

**VIRTUAL PLATFORMS: ACHIEVING PERFORMANCE AND
ISOLATION PROPERTIES IN VIRTUALIZED SHARED
MULTICORE SYSTEMS**

A Thesis
Presented to
The Academic Faculty

by

Priyanka Tembey

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
December 2013

Copyright © 2013 by Priyanka Tembey

**VIRTUAL PLATFORMS: ACHIEVING PERFORMANCE AND
ISOLATION PROPERTIES IN VIRTUALIZED SHARED
MULTICORE SYSTEMS**

Approved by:

Prof. Dr. Karsten Schwan,
Committee Chair
School of Computer Science
Georgia Institute of Technology

Dr. Ada Gavrilovska
School of Computer Science
Georgia Institute of Technology

Prof. Dr. Sudhakar Yalamanchili
School of Computer Science
Georgia Institute of Technology

Dr. George Cox

Intel Labs

Prof. Dr. Thomas Gross
Department of Computer Science
ETH Zurich

Date Approved: 25 October 2013

To my parents and sister.

ACKNOWLEDGEMENTS

This work would not have been possible to complete without the support and love of many people. I would like to thank them all and express my deepest gratitude.

First of all, I would like to thank my advisors: Karsten Schwan and Ada Gavrilovska. My interactions with them have helped shape this work, and me in many ways as a systems researcher. Over the course of my Masters' and Ph.D. at Georgia Tech, there have been many challenges, as there are bound to be, which have ranged from sticky bugs in code, to identifying next steps in my thesis, or being frustrated due to paper submission rejections to just dealing with plain procrastination. I could not have hoped for anyone better but Ada and Karsten to see me through them all, and continually support and inspire me to keep going forward. Ada has been in many ways a friend as well, and with whom I will continue sharing my shoe shopping escapades amongst other things even after graduation!

I will also like to thank my proposal and defense committee members: Sudha, George and Thomas for their insightful feedback through this process. Questions raised during my proposal presentation and ongoing discussions thereafter have helped define some of the next steps in my thesis, and I would like to acknowledge their help toward the completion of this work.

Next are all my friends and colleagues at Tech and in Atlanta: Vishakha, Romain, Adit, Mukil, Danesh, Vijay, Samrit, Dulloor, Qingyang, Meeta, Raghav, Rakshita and many other folks. I would like to thank them all for all the conversations, coffee-breaks, brain-storming sessions, or just crazy shenanigans that gave me a much-needed break before tackling the next bugs in my code or the ones in my head. Vishakha has been a dear room-mate, friend and support, and I will cherish in particular all the cooking and baking we did at home, along with all the little nuances that come along with being someone's room-mate.

Last but not the least, I would like to thank my family back in Pune for supporting me through these years, and being patient with a slightly forgetful daughter and sister. I'd like to acknowledge all their love and sacrifices, and words can never express my eternal gratitude towards them.

Finally, I would also like to acknowledge all the wonderful cafes and restaurants in Atlanta and the city itself which has been a warm and snuggly home to me all these years.

This work is dedicated to all these folks.

TABLE OF CONTENTS

DEDICATION	iii
ACKNOWLEDGEMENTS	iv
LIST OF TABLES	ix
LIST OF FIGURES	x
SUMMARY	xii
I INTRODUCTION	1
1.1 Background	2
1.2 Motivation	5
1.3 Thesis Statement	8
1.4 Thesis Contributions	8
1.5 Thesis Organization	10
II VIRTUAL PLATFORMS: IMPROVING ISOLATION PROPERTIES IN SHARED MULTICORE SERVERS	11
2.1 Architecture	12
2.1.1 Modeling Resource Usage	12
2.1.2 Modeling Application Sensitivity	16
2.1.3 Managing Interference	17
2.2 Virtual Platforms – Implementation	25
2.3 Experimental Evaluation	27
2.3.1 Stream and SPEC2006	30
2.3.2 Voldemort with MapReduce	31
2.3.3 Streaming-Server with MapReduce	33
2.3.4 Discussion	33
2.4 Related Work	35
2.5 Chapter summary	36

III	MERLIN: ARBITRATING ELASTICITY RESOURCE DEMANDS IN OVER-SUBSCRIBED MULTICORE SYSTEMS	38
3.1	Background and Experimental Motivation	40
3.2	Merlin Arbitration Methodology	44
3.2.1	Assessing VP Sensitivities to all resource types.	46
3.2.2	Costs of resource reconfiguration.	46
3.2.3	Choosing the right reconfiguration with Merlin	47
3.3	Experimental Evaluation	51
3.3.1	Experiment Methodology	51
3.3.2	Multi-dimensional sensitivity knowledge helps avoiding costly operations	52
3.4	Chapter Summary	57
IV	INTUNE: TOWARD FUTURE ISLANDS-BASED SYSTEMS	58
4.1	Introduction	58
4.2	Motivation for Islands and inTune	62
4.3	inTune Architecture	66
4.3.1	Resource Islands	66
4.3.2	Management Overlays	67
4.3.3	Coordination with inTune	70
4.4	Implementation	76
4.5	Experimental Evaluation	79
4.5.1	Meeting Global Platform Properties	79
4.5.2	Application Overlays	83
4.5.3	Arbitration among Policy Overlays	89
4.5.4	Summary of Evaluation	91
4.6	Related Work	92
4.7	Chapter Summary	94
V	PREVIOUS WORK	95

VI CONCLUSIONS	97
6.1 Contributions	97
6.2 Concluding Remarks	99
VII FUTURE WORK	100
REFERENCES	102
VITA	111

LIST OF TABLES

1	Platform Efficiency (PE) for each workload mix.	32
2	Platform Efficiency (PE) and other metrics for each allocation policy. . . .	53
3	inTune Coordination messages.	75
4	PARSEC – platform efficiency.	83
5	RUBiS – Throughput Results.	84
6	MPI collectives overlay.	86

LIST OF FIGURES

1	Unpredictable variation across runs for Voldemort response times in consolidated platforms	5
2	Example platform topologies	14
3	Memory intensity (L3misses) and Memory factor (L3/L2misses) for sample applications	15
4	State Transition model capturing possible interference at shared resource points: Cache, MC, IC.	18
5	VCPUs and Memory colocation scenarios	20
6	Performance degradation due to interference at shared resource points: Cache, MC, IC and model validation	21
7	Cache interference Mitigation: Mcf	23
8	MC/IC interference Mitigation: Milc	23
9	Virtual Platforms Implementation in Xen	26
10	Performance degradation and variation improvements with VP-enabled hypervisor	28
11	Execution timeline for VP-Xen	32
12	Two applications contained within individual VMs running on a 2-socket NUMA platform	40
13	Reconfiguration of CPU resource shares for Matrix-Multiplication hurt Voldemort due to consequent increase in memory intensity	42
14	Merlin arbitration steps and timeline	48
15	Variable CPU demand of Mapreduce codes	52
16	Reconfiguration of CPU resource shares for Matrix-Multiplication hurt Voldemort due to consequent increase in memory intensity	53
17	How to maintain global properties in multi-resource-manager manycore platforms?	61
18	RUBiS Components on IXP and x86 systems and their Interactions on Receive Path.	62
19	RUBiS: Variation in minimum-maximum response latencies.	63
20	Islands reduce performance variability and at increased loads, coordination helps in island right-sizing.	65

21 inTune architecture components. 67

22 Overlay management actions using inTune API. 69

23 inTune coordinator and inTune messages. 71

24 inTune components in Xen hypervisor. 76

25 PARSEC: average completion times. BS: Blacksholes FA: Fluidanimate . 80

26 PARSEC: average VCPU_wait times. 81

27 RUBiS Min-Max Response Times. Coordination helps in peak response
latency alleviation. 84

28 Apache Web server throughput scales with increasing number of cores,
especially when its ‘borrow’ request is prioritized over other overlays. . . . 89

SUMMARY

Multicore servers in datacenter systems are routinely used to run multiple disparate application workload mixes. Analysis performed in Google’s datacenters show, for instance, components (i.e., processes) of up to 19 distinct applications to be co-deployed on a single multicore node [43]. Virtualization technology further encourages this trend, increasing platform utilization via higher levels of workload consolidation. Systems software on these shared server nodes must meet challenges that include (a) providing end-to-end performance guarantees for possibly multiple applications and delivering global platform-level properties such as platform-level power or utilization caps., (b) mediating use of shared resources efficiently while offering isolation guarantees for multiple applications running on consolidated platforms to maintain their performance properties predictably, and (c) meeting multiple dynamic competing application performance levels and platform-level properties efficiently, especially in oversubscribed systems.

The goals of this thesis addresses (a)-(c) as follows: (1) by developing system-level mechanisms for addressing challenges (a)-(c), (2) by demonstrating their ability to deliver improved application performance with less variability and improved platform efficiency, and (3) by creating principles and representative methods for realizing the isolation properties sought by applications and the efficiency sought for platforms. The concrete realization of these goals is a *Virtual Platforms (VP)* enabled hypervisor - where per application or platform-level policy objectives are expressed at the system-level via elastic resource abstractions, which may also change dynamically during system runtime. For multiple consolidated applications (and their virtual platforms), there are methods that monitor and mediate their use of shared platform resources to deliver improved isolation for predictable

performance, while *Merlin*: a resource allocator for shared multicore servers makes it easier to implement higher-level arbitration policies while meeting multiple performance and platform properties.

As single-node multicore platforms evolve further from small numbers of homogeneous cores toward multiple sets or islands of potentially heterogeneous cores residing on a single chip, such platforms will have multiple resource managers managing their respective ‘islands’ of resources. Though geared toward improved scalability and functionality, for applications spanning across multiple diverse resource islands to realize such opportunities, systems software must make it easier for them to interact with the island managers; and also help islands based systems achieve end-to-end performance properties via joint coordination amongst their island managers. In order to meet the challenges in maintaining performance objectives on future ‘scale-out’ platforms, this thesis contributes *inTune*: a framework for inter-island operation, offering APIs and mechanisms that permit applications (and their virtual platforms) to interface with resource islands and their resource managers to jointly achieve application performance guarantees and global platform-level properties.

This thesis focuses on the management of compute, physical memory and memory bandwidth resources of single node server platforms, however the methods presented in this work can be extended to other resource types including network and storage resources. InTune and Virtual-Platforms are implemented in the Xen hypervisor for x86 multi-core platforms with multiple NUMA memory nodes. Evaluation with representative parallel, web-based, and real-time applications and application mixes demonstrate the benefits of using our methods to achieve application performance and platform policy objectives.

CHAPTER I

INTRODUCTION

Multicore servers in datacenter systems are routinely used to run multiple disparate application workload mixes. Analysis performed in Google’s datacenters show, for instance, components (i.e., processes) of up to 19 distinct applications to be co-deployed on a single multicore node [43]. Virtualization technology further encourages this trend, increasing platform utilization via higher levels of workload consolidation. Systems software on these shared server nodes must meet challenges that include (a) providing end-to-end performance guarantees for possibly multiple applications and delivering global platform-level properties such as platform-level power or utilization caps., (b) mediating use of shared resources efficiently while offering improved isolation for multiple applications running on consolidated platforms for more predictable execution, and (c) meeting multiple competing application performance levels and platform-level properties efficiently, especially in oversubscribed systems. Going forward, foreseeing the challenges in managing ‘scale-out’ platforms that may comprise of multiple diverse resource ‘islands’ [36], systems software also needs to explore novel methods and interfaces to deal with islands-based platforms in order to support realization of application and platform level properties spanning across multiple island managers.

This thesis introduces system-level methods in hypervisors that enable creation of *Virtual Platforms* that represent applications and their properties at the system layer via resource-level abstractions. These abstractions are multi-dimensional; including compute, memory pages and memory bandwidth resource types, and are aware of inter-resource sharing topologies in platforms. Virtual Platform management within the hypervisor monitors and mediates the sharing of a server’s compute, memory and memory bandwidth resources via

system-level methods, while interfacing with higher-level arbitration policies that help enable more efficient application execution, and prudent usage of platform resources. This thesis further introduces *inTune*: comprising standard coordination interfaces and API for islands-based ‘scale-out’ platforms to let their island managers interact with each other and jointly achieve properties pertaining to virtual platforms and their applications.

