which generates key-value requests (with a value size of 1MB) for the Voldemort server. Voldemort server application is collocated with 3 copies of Matrix Multiplication codes, each deployed in separate VMs of 4 VCPUs each and 4GB each; and 2 copies of Wordcount codes also deployed in separate VMs of the same configuration. Each Matrix-Multiplication and Wordcount VM are allocated 2 physical cores each initially, and Dom-0 executes alone in Node-0 of the four-node Westmere platform (see Section 2). We execute the application-mix choosing different startup order of applications every run so as to not bias colocation on the same NUMA node.

Figure 6 shows how the Mapreduce codes (Matrix-multiplication and Wordcount) vary their CPU demand over time brought about by the non-uniform distribution of input data amongst its compute threads. Due to initial allocation of two physical cores, when the CPU demand increases beyond *200*, the application's VP monitor thread will request an additional core (Section 4).

Figure 7 shows the performance degradation experienced by the application mix depicted via the *h-spread* of the degradation amounts, and Table 1 shows additional metrics that help evaluate the effectiveness and efficiency of each allocation policy studied. Total number of VCPU and memory migrations shown in Table 1 depict the total number of reconfigurations that each allocation policy performed while re-allocating resources to applications, while CPU utilization is the average overall CPU usage observed during the execution. It is important to note that the CPU usage also includes CPU usage attributed to the management operations such as performing memory migrations. Table 1 also introduces a new metric, Platform Efficiency (PE), which we define as the *(normalized performance improvement for all applications / normalized total CPU utilization)*. In our evaluation, performance improvement is the inverse of the performance degradation values shown in Figure 7 (e.g., for Voldemort in the local case it is evaluated as 1/1.23, and in the Merlin-case, it is evaluated as 1/1.06, etc.). The CPU

Table 1: Platform Efficiency (PE) and other metrics for each allocation policy.

Policy	CPU-Util	No. of VCPU Migrations	No. of Memory migrations	P.E.	L3 MPKI	CPI
			Mix-1			
Local	2370	28	7	2.20	165	3.172
Low-Cost	2140	22	0	2.53	135	2.556
Random	2150	42	0	2.48	145	2.568
Merlin	2190	20	2	2.70	108	2.078
			Mix-2			
Local	1985	20	5	3.17	140	3.281
Low-Cost	1900	0	0	3.28	125	2.476
Random	1920	38	0	3.23	147	2.654
Merlin	1925	12	0	3.39	117	1.989
			Mix-3			
Local	2390	36	12	2.28	187	3.356
Low-Cost	2240	20	0	2.41	167	2.842
Random	2250	22	0	2.38	178	2.756
Merlin	2285	20	5	2.42	134	2.439

utilization shown in Table 1 is normalized to 2400, for 24 cores of three NUMA nodes used for running application VMs. A higher platform efficiency metric for a resource allocation policy indicates (a) improved performance per unit of resource (CPU) usage; hence demonstrating (b) improved ability to deal with dynamic resource allocations by choosing both *cheaper* and more *effective* reconfiguration options which cause *less overall hurt*. Finally, we also show the L3 MPKI and CPI (Cycles per instruction) metric to show respectively the shared LLC/MC/IC behavior and the absolute instruction performance in each policy case. Both these metrics are affected by management operations of the policies: as VCPU and memory migrations also trigger L3misses, lower L3misses and CPI values are demonstrative of fewer VCPU and memory migrations. Next, we make the following key observations from the performance and efficiency metrics.

1) Local policies may hurt other applications and overall platform efficiency. In the *Local* allocation policy case, whenever Matrix-multiplication is collocated with Voldemort codes and needs more compute capacity, its VMs are allocated CPU from the local node, in order to preserve their memory affinity to local accesses. However, this policy hurts collocated applications such as Voldemort (suffering up to 23% degradation in response times) due to increased memory intensity at the node cache and memory controllers, resources that Voldemort is sensitive to. Next, when local compute resources are unavailable, the *local* policy allocates re-

mote CPU (on remote node), however in addition also migrates the application's pages to the remote node. As seen in Table 1, local policy allocation performs 7 VM memory migrations, and 28 VCPU migrations during the execution run; the number of memory migrations being the highest from amongst all policies. Some of these migrations may be superfluous as for e.g., both matrix-multiplication and Wordount are insensitive to remote memory latency, and hence do not benefit from local memory accesses. The cost of these superfluous migrations is seen in both the increased resource consumption (up to 2 additional physical CPUs usage compared to other allocation policies in Table 1), and increased variability (up to 13%) in performance degradation across runs for all Mapreduce codes as seen in Figure 7.