1.1 Background

This section explains the state-of-art methods in resource management that prevail in most commercially used hypervisors [3, 99], and motivates the need for *Virtual Platforms* enabled hypervisors. Current resource allocation for data center nodes proceeds in the following manner. Applications may span multiple virtual machines (VMs), where each VM encapsulates components (i.e., processes) of the single application. Examples of such application deployments include three-tier web serving applications, where each of the tiers (Web, Application and Database) are hosted within individual VMs [18]; or key-value stores where their data is replicated across multiple VMs for better reliability and fault tolerance [55]. Hypervisors typically allocate compute and memory resource shares to these VMs [10, 99, 3] leveraging hardware support to create space and time-based partitions of these resources [100, 96]. Using proportional fairness-based [32, 102] and/or reservation-based scheduling algorithms (e.g., credit-based scheduling in Xen hypervisor [10]), compute (i.e., CPU) resources are multiplexed across VMs. Memory allocators typically operate independent of CPU allocation, and allocate memory pages randomly based on availability in the physical address space.

These management methods continue to be challenged in their ability to meet application performance needs and further, to provide performance isolation to consolidated workloads. This is because (a) application performance is determined not only by their use of CPU and memory capacities, which can be carefully allocated and partitioned [10, 99], but also by the shared platform resources whose use is not easily controlled, including caches,

memory bandwidth, and I/O resources. Furthermore, (b) CPU schedulers and memory allocators operate in isolation despite their operations (e.g., migrating a VM's virtual CPUs (VCPUs), or allocating memory pages on certain NUMA nodes) having implications on use of shared resources such as caches, and memory controllers, which may cause interference to other applications. Exacerbated by applications' dynamic resource usage profiles and by the arbitrary sharing patterns of their consolidated workloads, consequent potential interference effects are well-known [20, 13]. Less clear, however, is how and to what extent each specific application's performance is impacted by the levels of interference – its *resource sensitivities* – occurring at certain shared resource points; because such sensitivities differ across applications and for each application, across types of resources, causing differences in the degrees of performance degradation experienced. Summarizing, *unpredictable performance degradation is caused by (i) varying resource intensities, i.e., application resource usage or the ability of applications to cause interference at shared resource points; (ii) interference caused due to management operations of CPU schedulers and memory allocators; and (iii) 'sensitivity' – a measure of the varying susceptibility of an application to get 'hurt' due to such interference.*

This thesis describes a new isolation management solution that deals with both resource 'intensity' and 'sensitivity', termed a *Virtual Platform (VP)*. A VP (i) *allocates and maintains elastic resource shares for all resource types (in particular, compute, memory and memory bandwidth resource types)* relevant to an application, and then (ii) mitigates sharing effects in ways that *isolate the shares of each single application* from interference caused by others, particularly for the types of resources to which it is *sensitive*. To obtain high levels of utilization, mitigation (iii) *limits and/or reconfigures an application's resource shares only if its usage intensities unduly interfere with others*, i.e., there exists actual *hurt*. (iv) Virtual Platforms are implemented in software and with low overhead, informing future hardware about needed functionality while at the same time, making them suitable for use in practical systems and across a range of existing multicore server platforms.

The virtual platform approach is motivated by the fact that from a server architecture perspective, as consolidation becomes the norm, hardware and software mechanisms must operate in unison to manage server resources in ways that maintain isolation as a first-class management principle. Operating on commodity hardware, virtual platforms make extensive use of existing performance counters and in addition, leverage the hardware's coarse socket-level isolation, provided by separate memory controllers and caches per NUMA node. Further enrichments of the VP approach may be provided by the novel mechanisms proposed for future memory subsystems, like those that allocate to applications fixed shares of caches and memory bandwidth [20, 5, 84, 69, 42, 79]. Such hardware can be used to 'enforce' resource isolation, but software solutions like those present in VPs remain necessary for two key reasons. First, proposed hardware requiring apriori knowledge of worst-case resource needs, via application profiles [69], cannot by itself deal with the dynamic nature of applications and consolidated application workloads. Second, with VPs, we can flexibly choose and vary the higher-level software policies that react at runtime to changes in workloads, operating conditions, and application behavior [95].

Virtual platforms extend previous work on software methods dealing with interference in the memory subsystem [49, 111, 13, 48, 66]. Such methods aim to increase overall system throughput, but because they do not account for application-specific sensitivities, they have limited ability to run datacenter applications at the levels of predictability and timeliness they require. Recent work underlines the importance of providing such guarantees, an example being the Bubble-up framework [60, 108] that deals with one important configuration of consolidated datacenter workloads in which latency-sensitive applications receive performance guarantees when improving overall platform utilization. VPs, in comparison, recognize the sensitivities of all currently running applications, and limit resource use only when it induces hurt, thereby improving the effective levels of performance experienced by server applications.

In summary, the Virtual Platform (VP) approach is a software solution to allocating,

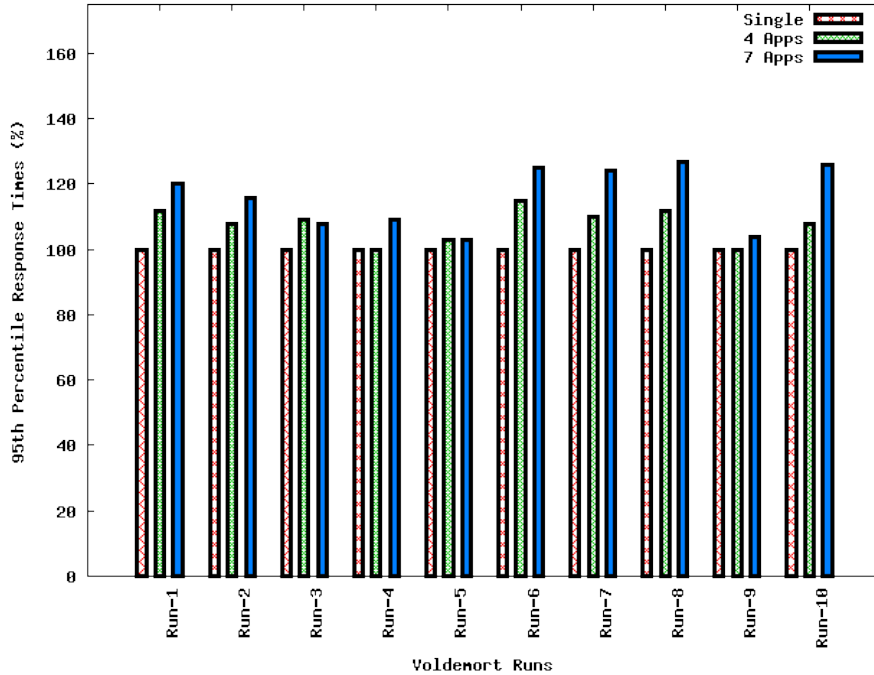


Figure 1: Unpredictable variation across runs for Voldemort response times in consolidated platforms

maintaining, and limiting application resource shares on multicore server systems. The approach specifically targets NUMA servers' *shared resources in their memory subsystems* [59], including caches, memory controllers, and interconnects. VPs are implemented as extensions to the Xen hypervisor layer to obtain generality of use.

1.2 Motivation

To highlight the problem of interference due to workload consolidation on multicore platforms, Figure 1 presents the performance effects seen when several applications are collocated on a single virtualized server node. The figure shows the 95th percentile response times for one of the consolidated applications – an in-memory key-value store, Voldemort, normalized to its performance when running alone.

Experimental platform. The experiment is run on a 32-core Intel Westmere machine with four NUMA nodes (each having memory config: 32GB DDR3 memory, 128GB in total, and CPU config: 8 cores, 256KB L2cache per core, 24MB L3 per node). The default

Xen credit scheduler and memory managers manage CPU and memory resources.

Application mix and experimental methodology. The applications used in the experiment include (1) the Voldemort server – a memory-backed key-value store [55] deployed within two VMs with 4GB memory and two VCPUs each, supporting a data-replication factor of one; and (2) and (3): two shared-memory MapReduce applications from the Phoenix application suite [80], where (2) runs Matrix-multiplication – of 600MB sized matrices deployed within one VM with 4GB memory and 4 VCPUs, and (3) runs Wordcount – parsing a 1GB text file, deployed within one VM with 4GB memory and 4 VCPUs. All guests run Ubuntu Oneiric with the Linux 3.0.2 kernel. An independent client machine generates load on the Voldemort server by sending key queries as ‘uniform’ query distributions that fetch a value size of 200KB. The experiment is run in three different configurations: (1) Single – when the Voldemort server application runs alone without any contention; (2) 4-Apps: where Voldemort runs with two Matrix Multiplication copies and one WordCount VM; and (3) 7-Apps: when Voldemort runs alongside three copies each of Matrix-Multiplication and Wordcount. For configurations (2) and (3), the experiment is run multiple times, by changing the startup order of application VMs. This ensures different initial VM memory and VCPU placement configurations across the four NUMA nodes, hence choosing different collocated application pairs per NUMA node across multiple runs; so as to not bias our experiment towards particular collocation pairs.

Observations. Figure 1 presents the performance effects observed in Voldemort’s 95th percentile response times in consecutive experiment runs for the 4-Apps and 7-Apps configurations, normalized to when it is running ‘single’. The following observations can be made. (1) Voldemort’s response times vary unpredictably across runs, the worst case degradation being 15% for the 4-Apps configuration, and 27% for the 7-Apps configuration. As every run corresponds to a different startup configuration, the variability in performance across runs suggests variability in interference caused to Voldemort, depending on how

intensively each contending co-running application uses a shared resource to which Voldemort's performance is 'sensitive'. (2) There are some runs, such as Run-4 and Run-9, where performance deterioration is negligible despite consolidation. This suggests that even in consolidated scenarios, there exist certain configurations that may cause less interference at shared resource points in the system than others, and hence, are better able to 'isolate' application resource shares.

These experiments demonstrate the following facts about the ability of virtualized state-of-the-art multicore systems to support workload consolidation. (1) *Lack of dynamic, fine-grained resource partitioning leads to highly variable levels of performance degradation.* The variation in performance degradation observed across applications suggests that applications differ both (i) in their sensitivity to consolidation and the resulting interference at certain shared resources, and (ii) in the threat they may impose on others, due to intensive use of a particular resource type. (2) *In current multicores, with their complex memory subsystems and interconnect topologies, fine-grained management of all resources is needed to curtail interference effects and provide isolation.* This is demonstrated by the Voldemort application, which intensively uses the memory subsystem (see Section 2.1), so that its performance degradation can be attributed to interference at the memory controller and off-chip interconnect, beyond the cache-level interference studied in earlier work. (3) Effects are aggravated by higher levels of consolidation, but are visible even at smaller scales.

These facts also directly motivate the key mechanisms supported in the Virtual Platforms architecture. (1) The VP abstraction captures an application's behavior at the system-level, including understanding its sensitivity to consolidation. (2) Interference models describe all potential interference points – shared resource points – on the platform being used. (3) Online measurements capture isolation threats to engender active VP management. The goal of the VP architecture is to ensure more predictable execution of applications in consolidated systems, as seen in Run-4 and 9, in Figure 1.

1.3 Thesis Statement

- With increasing complexity and diversity in hardware platforms with respect to compute, memory, memory bandwidth and interconnect resources; and disparate consolidated applications sharing these resources, systems software that intricately understands (a) the shared resource topology in the system, (b) leverages fine-grained controls to manage each of these resource types, and (c) coordinates the management of all resources in application-specific ways (including their sensitivities to each resource type) enables platforms that provide improved isolation for application properties, and better platform efficiency.
- While preserving application properties, management operations incur costs at system-runtime, not only in terms of consuming system resources but also in their implications or ‘hurt’ caused to performance of other applications. Resource allocation should consider cost- and- hurt- related metrics to improve both efficiency and effectiveness of management actions.
- Coordination amongst multiple platform-level managers will prove increasingly important in future ‘islands’-based platforms in order to enable them to realize application and platform-level global properties, with less variability. Exporting standard system-level interfaces and abstractions that hide away island complexity is both feasible and necessary to facilitate easier interaction with possibly diverse island managers.

1.4 Thesis Contributions

The specific theoretical and practical contributions of this work are the following.

1. Virtual Platforms (VPs) implement methods for ensuring that each application, i.e., its set of virtual machines (VMs), receives the portion of the platform’s resources committed to it by the hypervisor. VPs do not partition hardware resources, but instead, implement abstractions that manage elastic allocations for multiple resource types – including CPU,

memory, and memory bandwidth, in ways that make performance isolation a first-class management principle.