2) Low-cost allocations may not always be most effective. In the *low-cost* allocation policy case, VP reconfigurations are performed based on their cost of allocation. As seen in Table 1, this policy incurs no memory migrations, as it uses VCPU migrations alone (within a node, and next, across nodes) unlike the local policy which also migrates memory pages. However, this policy also disregards sensitivity of other applications to shared resource types, hence causing performance hurt. Voldemort incurs up to 22% degradation in response time latencies when collocated with Matrix-multiplication codes. Though *low cost* allocation policy incurs lower costs of reconfiguration (observed via lower CPU usage in Table 1 vs. local policy , and lower performance variability across applications in Figure 7), the degradation caused to Voldemort is substantial, and is caused due to disregard for its resource sensitivities.

3) Multi-dimensional sensitivity and cost knowledge guides Merlin toward better reconfiguration decisions. Merlin considers both sensitivity and costs of reconfiguration to make re-allocation decisions. When Matrix-multiplication is collocated with Voldemort and needs more CPU, it first allocates local CPU. However in the 'repercussions' time phase (Section 3.5), it observes that Voldemort undergoes a state-transition to become 'highly' MC/IC sensitive due to increase in L3 MPKI brought about by increased memory intensity of Matrix-multiplication. This leads Merlin to choose a remote CPU on a remote node for Matrix-multiplication codes. Merlin further incorporates the MC/IC insensitivity knowledge of Matrix-multiplication to avoid costly memory migrations with VCPU migrations; unlike the *local* policy. In the event that remote CPU is unavailable, Merlin migrates the Wordcount application to Voldemort's node, hence creating CPU availability for Matrix-multiplication codes; Wordcount being less memory intensive tends to not hurt Voldemort. As seen in Figure 7, Voldemort's response times suffer only up to 6% degradation due to Merlin's actions, while the degradation of other applications is also limited to 10%. *This also satisfies our goal of minimizing the maximum difference in degradation amounts across all applications.* Table 1 shows that the platform efficiency metric is the highest for Merlin showing improved ability to manage dynamic resource allocations. The two memory migrations are invoked by Merlin only when the memory intensity (L3misses/1000inst) at a NUMA node increases beyond 100 L3misses (see Section 3.4), which may hurt applications sensitive to memory controller resources. For e.g., when Voldemort is collocated with two Matrix-multiplication codes, Merlin migrates one of the Matrix-Multiplication VMs to a remote node while also migrating its pages. As seen in Table 1, this leads to slight increase in CPU usage for Merlin, however traded by improved performance, variability and platform efficiency.

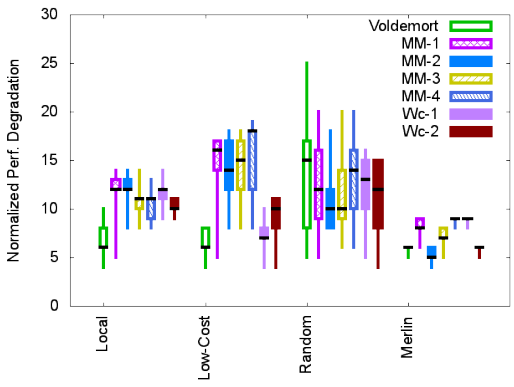

Figure 8: Reconfigurations due to cache and memory bandwidth re-allocation

Finally, as seen in Table 1, L3 MPKI and CPI metrics are the lowest for Merlin resource allocation policy, demonstrating absolute lower memory subsystem usage (due to invoking memory migrations only when needed, and better collocations suited to sensitivities of all applications), and therefore higher raw instruction throughput.