2. Online *interference models* are the theoretical underpinnings to obtain isolation under active VP management. These online observation-based models and metrics capture interference at all the *shared resource points* in the platform topology, namely, the processor, caches, memory controllers, and interconnects; by assessing varying application sensitivity and intensity at runtime, using per-application VM performance counters. This model-derived knowledge is then used to guide our interference management methods. Further, these models also capture the costs of dynamically reconfiguring elastic VP shares; this knowledge is used to drive more efficient and effective VP management solutions. Our models operate in a platform-independent fashion, seeking to better define future hardware/software interfaces for further improving performance isolation on multicore server platforms.

3. A *Xen hypervisor-level implementation* of VPs and interference models is validated experimentally for an Intel x86 NUMA platform. Evaluations use representative server workloads, including a commercially used key-value store, an online streaming server, and multiple MapReduce application tasks. Performance results demonstrate that with VPs, multicore resources in virtualized systems can be shared with lower levels of performance degradation while more efficiently using platform resources, which directly impacts their ability to permit higher degrees of consolidation.

4. For platforms increasingly scaling out to many, potentially heterogeneous compute cores and memory designs, this thesis presents inTune: comprising of novel coordination interfaces and abstractions at the system-level to let independent resource managers jointly host applications and their VPs, and further, achieve platform and application properties. inTune is implemented within the Xen hypervisor for software islands partitioning an x86 multicore platform. Evaluated applications include web-servers, parallel benchmarks and 3-tier application suites and we demonstrate the use of inTune-enabled coordination in

achieving better platform efficiency and less performance variability.

1.5 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 explains the Virtual Platforms architecture and online models that capture the interactions of applications at different shared resource points in the system: namely the caches, memory controllers and interconnects. Next, chapter 3 describes Merlin: our multi-dimensional resource allocator for shared multicore servers, and the methods and algorithms it uses to manage multiple applications and their elastic VP resource shares. Each chapter also summarizes relevant previous related work in the domain, and the experimental results that demonstrate utility of Virtual Platforms and Merlin resource allocation in maintaining key application-level performance with less variability. Chapter 4 describes inTune: the coordination interfaces and abstractions that will enable future ‘scale-out’ platforms to meet application and platform-level goals. Chapter 5 provides a brief summary of relevant previous work, explored in more detail in individual chapters. Finally, chapters 6 and 7 summarize respectively the thesis conclusions and future directions at the end.

CHAPTER II

VIRTUAL PLATFORMS: IMPROVING ISOLATION PROPERTIES IN SHARED MULTICORE SERVERS

Datacenter multicore servers experience ever higher levels of workload consolidation, challenging their ability to provide workloads with their required degrees of performance isolation. Addressing these challenges, this thesis presents an approach to understanding and mitigating cross-application interference arising from the uncontrolled sharing of server platform resources like caches, interconnects, and memory bandwidth. Novel about the approach is that it captures both the ‘points’ of interference experienced by running applications and the ‘sensitivities’ of these applications to the levels of interference being observed. Such modeling and observation, then, give rise to software abstractions and management methods that provide applications with improved levels of isolation and predictability, by creating and maintaining for each application a *virtual platform* (VP) as the system’s commitment of its resource shares. Our implementation of a VP-enabled hypervisor offers improved isolation properties, therefore, not by strictly partitioning hardware resources and thereby limiting potential consolidation benefits, but by instead, (i) constructing online models to capture interference effects of consolidated applications at shared resource points in the underlying hardware, (ii) understanding each application’s sensitivity to such interference, and then, (iii) using this data to manage virtual platforms in ways that make performance isolation a first-class management principle. Experimental evaluations on a high core-count multi-core machine demonstrate improved performance with reduced performance variation and increased system throughput for a wide range of enterprise applications, including a media streaming server and Voldemort, a popular data-center key-value store.

2.1 Architecture

As evident from the performance studies above, isolation causes and effects strongly depend on application characteristics and resource use. Specifically, it is only when applications are sensitive to certain resources that care must be taken to allocate to them only isolated shares of those resources. The Virtual Platform approach, therefore, begins by creating and using online interference models that capture the key interference points affecting applications on a multicore machine. This involves evaluating applications' (i) *resource intensities*, i.e., a measure of their ability to cause 'hurt' and (ii) their *sensitivities* to such interference. Guided by these interference models, software methods then manage underlying hardware resources in ways that improve isolation.

2.1.1 Modeling Resource Usage

In keeping with common practice for consolidated server systems [66], we assess application performance y (e.g., requests/sec for a web-server, job completion times) at time k based on its current resource-share allocations – $x[k]$ – as a weighted function of its CPU, memory, and memory bandwidth shares used, with generalization to other resource types like I/O expressed as:

$$x[k] = \gamma[CPU[k], Memory[k], Memory_BW[k]] \dots (1)$$

Equation-(1) represents an application's *virtual platform* of resource shares that must be allocated and maintained in order to provide performance isolation guarantees.

Most current hypervisors carefully control application compute and memory resource shares via pre-sized VM configurations [3], leveraging hardware support like EPT or VT. Memory bandwidth shares are difficult to isolate, however, particularly with the complex memory hierarchies present on today's commodity servers. Specifically, an application's current memory bandwidth use in a typical NUMA platform may be influenced by the way its data is distributed across the local cache, local memory-node banks, remote cache, or remote memory node banks. When data resides in a remote memory node, an additional issue

is the application’s sensitivity to remote access latency (RL) over an off-chip interconnect like QPI or HyperTransport. This may affect the performance of server applications like the memory latency-sensitive in-memory key-value store [55] evaluated in this thesis. We therefore, extend Equation (1) as (‘b’ being a binary input variable):

$$x[k] = \gamma[CPU[k], Memory[k], (b.Cache_usage[k], \\ b.Local_Mem_BW[k], b.(Remote_Mem_BW, RL)[k]) \dots] \dots (2)$$

Ideally, in order to define an application’s virtual platform in terms of the resource usages in Equation-(2), these quantities should be precisely measurable [42]. However, contemporary hardware does not offer controls for defining shares of caches and going further, of local memory buses, memory controllers (MC), and interconnects (IC). Consequently, we use observation-based techniques to approximate an application’s use of these resources. In our experimental Westmere platform, for instance, each physical core has an independent L1 and L2 cache, while the last-level L3 cache is shared among the cores in a single socket. Given that these caches are inclusive, an L2 miss will always result in either an L3 hit or miss. The last-level cache usage of an application can therefore be approximated as being equivalent to $L2misses - L3misses$ (where cache miss values are all evaluated per 1000 instructions). Using L3misses as an indicator of bandwidth intensity (as also done in [90, 13]), local and remote memory bandwidth intensity can then be expressed as *local L3misses/1000instructions* and *remote L3misses/1000instructions*.

Cache vs. MC/IC usage factor of an application. We assess an application’s use of the cache and MC/IC using these metrics, but note that another useful quantity is the *ratio* of its cache vs. memory (MC/IC) usage. This ratio helps us understand the relative importance of one shared resource vs. the other. This makes it possible to assess an application’s ‘sensitivity’ to cache vs. MC and IC contention (as will be shown in Section 2.1.2), a key input to effective performance isolation methods. Toward this end, we also introduce the notion of *Memory Factor*. Stated precisely, the Memory-Factor (MF) of an application is

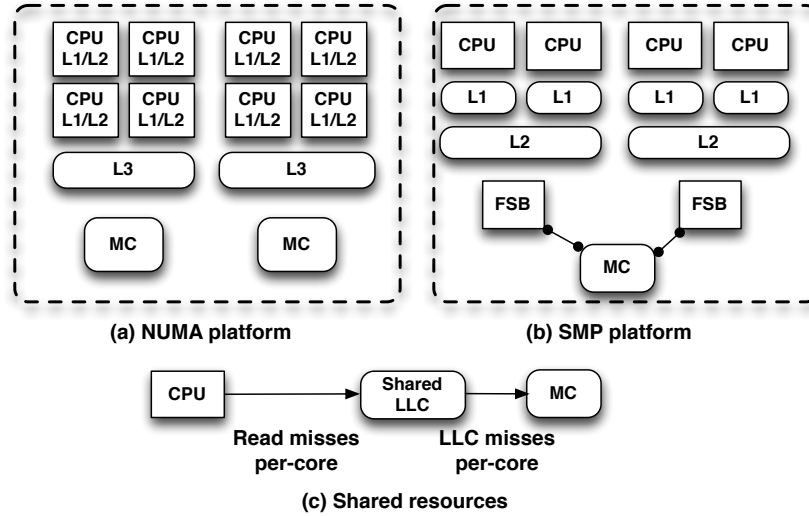


Figure 2: Example platform topologies

determined by the fraction of L2misses that end up as memory accesses, and we represent it as $L3misses/L2misses$ per 1000 instructions. 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 being served as memory accesses.

Summarizing, an application' use of shared resources can be approximated by assessing the following two metrics: (i) $L3misses/1000inst$: a measure of absolute MC/IC usage, and (ii) MF: a relative measure of cache vs. MC/IC usage. Cache usage can be implicitly calculated from the $L3miss/1000inst$ and MF metrics, and does not need explicit calculation.

Platform generality of metrics. Our metrics are generalizable to other platforms beyond NUMA, including SMPs, as follows. Figures 2.a. and b. show prototypes of two popular microarchitectures – NUMA and SMP-based – commonly used in today's data-centers. (i.e., $L3misses/1000inst$ for Westmere, and $L2misses/1000inst$ for the SMP prototype). Similarly, the MF metric, i.e., the ratio of an application's cache vs. memory controller usage, is measured as the ratio of per-core shared LLC misses and per-core independent cache misses (L2 or L1misses, respectively). Figure 2.c summarizes the hardware performance counters required by the VP management software in order to approximate an

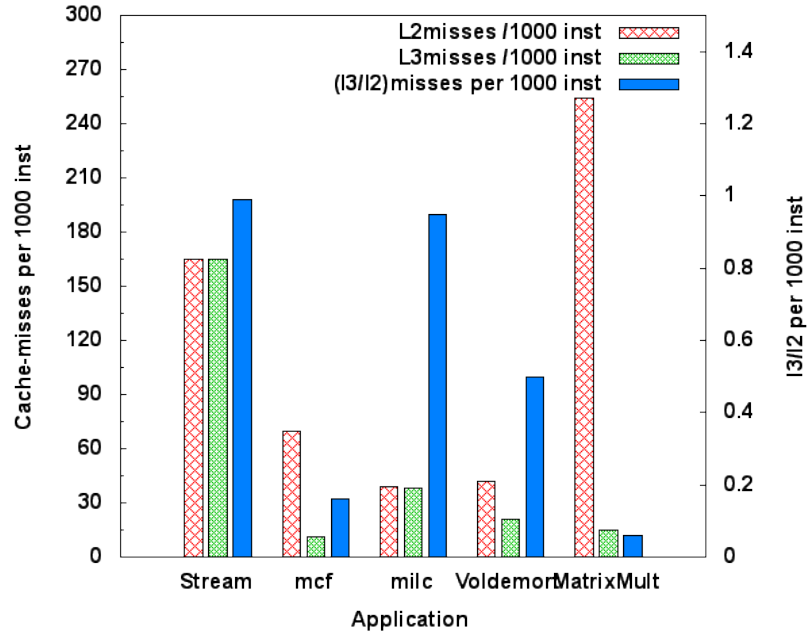


Figure 3: Memory intensity (L3misses) and Memory factor (L3/L2misses) for sample applications

application's memory bandwidth and shared cache usage.

Characterizing application resource intensity. Based on the L3 misses, we categorize applications as follows: (1) Pugs: $L3 < 2$ per 1000 instructions, are the least memory intensive; (2) Terriers: $2 < L3 < 15$ per 1000 instructions, have medium-high memory intensity; and (3) Bulldogs: $L3 > 15$ per 1000 instructions, are the most memory intensive and most intensively use the MC and IC resources. The applications are further categorized based on their MF measures as follows: (1) $MF < 0.25$: less than 25% of L2misses result in memory accesses, showing good cache reuse; (2) $0.25 < MF < 0.6$: less than 60% and greater than 25% of L2misses result in memory accesses; these application states represent a fairly high use of MC and IC resources; (3) $MF > 0.6$: with most L2misses resulting in memory accesses; these states show the highest use of MC and IC and negligible cache reuse. The bounds for each category are empirically derived using benchmarks with well-understood resource usage patterns, and the resulting 9 application states are shown in Figure 4.

As an illustration of the ability of the above classification to capture resource intensities, consider the sample server application-set used in this thesis. Figure 3 shows the L2misses, L3misses, and MF of the application-set, which includes: (1) Stream with OpenMP: an industry-standard memory-streaming benchmark, (2) Mcf and (3) Milc: two well-known memory-intensive benchmarks from the SPEC2006 suite, (4) Voldemort: an in-memory key-value store, and (5) the Phoenix Mapreduce version of Matrix Multiplication codes [80]. Stream and Milc display an MF value almost equal to one, suggesting that they have almost negligible cache reuse. Mcf and Matrix-multiplication show the best cache reuse ($MF < 0.25$), and Matrix-Multiplication is also memory-intensive (high L3miss/1000inst), while Voldemort shows reasonably good cache reuse (MF equals 0.5), with high memory intensity. Section 2.2 describes details of how this performance monitoring data is collected.

2.1.2 Modeling Application Sensitivity

In order to capture the ill effects on an application’s performance due to interference caused by varying application resource-intensities (Section 2.1.1), it is important to understand how applications differ in their *sensitivities* to interference at caches, MC, and IC resources caused by other applications. We define an application to be ‘sensitive’ to its share of a resource type, if a reduction in the resource share leads to ‘significant’ performance hurt for the application. We use both (i) the MF value of an application and its (ii) LLCmisses/1000inst to ascertain an application’s sensitivity to contention at each of the shared resource points (cache, MC, IC) as follows.

Sensitivity to cache contention. Applications like Matrix-Multiplication codes have $MF < 0.25$, and have a reasonably high memory intensity, or a high L3misses/1000inst count (see Figure 3). This implies *high cache footprint reuse* compared to the cache reuse of Pug applications with $MF < 0.25$, and a lower memory intensity. Hence, matrix-multiplication codes are *cache-contention sensitive*, because like other applications in these

states, they are more susceptible to losing their cache-shares in the presence of co-running cache-intensive apps.

Sensitivity to MC/IC contention. Applications like Milc have MF values that are almost equal to one, and they are bulldogs in terms of L3miss/1000inst counts or memory intensity. These applications use higher shares of MC and IC resources compared to Pugs with a similar MF value and hence, they are *RL, MC and IC-contention sensitive*.

In the presence of contention at shared resources, modeling an application's sensitivity as a factor of both MF and L3miss/1000inst metrics provides insight into which resource type shares (cache vs. MC/IC) need to be *isolated*. For instance, Voldemort and Matrix Multiplication have similar MC/IC intensities (as seen by the L3miss bars in Figure 3), but Matrix-Multiplication is not categorized as MC/IC sensitive (while Voldemort is), and is much less affected by contention at MC/IC. Finally, pugs use less of both caches and MC, IC and therefore, are relatively *insensitive* to contention. For SMP platforms (Figure 2.b.), applications sensitive to MC contention will also be sensitive to FSB contention.

2.1.3 Managing Interference

Sections 2.1.1 and 2.1.2 show how an application's shared resource usage intensities and sensitivities may be approximated using the MF and L3miss/1000inst metrics. We next describe how VP management is guided by intensity and sensitivity knowledge to improve isolation of an application's sensitive resource shares. This involves (i) detecting interference at shared resource points (cache/MC/IC) that may hurt the isolation of an application's sensitive shares, resulting from varying resource intensities of other corunning applications at those points, and next (ii) mitigating the interference by limiting corunners' intensities to improve the isolation of applications' sensitive shares via software methods.

Identifying Interference Points The VP approach builds for each application a two-dimensional characterization of its MF metrics and absolute L3misses/1000inst. This characterization denotes an application's 'interference state', which represents both (i) the intensity with

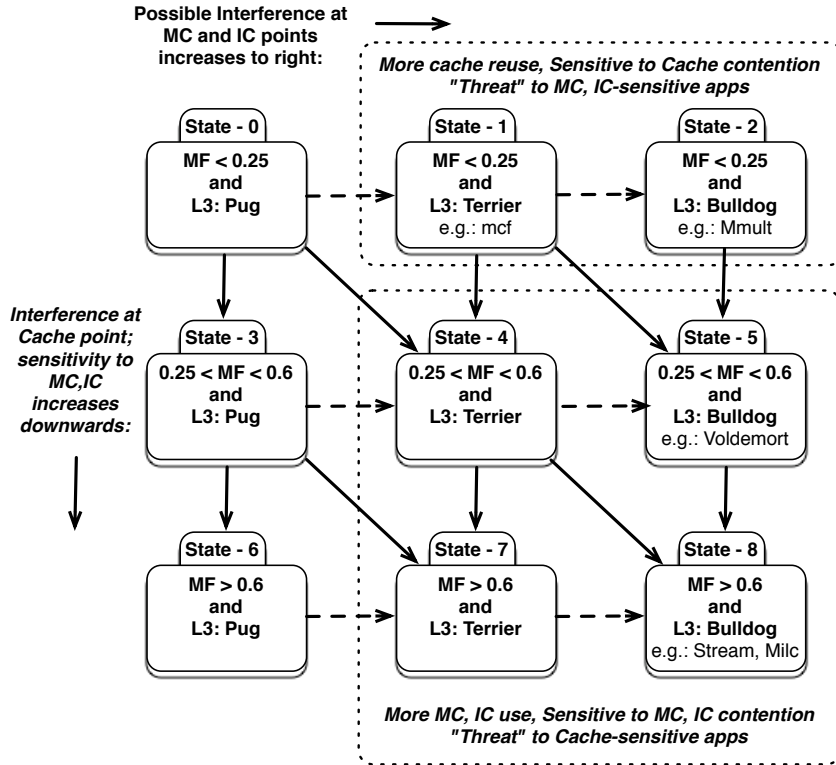


Figure 4: State Transition model capturing possible interference at shared resource points: Cache, MC, IC.

which an application uses the cache, MC, and IC resources, and thus, its ability to cause interference at these resource points, as well as (ii) the sensitivity an application exhibits to interference at these points. This two-dimensional classification yields the 9-state transition matrix shown in Figure 4. The classification thresholds defining each state are as described in Section 2.1.1. Transitions between two states in Figure 4 denote an application's change in its resource-intensiveness as well as in its sensitivity to contention at caches, MC, and IC points.

When an application transitions towards the 'right' (i.e., towards State-2,5,8), it signifies an increase in its absolute L3misses/1000inst values, i.e., increased memory (MC/IC use) intensity. When an application transitions 'along the downward dimension' of MF, it signifies a change in LLC cache access behavior without a corresponding change in L2

access behavior, indicating cache interference that can be potentially ‘hurtful’ to an application with cache sensitivity. Transitions to the left and upward signify respectively a reduction in MC/IC intensity, and mitigation of cache interference. Finally, ‘downward’ transitions can no longer be detected for State-7,8 applications, as they are completely cache-insensitive. These states correspond to being MC/IC and RL sensitive.

Experimental validation. We next validate our model’s ability to capture interference effects at cache/MC/IC points, via its state transitions. The approach creates explicit interference configurations targeting different shared resource points (cache/MC/IC) in the representative NUMA server described earlier. Specifically, different VCPU and memory placement configurations are chosen for co-runners (Q) relative to concerned application (P), as seen in Figure 5, in order to create interference at different shared resource points in the system. NUMA-aware memory management techniques are used at hypervisor-level (see Section 2.2) to confine an application VM’s address space to a particular NUMA node, and VCPU pinnings are used to pin an application VM to the CPUs of each NUMA node. Sample placement scenarios are shown in Figure 5: (1) P and Q contend for cache and MC resource shares. (2) they contend for only MC resource shares with VCPUs pinned to nodes with separate caches. (3) P runs alone but its memory is placed on a remote node, highlighting sensitivity to remote latency. (4) P and Q contend for Remote Cache, IC, and MC. To generate contention, we run three separate experiments with four copies of Stream, Mcf, or Milc, each, and two experiments with Matrix-Multiplication and Voldemort, where we reverse their role as P and Q. Figure 6a shows the performance degradation caused to the concerned application P (when running in each placement configuration shown in Figure 5) relative to when it is running alone, in the most ideal configuration: application VCPUs pinned to local NUMA node.

Figures 6b and 6c show the overall L3misses/1000instructions, and the MF measures for each of the experiment runs. We use the observations in Figures 6b and 6c to explain

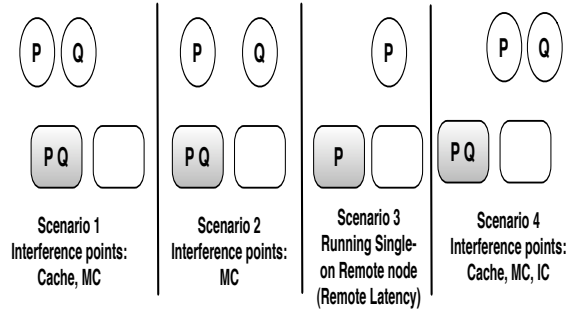


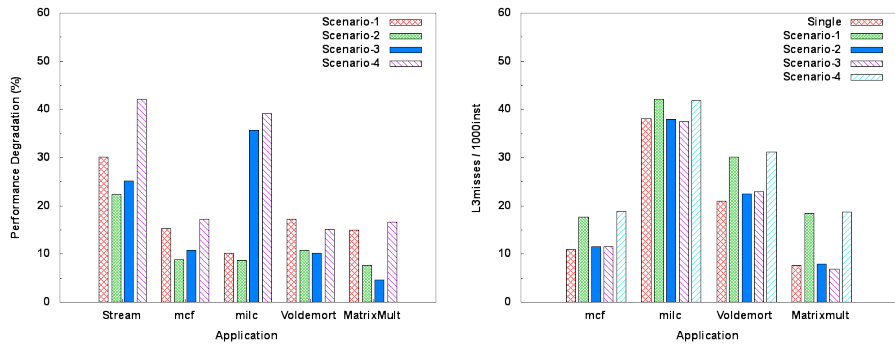
Figure 5: VCPUs and Memory collocation scenarios

performance degradation in each of the scenarios in Figure 5, hence validating the interference state model. The first bars in each of Figures 6b and 6c show the respective counter measurements when the application runs alone.

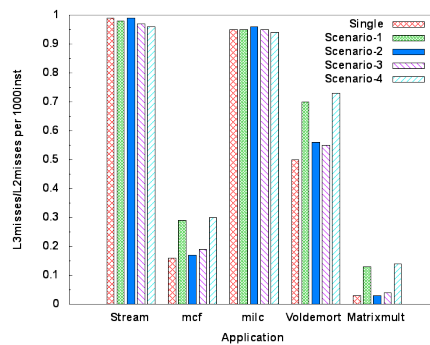
As an example of cache interference, in Scenarios-1 and 4, Mcf transitions from State-1 to State-4 due to an increase in MF values (see Figure 6c), suggesting that cache interference is degrading its performance by 15-18% in Scenarios 1 and 4. Also, consider the Voldemort application in Figure 6a: Voldemort suffers a 15-17% increase in its response time latencies, corresponding to a change in its MF values from 0.5 to 0.7 (see Figure 6c).

As an example of MC/IC interference, we consider the case of the Milc benchmark, an example application with high memory intensity and MF values (see Figure 6b and 6c). The maximum performance degradation it sees is when it is running remotely (i.e., the third bar in Figure 6a: ~35%) highlighting its sensitivity to RL. In summary, these observations validate the ability to use the state model to dynamically identify possible interference points. There are other shared resources in the system such as the TLB, and having multiple workloads in the system may affect the use of these resources. Our model can be extended to include these resources. However, our model is sufficient for VP resource management to make scheduling and mapping decisions, while we leave precise modeling of application performance as future work.

Management Algorithm



(a) Performance degradation due to interference at shared resource points (b) Changes in Memory Intensity (L3misses)



(c) Changes in use of Memory subsystem (L3misses / L2misses)

Figure 6: Performance degradation due to interference at shared resource points: Cache, MC, IC and model validation

The VP approach's current software realization uses a global platform manager combined with per-application VP monitor threads to jointly manage application resource shares guided by the state-transition model in Figure 4. Each application is assigned an independent VP monitor thread to periodically measure MF metrics and absolute L3misses/1000inst, and observe its interference states.

Upon detecting 'downward' interference state transitions, an application's VP monitor alerts the platform manager of possible cache interference and requests isolation for its application's cache share. In response, the platform manager considers the interference states of the concerned application's co-runners sharing the LLC, and may use one or combination of the following mitigation actions on NUMA platforms: (i) migrating the corunner application's VCPUs to another cache to isolate caches. This is subject to available free

CPU capacity on other nodes. The platform manager also considers ‘sensitivity’ of the interfering corunner to remote latency (RL). If sensitive to (RL), then the platform manager additionally (ii) migrates memory of the VM hosting the corunner application along with VCPU migration. This means that ‘sensitivity’ knowledge about applications is used to avoid the costly step of memory migration whenever possible. Finally, in highly consolidated cases or in SMP architectures when VCPU and/or memory migration is no longer a possible mitigation action, due to limits in CPU availability on other nodes, the platform manager may (iii) cap the CPU shares of an interfering VP within a socket, thereby indirectly affecting (i.e., limiting) its cache/MC/IC usage.