5.2.2 Considering multi-dimensional Sensitivity when Reallocating LLC and MC/IC Demands

Our next experiment demonstrates how Merlin compares against other allocation policies when applications need reconfiguration of their cache or MC/IC resource types. This experiment also shows the generality of Merlin's methods in managing multi-dimensional allocations. In this experiment, we use different VM configurations for the same application mix using the Voldemort key value store (split across 2 VMs with 2 VCPUs each), corunning with batch processing workloads using four copies of Matrix-Multiplication MapReduce codes (each in one VM of 2 VCPUs and 4GB memory) and two copies of Wordcount codes in separate VMs of four VCPUs and 4GB memory. Unlike the first experiment, all applications are allocated physical cores equal to their VCPUs. Figure 8 shows the performance degradation results of Merlin compared with 'local', 'low-cost' and 'Random' allocation policies for all applications, while 'Mix-2' in Table 1 shows efficiency metrics. In this mix, when Voldemort is collocated with a plurality of Matrix-multiplication codes,

the per-VP monitor observes a 'downward' transition for an initially cache and MC/IC sensitive Voldemort application. This is interpreted as 'need' for more cache resources (Section 3.4), and all allocation policies use their respective algorithms to allocate more LLC to Voldemort's VP.

'Local' policy migrates one Matrix-Multiplication VM to another remote node, also migrating its memory. Further migrations for Matrix-multiplication VMs may occur if collocated with other Matrix-multiplication codes to alleviate contention for cache resource, an ideal configuration being collocation with Wordcount instead (Wordcount is less cache-intensive than Matrix-multiplication). As seen in Figure 8, these migrations lead to performance variability for Matrix-multiplication, and also Voldemort and Wordcount due to memory migration overheads at source and destination memory controllers. An indication of these overheads can also be seen in Table 1 for 'Mix-2' where Local policy has the highest L3 MPKI amounts, and highest CPU usage needed to migrate memory pages. Local policy is also the least efficient shown by its lowest P.E metric. *It is interesting to observe that despite having lower performance variability than 'low-cost' and 'random' policies as seen in Figure 1, the local policy is still least efficient due to its high resource management costs. This makes a case for cost-aware resource management.* Next, 'low-cost' policy uses CPU-capping of Matrix-multiplication VMs as the cheapest method to indirectly free up cache usage for Voldemort VP. However, though this leads to low performance degradation for Voldemort, it causes significant hurt to Matrix-Multiplication VMs. *This hence shows that ignoring CPU sensitivity of Matrix-multiplication is not an effective policy, and makes a case for 'sensitivity-aware' resource management.* Random policy chooses a random node to migrate Matrix-multiplication VCPUs. This causes variability for all applications; Mapreduce codes are hurt due to their VCPUs separated across multiple caches hurting their cache sensitivities, while Voldemort response times are hurt (by up to 23%) when collocated with Matrix-multiplication codes disregarding its cache and MC/IC sensitivity. Finally, Merlin does not cap CPU for Matrix-multiplication accounting for its CPU sensitivity. It migrates Matrix-multiplication VCPUs to a remote node instead, while not migrating its pages due to knowledge of its MC/IC insensitivity. An ideal configuration involves one of the Matrix-multiplication and Wordcount code VCPUs to be placed remote. As seen in Figure 8, this causes some hurt to MM-4 and WC-1, however the degradation amounts are limited to 9%. As seen in Table 1, no superfluous memory migrations are triggered in Merlin's case, leading to highest P.E and lowest CPI metrics.

5.3 Importance of cost efficiency

In the next experiment, we execute a multithreaded Darwin Streaming Server from Cloudsuite [8] inside a VM with 4 VCPUs and 4GB of memory collocated with 3 copies of ScalParc and 2 copies of Fuzzy-Kmeans applications from