When an application is or transitions to being a State-7,8 application, its VP monitor can no longer detect further ‘downward’ transitions, as the application is completely cache-insensitive. The VP monitor then notifies the platform manager to flag the application as a ‘potential threat’ for causing MC and IC contention, as well as being sensitive to it. Finally, possible MC and IC contention is detected by the global manager by aggregating the memory load information (L3misses/1000inst) of all VPs sharing each NUMA node’s MC and IC link. For our experimental server system, we empirically define an ‘interference’ threshold of 100 L3miss counts per 1000 instructions per NUMA node as an indication of possible MC/IC contention (this value is derived from the Stream benchmark’s L3miss counts, Stream being the most memory-intensive standard benchmark). In the occurrence of such an event, the VPs with highest L3miss counts and MF values (State-8 VPs) are separated, and assigned independent NUMA nodes (using VCPU *and* memory migration), hence isolating their caches, MC and IC shares.

In summary, the state model presented above and the management algorithms based on the model capture **all** shared interference points (cache/MC/IC), triggering appropriate mitigation actions ‘guided’ by VP sensitivities and intensities at **all** shared resource points. Next, we experimentally demonstrate the effectiveness of model-based management methods for mitigating interference effects.

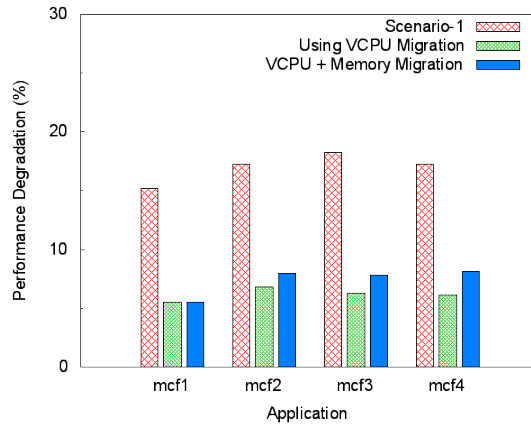


Figure 7: Cache interference Mitigation: Mcf

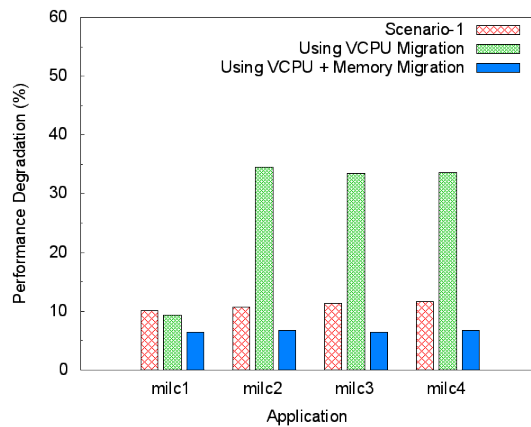


Figure 8: MC/IC interference Mitigation: Milc

Cache interference mitigation. Figure 7 shows how cache interference in Scenario-1 (see Figure 5) is mitigated for the Mcf application on a 4-socket NUMA platform. Upon getting invoked by Mcf-1’s VP monitor following a state transition, the platform manager considers ‘interference states’ of the other corunners. Noting that they are all cache-sensitive applications, the platform manager chooses to migrate each corunning application’s VCPUs to independent caches, hence isolating all cache shares and leading to least degradation for all (second bar in Figure 7). Finally, the last bars in Figure 7 represents the performance of Mcf after migrating memory pages along with its VCPUs in order to provide each Mcf application an independent node. It is interesting to observe that the

overheads involved in moving memory pages amongst nodes make seeking this ideal configuration *more* expensive, and hence lead to slight increase in degradation compared to using VCPU migration alone. Note that in this case, VP management will not select the page migration action, since it knows that the VPs associated with Mcf are not MC/IC-sensitive. From these experiments, it is clear that the model's ability to gauge application sensitivities to platform resources helps choose appropriate mitigation actions and avoid the potentially high costs of memory migration.

Finally, overheads in moving an application's VCPU to a different socket are small, on the order of ~3780 CPU cycles or 1.78 microseconds on our Westmere processor. These include only the cost of migrating the VCPU to another CPU and associated synchronization, and do not include overheads of losing cache-locality. The performance degradation caused to Mcf is <0.5% due to VCPU migrations.

MC/IC interference mitigation. Figure 8 shows MC/IC interference mitigation for Milc when it is deployed in Scenario-1 and suffers ~12% degradation (see first bar of bar-sets in Figure 8). Isolating caches alone by migrating corunner VCPUs to other nodes in the four socket system (see second bars in Figure 8) leads to small improvements in performance for milc-1, however with Milc being sensitive to RL, the corunners suffer upto ~39% degradation. By observing Milc's RL-sensitivity – since it will be classified in state-8 – the platform manager migrates memory pages next, in the 4-socket system, this leads to each Milc application having its own independent node. As can be observed from Figure 8 (see last bars), each Milc application suffers ~6% degradation in the process of mitigating such MC interference which can be attributed to the costs involved in migrating memory pages amongst nodes (Copying one 4kb page needs 430ns on our Westmere system). However, this degradation is still lower for all copies of Milc compared to the initial configuration in Scenario-1 (first bar in Figure 8).

This also indicates that for applications like Milc, upon detecting interference, the platform manager correctly uses memory migration in conjunction with VCPU migration upon

noting its sensitivities. For SMP architectures, we resort to CPU capping to indirectly limit the interfering VP's MC usage. It is possible in the future that our memory migration methods may be further optimized to leverage knowledge of hot pages within the VM, and/or compiler support within the VM providing data layouts that can be migrated first. Our current techniques work at the hypervisor level without need for guest operating system modifications.

IPC for Interference Management. Previous work [73, 108, 43] tracks the IPC metric to measure the sensitivity of an application to contention at shared resources, where a reduction in the IPC of an application when running with other collocated applications relative to its standalone (e.g., established via a profiling phase) behavior is indicative of potential hurt to performance. Though this has been demonstrated as being helpful, IPC alone is not sufficient to categorize cache vs. MC/IC sensitivity. Further, IPC-based models alone cannot assess the resource intensities of collocated applications: they need to be supplemented with information about cache misses. We hence regard IPC as a complementary interference signaling mechanism, and use our cache-miss based metrics to assess VP sensitivities and intensities so as to better guide our interference mitigation methods.

2.2 Virtual Platforms – Implementation

Figure 9 depicts the different implementation components of the virtual platforms architecture realized in the Xen hypervisor for NUMA multicore platforms. The Global Platform Manager along with individual application Virtual-Platform Monitors (VPM) form a hierarchy of management daemons that are jointly responsible for creating and maintaining the properties of the virtual platforms assigned to individual applications. VPMs and the Global Manager run as user-level threads within Xen's privileged domain, Dom0.

Virtual Platform Creation. In keeping with standard practice in consolidated virtualized systems, applications are deployed within one or more virtual machines – e.g., Figure 9

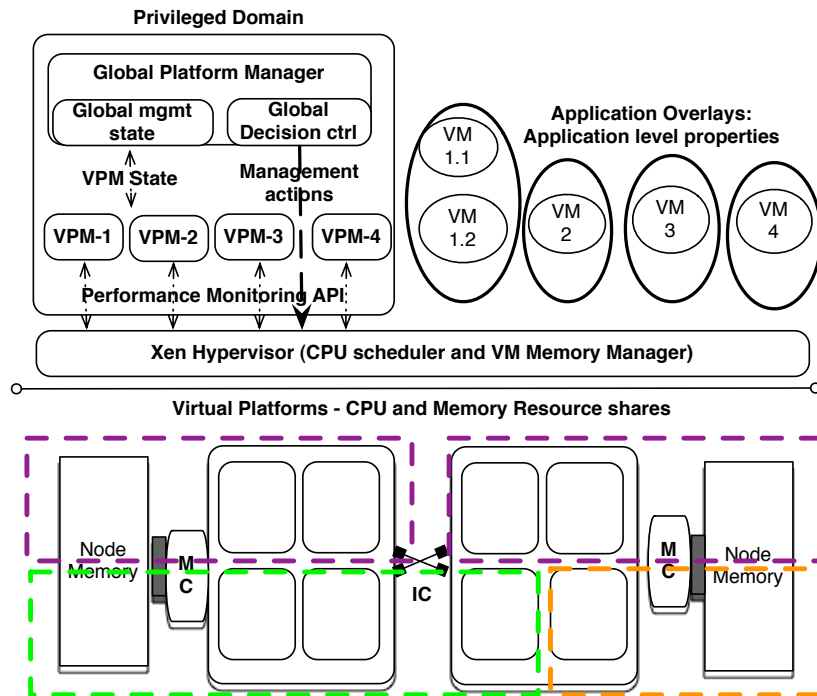


Figure 9: Virtual Platforms Implementation in Xen

illustrates 1 multi-VM and 3 single-VM applications. Allocation requirements are specified in a configuration file that states initial *CPU and memory requirements* as numbers of VCPUs and memory size. The Platform Manager allocates requested CPU and memory resource shares to the application VM(s), always attempting to allocate local CPU cores and memory, by sequentially traversing sockets. Local node data access avoids potential remote interconnect latency issues arising in NUMA platforms. CPU sharing is controlled by pinning VCPUs to independent physical cores within a socket, and a modified NUMA-aware version of the Xen memory manager [81] *confines* VM memory pages to particular NUMA nodes. We also add functionality enabling *percentage-splitting* of the VM memory across a configurable number of multiple NUMA nodes. After creating the application VMs and finalizing their resource allocations, the platform manager spawns a VPM thread to periodically monitor the VP resource shares using hardware performance counters. It also maintains topology-aware state regarding the VP's use of shared resources in the NUMA platform (caches, IC and MC).

Virtual Platform Monitor: monitoring application resource shares. The Virtual Platform Monitor (VPM) is a user-level management daemon responsible for monitoring periodic state about application resource usage, using black-box monitoring techniques. The VPM thread periodically (every 1 second) collects 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. The per-VCPU monitoring history up to one second is stored in Xen data-structures which is then exported to the user-level (Dom0) VPM via newly introduced Xen hypercalls. Further, the VPM calculates L3miss/1000inst and MF values and uses the state model described in Section 2.1.3 to detect state transitions indicating potential interference.

Platform Manager: interference mitigation and arbitration. After creating VPs with their initial resource shares and monitors, the Platform Manager (PM) continues to interact with VPMs. Interactions occur when a VPM detects possible interference or high memory intensity (i.e., the application is in State-9), and involve the PM and VPMs of all virtual platforms whose resource shares may be affected by possible mitigation actions. The current PM implementation supports the three mitigations actions described in Section 2.1.3, realized as follows. VCPU migration relies on standard Xen-supported techniques for VCPU pinning. Migration of VM pages to a remote NUMA node is implemented via new functionality performing NUMA-aware live memory migration of VM pages from one NUMA node to another. Finally, adjustment of CPU shares is performed by CPU capping supported by the Xen credit scheduler. The default capping performed per management action is 10%.