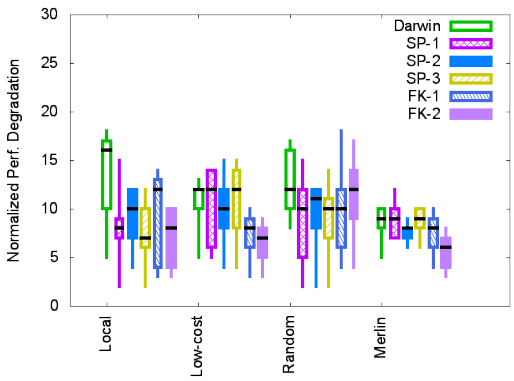

Figure 9: Reconfigurations occur due to cache and memory bandwidth re-allocation

the MineBench data mining suite. The MineBench VMs are each configured with 4 VCPUs and 4GB memory. The Darwin Streaming Server is allocated 2 physical cores initially and its CPU demand is varied using an external client workload which simulates 100 clients and requests low-bit rate (480Kbps) and high bit-rate (720Kbps) media streams of 1-min and 5-min durations over the experiment run. Serving higher bitrate streams with good fidelity increases CPU requirement at the server end. Similar to the first experiment, the streaming server's VP monitor will request additional cores when CPU capacity exceeds *200*. All workloads in this mix exhibit high MC/IC sensitivity and intensity.

Figure 9 shows performance degradation amounts for all applications across all allocation policies while 'Mix-3' in Table 1 shows corresponding efficiency metrics. Local policy first allocates local CPU to Darwin Server VP, however this leads to an increase in its MC/IC usage. In its global memory load balancing algorithm [2], local policy periodically rebalances memory-intensive workloads across NUMA nodes, by migrating their VCPUs and memory pages. This policy hurts Darwin Server showing degradation up to 17% in its bit-rates. As seen in Table 1, local policy undertakes highest number of memory migrations leading to high CPU usage and low efficiency (as seen from the lowest P.E metric). Low-cost policy disregards CPU sensitivity of ScalParc and Fuzzy-KMeans, when attempting to alleviate MC/IC usage by indirectly capping their CPU usage, while Random policy disregards all workloads' sensitivity to MC/IC resources by choosing random NUMA nodes to migrate their VCPUs. Both policies hence lead to increased variability in performance. Finally, Merlin first allocates local CPU to Darwin Streaming server, similar to local policy. However in the 'repercussions' phase (Section 3.5), upon observing that the subsequent MC/IC usage hurts the ScalParc workloads, Merlin needs to re-do the reconfiguration for Darwin server. Leveraging the MC/IC sensitivity knowledge of all workloads, Merlin has no option but to migrate either Darwin server VP or data-mining workload to another

node. The results in Figure 9 show degradations for when Merlin chooses to switch ScalParc VP (instead of Darwin server) with Fuzzy-KMeans VP. Migrating Darwin server instead causes 13% degradation to its bit-rates, higher than that caused to data-mining workloads. We postulate that the decision to migrate Darwin or data-mining workloads can be arbitrated via Merlin's use of priorities; or via application-level performance monitors providing feedback to Merlin's management. Efficiency metrics show that Merlin has the highest P.E metric despite undertaking migrations. This is due to Merlin's methods of tracking per-application VP resource needs and sensitivities which lead to management operations only when needed; instead of triggering more frequent global reconfigurations such as in local policy. This also leads to fewer costly memory migrations.

Re-allocation overheads. Finally, overheads in moving an applications VCPU to a different socket are small, on the order of $\tilde{3}$780 CPU cycles or 1.78 microseconds on our Westmere processor. These include only the cost of migrating the VCPU to another CPU and associated synchroniza- tion, and do not include overheads of losing cache-locality. Copying one 4kb page needs 430ns on our Westmere system. When page migration is inevitable, such as in the above example, up to 11% migration overheads are observed. However, these are incurred for the right workloads using Merlin's cost-awareness. Merlin's sensitivity knowledge can help cap overheads up to 4% by avoiding migrations. This motivates support for lower cost memory migration in future hardware.

Summarizing, knowledge of costs of allocation and multi-dimensional resource sensitivities helps Merlin in eliminating the less effective configurations and choosing the more effective ones for all applications while always seeking improved platform efficiency. This is shown via low performance degradation, variability and low costs of management. Consistently higher P.E metrics demonstrate that platforms and their consolidated VPs managed via Merlin's methods are indeed 'better managed' systems.