2.3 Experimental Evaluation

We next present experimental evidence of the viability and utility of automated interference diagnosis and mitigation via a VP-enabled hypervisor. VP-based interference detection

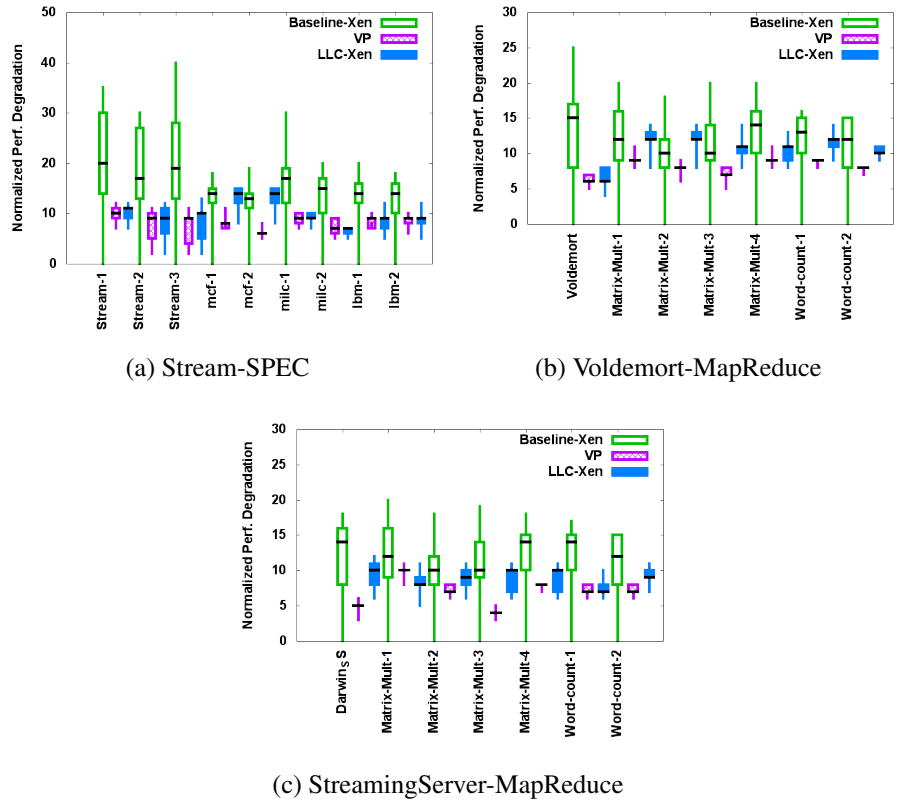


Figure 10: Performance degradation and variation improvements with VP-enabled hypervisor

methods and subsequent mitigation can help improve the isolation properties of virtualized multicore platforms.

Experiment Methodology. Evaluations use the same experimental platform as explained in Section 1.2. Workloads consist of well-known microbenchmarks as well as those that represent the emerging enterprise/cloud workloads prevalent in today’s datacenters. We evaluate the following application mixes: (1) a microbenchmark mix, consisting of the memory-intensive STREAM benchmark and known memory-intensive applications from the SPEC2006 suite: Lbm, Mcf and Milc. (2) a cloud mix, consisting of Voldemort, an in-memory key-value store, combined with throughput-oriented workloads from the Phoenix MapReduce application suite: Matrix-multiplication and Wordcount, and (3) an enterprise mix, consisting of the Darwin Streaming Server – a media-streaming server application, combined with MapReduce applications. The Darwin Streaming Server is part of

the Cloudsuite Benchmark [26] and is representative of server applications hosting media streams. All applications are encapsulated within guest virtual machines, using VM configs explained later in the section. For the purposes of this evaluation, we disable prefetching. Further discussion on prefetcher bandwidth is deferred to Section 2.3.4.

Experiment runs are performed for each of the three application mixes for three cases: (i) VP-enabled case: with our VP-enabled hypervisor managing the consolidation mix and the platform's resources, with results compared against (ii) the 'baseline' case using default Xen to manage the platform's resources, and (iii) the 'LLC-Xen' case: where we use only the L3miss/1000inst metric for interference mitigation similar to previous work [13], not accounting for application sensitivity. In case of (iii), the MF metric is not calculated, and CPU caps are not used as a mitigation technique. Instead, we use the same algorithm as in previous work [13] to invoke mitigation actions that *distribute memory intensities* amongst NUMA nodes using only VCPU and memory migration. Each experiment run starts the VMs and their applications in a consecutively different order, so as to remove bias toward certain corunner applications. We then accumulate results for each such experiment-run, remove any 'outliers' due to arbitrary application stoppage, and present the performance results as follows for each application. We observe the 'best' performance of the application in a run from the 'baseline' case, then normalize the rest of the results in all three cases as 'performance degradation' compared to this best performance value. We then present the performance degradation values as median, 50% of values about the median (*h-spread*), and the worst-case degradation values in Figure 10. For each experiment run, we isolate Dom-0, the privileged Xen domain to NUMA node-0's 8 CPU cores and deploy our application-VMs in the other three nodes, hence using 24 cores total. The results seen for each of the application-mixes are explained next.

2.3.1 Stream and SPEC2006

The following experiments assess and demonstrate how virtual platforms function to improve application isolation. We use an application mix consisting of three 4 VCPU, 4GB VMs running the Stream benchmark (operating on 2G data with 4 threads), and two copies each of: Mcf, Lbm, and Milc from SPEC2006 suite, each encapsulated in a separate Xen HVM deployed with 2 VCPUs. All HVMs run Linux(v2.6.38).

Figure 10a presents results showing the baseline-Xen performance compared against the VP-Xen and LLC-Xen cases, where the performance of each application is measured as its completion time. The following observations can be made. (i) Worst-case performance degradation in VP-enabled Xen is limited to a maximum of 10-12% across all applications, which is consistently better than the worst-case behavior seen for the baseline-Xen (up to 3x for Stream-3). This suggests that VP-enabled Xen is well-equipped to handle pathological initial configurations that may occur in a naive Xen environment, and by then using its interference detection and sensitivity knowledge about applications, it can dynamically isolate VP shares to create ‘healthier’ configurations. Stream, being a State-8 application, is highly sensitive to MC, IC contention, and to remote latency, and so, it is moved to a separate NUMA node by VP-enabled Xen irrespective of its initial configuration, which may not happen in baseline-Xen. (ii) Performance variation across runs also reduces to within 3-4% in VP-Xen, compared to several tens % for baseline-Xen. This suggests that when the VP-enabled Xen hypervisor keeps track of application ‘interference state’ transitions (Section 2.1.3), it successfully isolates the application’s sensitive shares, using interference mitigation methods that limit/reconfigure the VP resource shares of corunners. (iii) Performance degradation and variability is consistently lower in VP-Xen compared to LLC-Xen for which the worst case degradation is limited to 15% (see Mcf in Figure 10a), and variability is within 6-8%. LLC-Xen applies global reconfigurations to *all* applications while attempting to pair the workload with highest and lowest cache misses on a NUMA node. VP-Xen applies global actions to only State-8 applications, thus results for Stream and Milc

are comparable to LLC-Xen. However, LLC-Xen lacks VP-Xen’s sensitivity state knowledge. While LLC-Xen pairs ‘Mcf’ with Stream and Milc, VP-Xen instead pairs ‘Lbm’. This is because despite both being terrier applications, ‘Mcf’ is more cache-sensitive as a State-1 application, and therefore, should not share cache with State-8 applications. VP-Xen also triggers fewer memory migrations (five) compared to LLC-Xen’s eleven, which results in lower variability for VP-Xen.

2.3.2 Voldemort with MapReduce

This application mix emulates online web data processing, using the Voldemort key value store (split across 2 VMs with 2 VCPUs each), corunning with batch processing using multiple 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. Figure 10b shows the performance degradation results of VP-enabled Xen compared to the ‘baseline’ Xen case and LLC-Xen, where the performance of MapReduce applications is measured in terms of their completion times, and Voldemort performance is measured as the 95th percentile response times observed by its client. As seen in Figure 10b, the worst-case performance degradation improves by almost 3 times in case of Voldemort, combined with consistently less variation (2-3%) across runs for all applications. In this mix, both Voldemort and Matrix-Multiplication codes are ‘Bulldogs’, while Wordcount is a ‘Terrier’ application. However, Voldemort has MF sensitivity in the range of 0.55, higher than both Matrix-Multiplication and Wordcount (both $MF < 0.25$), and hence, is more sensitive to MC and IC contention. Therefore, when it is initially collocated with multiple Matrix-Multiplication codes, Voldemort response times may suffer by up to 25% in the baseline Xen case. VP-Xen also performs consistently better than LLC-Xen: a result of combining sensitivity knowledge (e.g., Wordcount is not paired with Voldemort due to its cache-sensitivity), and avoiding superfluous global migrations (VP:6 vs. LLC-Xen:13 memory migrations).

The outcomes shown above are important because it is often imperative to protect online

Table 1: Platform Efficiency (PE) for each workload mix.

App-Mix	CPU-Util (Xen)	CPU-Util (VP-Xen)	CPU-Util (LLC-Xen)	PE (Xen)	PE (VP-Xen)	PE (LLC-Xen)
Stream-SPEC	2376	2336	2370	7.838	8.556	8.272
Voldemort-1920 MapRed	1900	1900	1916	7.793	8.205	7.931
Darwin-MapRed	1970	1950	1964	7.603	8.064	7.848

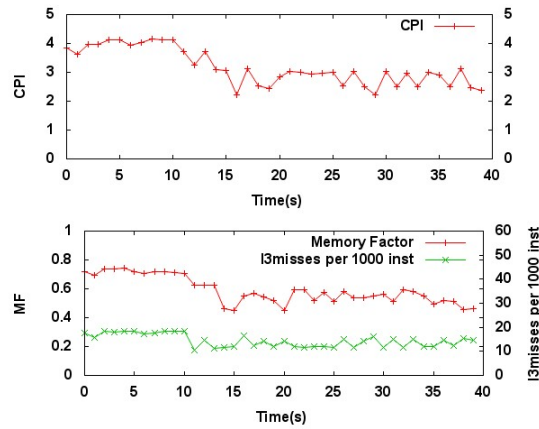


Figure 11: Execution timeline for VP-Xen

processing done with key-value stores like Voldemort from the offline or batch processing actions also being performed. Further evidence of the ability of the VP solutions to provide such needed isolation is shown in Figure 11, which depicts results for a snippet of Voldemort’s execution when running alongside two Matrix-Multiplication codes in the worst-case configuration, when the Voldemort MF sensitivity value exceeds 0.6. At $t=10\text{sec}$, the VP-enabled hypervisor migrates MatrixMult-2 away from Voldemort’s node, and further reduces the CPU cap of MatrixMult-3 by 10%, at $t=14\text{sec}$. This restores Voldemort’s MF value back to 0.55 and improves its CPI by $\tilde{50}\%$.

2.3.3 Streaming-Server with MapReduce

Web services critically rely on the timely delivery of multimedia data to end users. This means that 'interactive' services like those provided by the Darwin Streaming server (deployed in a VM of 4 VCPUs and 4GB memory) serving media streams to an external client, should be isolated from batch applications, which we emulate using the same mix of MapReduce applications as in the cloud mix above. In this experiment, the Media Streaming server client simulates 100 clients, and requests low-bit rate (100Kbps) and medium bit-rate (200Kbps) media streams of 1-min and 5-min durations over the experiment run. The performance of the streaming server is measured as the bit-rates observed at the client side for each of the served media streams. The streaming server has a similar MF sensitivity value as Voldemort (about 0.52), but it is a 'Terrier' application, which means it is not as memory-intensive. As seen in Figure 10c, its performance degradation in the worst-case of baseline-Xen configuration (about 18%) is reduced to about 6% as seen in Figure 10c, and is limited to within 12% across all applications. Darwin server being a 'Terrier' application, LLC-Xen attempts to colocate it with Matrix-Multiplication and triggers more frequent migrations during its execution, hence resulting in more variability than VP-Xen and worse degradation.

2.3.4 Discussion

Experimental evaluations show that VP-enabled platforms not only successfully reduce the performance degradation caused to applications, by detecting and mitigating interference, but also help increase predictability in application performance, by reducing performance variability across runs. This holds for a variety of application mixes with diverse resource-intensity and sensitivity characteristics, as seen in Sections 2.3.1-2.3.3. These improvements can be attributed to the interference state transition model described in Section 2.1.3, which dynamically tracks an application's resource intensity (see Section 2.1.1),

and its sensitivity (see Section 2.1.2). Sensitivity knowledge ‘guides’ VP-Xen’s management actions, in ways suited to per-VP characteristics compared to the global actions of LLC-Xen. This leads to choosing ‘healthier’ configurations that isolate the sensitive resource shares of applications coupled with fewer costly memory migrations, as seen for all application-mixes. For e.g, Stream, the Voldemort application and the Streaming Server are isolated per node, using both VCPU and memory migration methods, given their high MC/IC contention sensitivities. For cache-sensitive applications, it suffices to migrate only VCPUs avoiding memory migrations.

Table 1 shows the average CPU utilization measured for each application consolidation mix when running on default Xen, LLC-Xen and with VP-enabled Xen. It also introduces a new metric, Platform Efficiency (PE), which we define as the $(\text{normalized performance improvement for all applications}) / \text{normalized total CPU utilization}$. In our evaluation, performance improvement is the inverse of the performance degradation values shown in Figure 10 (e.g., for Stream-1 in the Xen case it is evaluated as 1/1.2, and in the VP-case, it is evaluated as 1/1.1, etc.) The CPU utilization shown in Table 10 is normalized to 2400, for 24 cores of three NUMA nodes used for running application VMs. A higher platform efficiency value indicates the platform’s ability to cause less degradation to the consolidated workloads, while utilizing less of the platform resources, thereby indicating an *improved ability for consolidation*. Note that the VP case also caps CPU, as one of its mitigation methods, hence showing lower CPU utilization, but leading to less overall performance degradation across applications. Table 1 shows that the VP architecture leads to consistently higher platform efficiency compared to using default Xen and LLC-Xen. Our VP-based approach hence shows improved efficiency in managing the resources of consolidated server platforms.

Figure 10 also shows that the best performance achieved in any of the VP-enabled runs is slightly lower than the best performance achieved for a run in the baseline Xen

configuration. This creates opportunities for future hardware support for maintaining per-VP sensitivity and intensity metrics, and execution of interference mitigation actions such as VCPU and state migration across sockets and NUMA nodes to lead to further benefits of the VP approach. Finally, our model-based management does not consider hardware prefetchers (hardware prefetch and adjacent cache line prefetch features are disabled in our experimental evaluation). It is possible to enhance the model for this purpose, as shown in previous studies [90, 60] illustrating memory bandwidth intensity estimation models with prefetching enabled, using the Nehalem’s ‘uncore’ performance counters [39]. We have not done so because the inability to control the behavior of hardware prefetching challenges the effectiveness of the software-level mitigation methods used by the current VP implementation.

2.4 Related Work

Isolation enforcement using hardware enhancements. Previous research efforts [44, 33, 20, 70, 69, 42] have devised hardware solutions that enforce cache and bandwidth shares to applications, and their virtual platforms [70]. Additional research has optimized memory scheduling policies [47] targeting latency-sensitive vs. bandwidth-intensive threads, to maximize fairness and overall system throughput. Such hardware-centric efforts complement our work on methods that seek to isolate entire, multi-VM applications under high levels of server consolidation, targeting commodity multicore servers. Second, with VPs, we can flexibly choose and vary the higher-level software policies that react at runtime to changing application dynamics.

Software solutions to improve performance isolation. Previous work addressing consolidated workloads [25, 13, 111, 48, 49] uses a combination of well-known techniques like workload characterization, online prediction of miss-rates, and then enhancing the CPU scheduler’s load balancing algorithms, to distribute the memory intensity over an entire

platform. Other solutions at the software layer specifically focusing on interference mitigation in NUMA systems have also been proposed [82, 54, 23]. Their contention-resolution algorithms focus on maximizing system throughput rather than protecting individual applications. In comparison, our work isolates applications in the presence of shared platform resources via a clearly defined interference model that guides mitigation actions, based on novel abstractions that account for application sensitivity.

Resource Management for Isolation in Operating Systems. Previous research has explored use of reservation-based algorithms at the OS level, to guarantee isolation for application properties [97, 89]. However their algorithms consider isolating only the CPU and memory resource types, and not the sharing of resources that cannot yet be partitioned using hardware controls, such as memory bandwidth.

Sensitivity-driven interference mitigation. Factoring in application sensitivities to its resource shares, while mitigating interference, is studied in recent work like the Bubble-up framework [60, 108, 73], which seeks to improve platform utilization while delivering performance guarantees. We build on such research, but also cater to interference issues in NUMA architectures, attempting for generality across server platforms. Further, our interference models distinguish between cache vs. MC/IC sensitivity, which enables them to guide management that mitigates contention in a fine-grained manner.

2.5 Chapter summary

This thesis presents a rigorous approach to improving performance isolation for multicore server platforms. It introduces a new abstraction – Virtual Platforms (VP) – to make commitments of resource shares to entire applications and to manage shared server resources in ways that maintain isolation as a first-class management principle. Specifically targeting shared memory subsystem resources, the VP approach uses a well-defined ‘interference model’ to diagnose the interference effects seen for applications by assessing their sensitivity to each of the the platform’s shared resource points. A hypervisor-based realization

of the approach then uses software mechanisms to mitigate interference, always attempting to isolate, and if needed, limit an application's resource shares to improve its performance predictability. These management methods are implemented in a platform-agnostic manner and within modest overhead bounds. The software methods shown and evaluated present opportunities for improving the hardware/software interfaces and functionality of future server systems used in consolidated and cloud datacenters.

Implemented in the Xen hypervisor using black-box monitoring, the VP approach does not require guest operating system or application inputs. Evaluations with various application mixes demonstrate the ability of VP-enabled hypervisors to improve isolation for consolidated server loads, i.e., to strongly reduce the variability in performance seen for consolidated applications (5-10%), in addition to significant improvements in their worst-case performance degradation (40-50%). VP-based techniques implemented in the Xen hypervisor lay the foundations for designing future hypervisors which should be architected keeping performance isolation for applications as a first-class management principle. The VP abstractions can also easily leverage additional hardware support for resource isolation in caches and memory bandwidth on future platforms. Future work will use online VP-based management to address additional issues, including the use of arbitration policies that take into account application priorities.

In the next chapter, we will see how VP elasticity is enabled for multiple applications to meet their dynamic resource needs especially in oversubscribed systems, while also maintaining isolation of their VP resource shares.

CHAPTER III

MERLIN: ARBITRATING ELASTICITY RESOURCE DEMANDS IN OVERSUBSCRIBED MULTICORE SYSTEMS

Applications and their Virtual Platforms tend to have elastic resource requirements based on variable load, or changes in application phase. For instance, web-servers may need to service volume spikes [29] in client requests upon certain events of interest, or may observe fluctuating load on a daily basis based on diurnal request load changes [107]. However their performance needs (e.g., web server response time latencies) still need to be met. Similarly applications may exhibit changing phases during the course of their execution. A typical instance of phase change is observed in MapReduce [22] applications when the workload shifts from the Map phase to Reduce phase of operation where the two phases differ in the manner of using resources, however together contribute to the overall performance of the application. Phase transitions and load fluctuations lead to applications having dynamic resource requirements. These resource needs are not just specific to a single resource, but vary along multiple resource dimensions such as compute, memory, memory bandwidth and I/O. An application's virtual platform (VP) is hence an elastic abstraction of resources. *In order to meet such variable performance demands of an application, a resource allocation system needs to (a) capture varying application resource needs along multiple resource dimensions either using application input and/or system-level metrics and then (b) elastically allocate these resources 'on-demand' with low cost of allocation and high efficiency.*

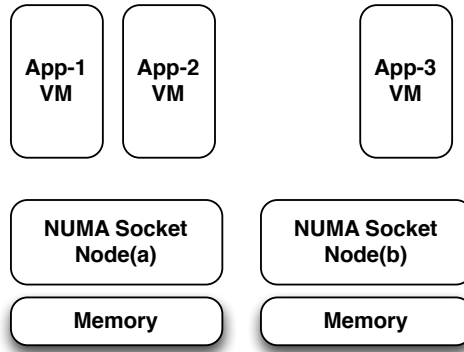
However as seen in the previous chapter 2, there are other realities in today's data centers that make multi-dimensional resource allocation a harder problem to solve. Server consolidation leads to sharing of resources, which is harder to control for certain resource

types more than others (e.g., cache and memory bandwidth shares). Furthermore, in current platform architectures and performance dependence along multiple resource dimensions, changes in resource allocation along one resource dimension may lead to indirect perturbations in 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 [94] may lead to an associated increase in its cache and memory bandwidth usage. This may in turn *cause arbitrary hurt* to applications sharing their cache and memory bandwidth resources with the web-server. It can be inferred from this example, that resource allocations for applications cannot be single resource minded, and need to be aware of (a) *inter-resource dependencies within the platform architecture, and (b) how these resources are shared across applications and their sensitivities to each of the shared resource types.*

Furthermore, resource allocations and reconfigurations incur costs in terms of resources consumed during allocation. Typically in NUMA platforms, most NUMA-aware schedulers [13, 62] attempt to *always* allocate application memory local to its CPU placement. However, migration of an application’s memory pages consumes both memory bandwidth and CPU resources. Unnecessary migrations may lead to (a) inefficient and wasteful platform management due to high overheads, as well as (b) may cause ‘hurt’ to other applications sharing the resources consumed. Cost of resource allocation hence needs to be factored in to devise more efficient allocation methods.

Summarizing, with increasing complexity of CPU, memory and interconnect resources, and consolidation trends, resource allocation needs to be aware of all inter-resource dependencies and should be both *efficient* i.e., having lower cost of resource consumption, and *effective* i.e., allocate with prudence considering sensitivities of all applications to all shared resource types.

Merlin Resource Allocation. In this chapter, we discuss the Merlin resource allocator for shared multicore platforms which has the following properties: (1) Merlin manages application VPs, i.e., its management is multi-dimensional along compute, memory and



Example NUMA topology with two sockets. Both apps (1) and (2) are allocated CPU and Memory on Node-(a), while App-3 is allocated Node-(b). All cores in one socket also share one LLC.

Figure 12: Two applications contained within individual VMs running on a 2-socket NUMA platform

memory bandwidth subsystem resources (last-level caches, memory controllers and interconnects); and is aware of all VP mappings to platform topologies and shared resources. (2) When an application’s VP needs to be reconfigured (e.g., a web-serving application needs additional CPU capacity to deal with increase in client requests), Merlin’s arbitration methods are guided by online models that capture both the cost of VP reconfigurations, and the sensitivities of other applications and their VPs to resource types affected by the reconfigurations. (3) Guided by these two metrics, Merlin’s management methods seek improved platform efficiency and more ‘effective’ reconfigurations in choosing the ‘cheapest’ method of allocation which will also causes ‘least’ amount of hurt to all applications. (4) Finally, Merlin also accommodates priorities that relatively weigh application importance in the system. Merlin is implemented in the Xen hypervisor for virtualized multicore servers, and we evaluate its arbitration methods for a range of application mixes.

3.1 Background and Experimental Motivation

This section presents the current state-of-art in resource allocation for multicore systems. In order to illustrate how resource allocation for compute and memory resources proceeds, we use the application and platform configuration shown in Figure 12. 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). In the event that App-1 needs more CPU resources, we compare Merlin’s approach against three other widely used allocation policies for NUMA multicore systems. (Note that henceforth, we refer to the NUMA node that hosts most or 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 compute and memory resources on the same NUMA node: When a remote CPU is allocated for an application VM’s VCPUs, memory pages are migrated to the remote node as well in order to keep data accesses local. Achieving local accesses is intuitively better for application performance as remote interconnect latencies are avoided. However, the cost incurred to maintain local accesses is the sum of costs to migrate VM VCPUs to remote node, and migrating memory pages; the latter being an expensive operation. Local allocation policies tend to be ‘selfish’ to suit the needs of the application requesting allocation, and may induce ‘hurt’ to other applications due to possibly frequent memory migrations, and disregard to other applications’ resource sensitivities. This policy is representative of previous NUMA-aware allocation methods [13, 62]. E.g., in Figure 12, when App-1 requests increased CPU allocation and if a remote CPU is allocated on Node-(b), then its memory will also be migrated to Node-(b). However, this may ‘hurt’ App-3 if App-3 is sensitive to the memory subsystem resource, and migration being a memory intensive operation.

(2) *Random* policy allocates compute and memory resources randomly based on availability from amongst the nodes. This policy is typically not aware of topologies, or applications’ sensitivities to resource types. This policy is representative of current Xen CPU schedulers which may balance CPU queues based on load in the system. In the example of Figure 12, random allocation may choose any CPU available on Node-(a) or Node-(b).

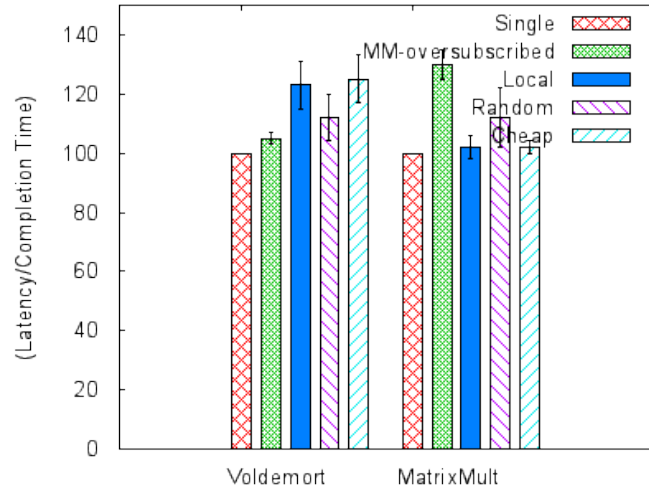


Figure 13: Reconfiguration of CPU resource shares for Matrix-Multiplication hurt Voldemort due to consequent increase in memory intensity

(3) *Cheap* policy always allocates compute and memory resources using the ‘cheapest’ method first. The software methods used by this policy to allocate and/or reconfigure an application’s resource shares are ranked in order of increasing resource consumption cost as follows: (a) Changing the CPU caps of an application’s VCPUs; (b) Allocating a local CPU on ‘home’ node; (c) Allocating a remote CPU and further migrating the application VCPUs to remote node; (d) Migrating memory pages of application to remote node. Allocating resources using the cheapest method first may be cost-efficient, however it may not be as effective due to disregard for applications’ and their corunners’ sensitivities to resource types. E.g., in Figure 12, 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.