6. Related Work

Prior work on isolation [3, 9, 12, 14, 26] has devised hardware solutions to enforce applications' and virtual platforms' [26] cache and bandwidth shares. Memory scheduling policies [15] in hardware maximize fairness and overall system throughput for collocated workloads. While Merlin already leverages existing hardware support for partitioning CPU and memory resources, and cache/MC/IC resources at a NUMA node level, further improvements can be gained from additional hardware features for fine-grained allocation of cache/MC/IC resources. Complementing such gains from potential hardware advancements, with Merlin, we can also flexibly choose and vary the software policies that adapt to changing application dynamics.

Previous work addressing consolidated workloads [2, 4, 17, 18, 29, 31, 39] uses a combination of techniques like workload characterization and online prediction of LLC miss-rates to distribute applications' memory intensity across an entire platform, thereby increasing total system throughput. Merlin's evaluations against 'local' throughput-oriented policies [2] demonstrate the importance of considering sensitivity knowledge of individual applications for more efficient platform management. Recent work also explores sensitivity-driven interference mitigation, for foreground/background applications [5, 21, 37]. None of these solutions, however, cater to individual applications, offer a single unified approach to dynamic multi-dimensional resource allocation for multicore machines, or study the implications on other resource types when adjusting resource allocation along some resource dimension. Further, they do not enhance sensitivity knowledge with cost-based optimization, to make platform management more efficient. Finally, in the presence of future hardware support for isolation, previous work [30, 34] in reserving application software shares can be further augmented with Merlin's arbitration methods to manage their multi-dimensional elasticity.

7. Conclusions and Future Work

The Merlin resource allocator for server platforms offers unique new functionality to address the increasing degrees of workload consolidation in today's datacenter systems. First, it manages entire applications, through its 'virtual platform' (VP) management abstraction. Second, management considers all resource dimensions affecting application performance on single multicore servers, including its compute, memory, and memory bandwidth resources (LLC, MC and IC resource types). Third, its automated methods for application reconfiguration use arbitration guided by online models capturing both the cost of VP reconfigurations and the sensitivities of other application VPs to the resource types affected by these reconfigurations. The outcome, guided by these two metrics, are management methods that improve platform efficiency and lead to more 'effective' reconfigurations, by choosing the 'lowest cost' allocation methods that will also cause the 'least amount of hurt' to all applications. This leads to improved management of server resources and fairness in performance degradation across all applications, particularly in oversubscribed systems.

Our future work continues to evaluate Merlin's resource allocation techniques to better understand the limits of reconfiguring a server's resource shares to provide appropriate allocations to all applications. Several topics of interest are: (i) to add notification interfaces between Merlin and higher-level cluster schedulers, e.g., to prompt application migration across servers, (ii) to gain additional insights on timing issues, such as suitable epoch times, reaction vs. decision delays, and stability guarantees (Merlin currently uses simple epoch histories to integrate across actions taken in different epochs).

References

[1] P. Barham, B. Dragovic, et al. Xen and the Art of Virtualization. In *SOSP*, 2003.

[2] S. Blagodurov et al. A case for NUMA-aware Contention Management on Multicore Systems. In *Usenix ATC*, 2011.

[3] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *SuperComputing*, 2007.

[4] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth. Traffic management: A holistic approach to memory placement on numa systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, 2013.

[5] N. Dejan et al. DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. In *Usenix ATC*, 2013.

[6] S. Di, D. Kondo, and W. Cirne. Host load prediction in a google compute cloud with a bayesian model. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, 2012.

[7] EPFL. `http://parsa.epfl.ch/cloudsuite/streaming.html`, 2012. "Darwin Media Streaming Server".

[8] M. Ferdman et al. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *Asplos*, 2012.

[9] F. Guo and Y. Solihin. A Framework for Providing Quality of Service in Chip Multi-Processors. In *(MICRO*, 2007.

[10] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.

[11] C. Isci, G. Contreras, and M. Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, 2006.

[12] R. Iyer, R. Illikkal, O. Tickoo, L. Zhao, P. Apparao, and D. Newell. Vm3: Measuring, modeling and managing vm shared resources. *Computer. Networks.*, 2009. URL `http://dx.doi.org/10.1016/j.comnet.2009.04.015`.

[13] M. Kambadur, T. Moseley, R. Hank, and M. A. Kim. Measuring interference between live datacenter applications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012.

[14] D. Kaseridis, J. Stuechelli, et al. A Bandwidth Aware Memory Subsystem Resource Management using Non-invasive Resource Profilers for Large CMP Systems. In *HPCA*, 2010.

[15] Y. Kim, D. Han, et al. ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers. In *HPCA*, 2010.

[16] A. Kleen. `http://halobates.de/numaapi3.pdf`, 2004. "A NUMA API for Linux".

[17] R. C. Knauerhase, P. Brett, et al. Using OS Observations to Improve Performance in Multicore Systems. *IEEE Micro*, 2008.

[18] Y. Koh et al. An Analysis of Performance Interference Effects in Virtual Environments. In *(ISPASS*, 2007.

[19] M. Lee and K. Schwan. Region scheduling: efficiently using the cache architectures via page-level affinity. In *Asplos*, 2012.

[20] Linkedin. http://project-voldemort.com/. "Project Voldemort: A Distributed Key-Value store".

[21] J. Mars et al. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations. In *MICRO*, 2011.

[22] A. Merchant, M. Uysal, P. Padala, X. Zhu, S. Singhal, and K. Shin. Maestro: Quality-of-service in large disk arrays. In *Proceedings of the 8th ACM international conference on Autonomic computing (ICAC)*, 2011.

[23] A. K. Mishra, J. L. Hellerstein, W. Cirne, and C. R. Das. Towards characterizing cloud backend workloads: Insights from google compute clusters. *SIGMETRICS Perform. Eval. Rev.*, 2010.

[24] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary. Minebench: A benchmark suite for data mining workloads. In *IEEE International Symposium on Workload Characterization*. IEEE, 2006.

[25] R. Nathuji et al. Q-clouds: Managing Performance Interference Effects for QoS-aware Clouds. In *EuroSys*, 2010.

[26] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual private machines: A resource abstraction. In *In University of Wisconsin-Madison, ECE TR*, 2007.

[27] C. Ranger, R. Raghuraman, et al. Evaluating mapreduce for multi-core and multiprocessor systems. In *HPCA*, 2007.

[28] D. Rao and K. Schwan. vNUMA-mgr: Managing VM Memory on NUMA Platforms. In *HiPC*, 2010.

[29] J. Rao, K. Wang, X. Zhou, and C.-Z. Xu. Optimizing virtual machine scheduling in numa multicore systems. In *High Performance Computer Architecture*. IEEE, 2013.

[30] D. G. Sullivan and M. I. Seltzer. Isolation with Flexibility: A Resource Management Framework for Central Servers. In *USENIX Annual Technical Conference*, 2000.

[31] L. Tang, J. Mars, X. Zhang, R. Hagmann, R. Hundt, and E. Tune. Optimizing google's warehouse scale computers: The numa experience. In *International Symposium on High Performance Computer Architecture*, 2013.

[32] L. Tang et al. The Impact of Memory Subsystem Resource Sharing on Datacenter Applications. In *ISCA*, 2011.

[33] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal. Dynamic provisioning of multi-tier internet applications. In *Proceedings of Second International Conference on Autonomic Computing*. IEEE, 2005.

[34] B. Verghese et al. Performance Isolation: Sharing and Isolation in Shared-Memory Multiprocessors. In *Asplos*, 1998.

[35] Vsphere. `http://bit.ly/efdO7H`, 2007. Vsphere: VMWare Cloud OS.

[36] A. Wang, S. Venkataraman, S. Alspaugh, I. Stoica, and R. Katz. Sweet storage slos with frosting. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'12, 2012.

[37] H. Yang et al. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *ISCA*, 2013.

[38] S. M. Zahedi and B. C. Lee. REF: Resource Elasticity Fairness with Sharing Incentives for Multiprocessors. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, 2014.

[39] S. Zhuravlev et al. Addressing Shared Resource Contention in Multicore Processors via Scheduling. In *Asplos*, 2010.