Finally, applications may be profiled and allocated resource shares as per their worst-case resource needs. However, this approach may be wasteful, and it may not be practical to have such apriori knowledge for highly dynamic cloud workloads such as web-server applications [14].

In order to evaluate the performance implications of the above resource allocation policies, we executed Matrix-Multiplication (App-1), a MapReduce application [80] collocated with Voldemort key-value store (App-2), each application hosted in a separate VM, in a similar configuration as shown in Figure 12, with the exclusion of App-3. The CPU configuration of the two applications is as follows: Voldemort VM is a 2-VCPU VM allocated two independent cores; while Matrix-Multiplication VM has four VCPUs, and is allocated two physical cores initially (running oversubscribed). The matrix-multiplication code is configured to execute four threads, and multiplies matrices with row-size equivalent to 4096. However, the matrix rows (data) are distributed unequally amongst the four threads, and hence create a variable CPU demand. As seen in Chapter 2, matrix-multiplication codes are highly cache-sensitive, but fairly memory intensive; while Voldemort codes are sensitive to both cache and memory subsystem resources.

Figure 13 shows the performance levels of the two applications when allocated resources using the above policies normalized to when they are running alone (first bar in Figure 13). Note that when running alone, matrix-multiplication VM is allocated four physical cores, equivalent to its number of VCPUs. The second bars show performance levels when Voldemort and Matrix-Multiplication VMs are running collocated on Node-(a); and Matrix-Multiplication VM is allocated only two physical cores. This configuration tends to hurt Matrix-multiplication which shows 25% performance degradation in completion times, however Voldemort response time latencies suffer only 5% performance loss. Next, we use the ‘local’ policy to allocate more compute capacity to Matrix-multiplication when needed. The ‘local’ policy allocates local CPUs for the Matrix-Multiplication VM. Though this results in improvement in degradation (from second bar to third bar) for matrix-multiplication, it increases response time latencies for Voldemort by almost 22%; while the response time variability increases to about 10%. This shows that ‘local’ policy is selfish towards the requesting application, and disregards shared resource sensitivity of other applications which may prove hurtful to their performance. Similarly, ‘cheap’ allocation

policy (fifth bar in Figure 13) also allocates the local CPU in this case, showing similar performance degradation for Voldemort due to ignorance of other applications and their sensitivities. Finally, the random allocation policy (fourth bars in the figure) may allocate a local or remote CPU from Node-(a) or (b) with equal probability. When allocated a local CPU, Voldemort response times are hurt due to increased memory intensity at Node-(a) to which it is sensitive to; however when a remote CPU is allocated, Voldemort’s performance degradation improves, and hence on an average the degradation is about 15%. Matrix-multiplication too observes variable hurt when allocated local vs. remote CPU. However, as Matrix-multiplication codes are memory-insensitive, remote placement of CPU is not as hurtful, thereby limiting the degradation observed to 15%, which is still better than the MM-oversubscribed case (second bar). The performance variability however is high (about 12%) for both applications due to the unpredictable method of random allocation.

The observations of the previous experiment further motivate Merlin’s resource allocation properties: (1) Allocation needs to be multi-dimensional and aware of all inter-resource dependencies between compute, memory bandwidth and memory resource types; (2) Corunning applications and their sensitivities to shared resource types need to be accounted for; and (3) costs of reconfiguration need to be considered along with sensitivity information to choose the right reconfiguration for all applications. For e.g., in the above example, choosing a remote CPU for matrix-multiplication is the ideal reconfiguration as it involves VCPU migration alone (and hence is cheap), while also considering Voldemort’s sensitivity; thus leading to ‘fair’ degradation for both applications.

3.2 Merlin Arbitration Methodology

Next, we describe Merlin’s arbitration methods. We construct our arbitration methods based on previous work in storage systems [63, 103], where storage needs of multiple applications are arbitrated to meet performance guarantees in oversubscribed systems. We

adapt the storage-based arbiter in Maestro system [63] to deal with multiple resource dimensions of compute, memory and memory bandwidth resource types, also accounting for cost of management actions, and application sensitivities to multiple resource types.

We formulate our arbitration goal as follows: Merlin allocates compute, memory and memory bandwidth resource shares i.e., VP shares to all applications while minimizing the maximum difference between possible performance degradation amounts i.e., ‘unfairness’ caused to all applications. Merlin also accommodates application priorities or weights while making allocation decisions, making it convenient to specify the relative importance of an application vs. others in the system. The following linear program captures our goal:

$$w_i(1 - (u_i)) - w_j(1 - (u_j)) < \epsilon \dots (1)$$

where, u_i represents the vector of future VP resource shares to be allocated to *application_i*’s for *time – interval_k*:

$$u_i = \gamma[\beta_{ci}.CPU[k], \beta_{mi}.Memory[k], \beta_{bi}.Memory_BW[k]] \dots (2)$$

Assuming that an application’s performance is a function of its VP resource shares, the term ‘(1 - u_i)’ in Equation-(1) delineates the deviation from ideal performance. As seen in the previous chapter 2, we monitor usage of all resource types in Equation-(2) using system-level metrics. Memory Factor (MF) and L3misses/1000inst metrics approximate the cache vs. memory subsystem usage; while CPU and memory utilization denote CPU and memory resource usage of the application’s VP. Each resource type is weighted by coefficients ‘ β_{ci} , β_{mi} , β_{bi} ’ for compute, memory and bandwidth resources respectively. These coefficients represent the sensitivity of *application_i* to each resource type. ‘ w_i ’ represents the relative priority of *application_i* in the system. Next we describe how we capture the sensitivities of applications to various resource types, as well as the costs of reconfiguration using online metrics.

3.2.1 Assessing VP Sensitivities to all resource types.

Merlin uses MF and L3misses/1000inst metrics discussed in Section 2.1.2 to assess the sensitivity of an application and its VP to last level caches and MC/IC resource types. In order to assess the sensitivity to CPU usage, Merlin uses the following metric: *CPU_utilization / Allocated CPU*. For instance, a web-serving application is allocated two physical cores; but its utilization is only equivalent to one physical core, which equates its sensitivity to 0.5 toward the CPU resource. However, with a burst of client requests, when the web-server's CPU utilization increases to two physical cores, its sensitivity is now equivalent to one and hence deemed highly sensitive to the CPU resource. This further implies that if the web-server's CPU allocation is not increased beyond two physical cores, it may cause severe 'hurt' to its performance. Finally, for the memory resource type, a similar metric: *Memory_utilization / Memory pages initially allocated* may be used to assess sensitivity using inputs from the application VM or at the hypervisor-level.

3.2.2 Costs of resource reconfiguration.

Let c_i represent the current resource shares of an application's VP. In order to reconfigure an application's c_i resource shares to a new configuration: u_i , there are costs incurred in terms of resources consumed during the reconfiguration. As seen in Section 3.1, Merlin's resource reconfiguration methods comprise of (ranked in terms of increasing order of resource consumption costs): increasing or decreasing CPU shares by using CPU capping, migrating VCPUs of application VMs within and across NUMA nodes by allocating local vs. remote node CPUs, and migrating memory pages across nodes. Merlin is aware of inter-resource dependencies, and hence has knowledge of how each of these reconfiguration methods may affect usage of other resource types; thus contributing to overall costs. For instance, increasing or decreasing CPU shares of an application may affect its cache and memory subsystem usage. Migrating an application's VCPUs alone to another node leads to losing LLC locality, and using another node's LLC; while also using the off-chip

interconnect for subsequent memory accesses. Finally, migrating an application's memory pages may consume memory bandwidth and memory subsystem resources of the source and destination memory controllers, memory buses and interconnects; as well as compute capacity used for memory-copying.

A second cost of reconfiguration is the 'hurt' that it may cause to other applications due to possibly imposing on their sensitive resource shares. It is due to this second cost that choosing one reconfiguration method vs. others may incur 'variable' costs over and above the 'fixed' resource consumption costs described above, depending on the state of execution for all applications in the system, and needs to be considered while making the reconfiguration decision.

Next we see how application sensitivity related metrics and knowledge of resource reconfiguration costs guide Merlin's dynamic resource management methods when there is a need to reconfigure applications' VP shares.

3.2.3 Choosing the right reconfiguration with Merlin

Using the two-level management architecture of Virtual Platforms, per-VP threads continuously monitor an application's resource usage and sensitivity information for all resource types (CPU, cache, MC/IC and memory) in Equation-(2). When the VP monitor notices that an application's VP is 'highly' sensitive to a particular resource type, it alerts the platform manager of its resource needs. The platform manager next consults its Merlin resource arbiter to make the right reconfiguration decision. We next explain the decision algorithm that Merlin uses to reconfigure VP shares along each resource type.

CPU re-allocation. When the CPU utilization of an application nears the actual allocated CPU shares ($\text{CPU_sensitivity} > 0.8$), the application is deemed as being highly sensitive to the CPU resource type. The application's VP monitor requests the platform manager to increase the application's CPU resource shares. Given CPU availability on the platform, there may be multiple options (in the order of number of available CPUs) that

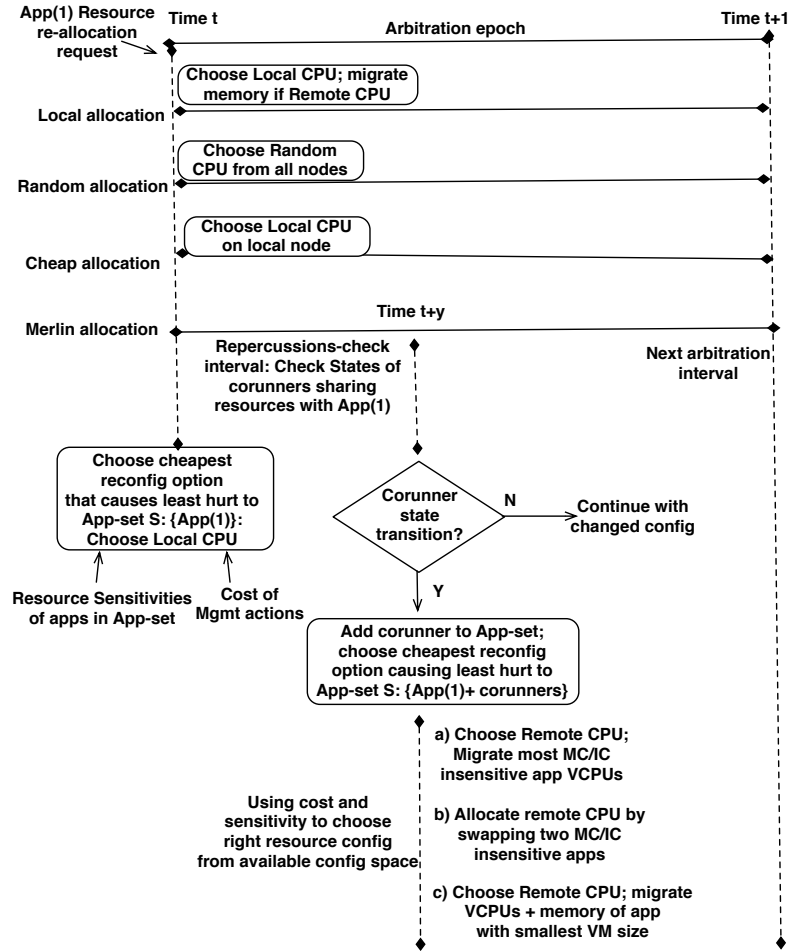


Figure 14: Merlin arbitration steps and timeline

may suit the application’s need. However the Merlin resource arbiter uses sensitivity and cost-related metrics to prune this reconfiguration space to find the most effective CPU re-allocation option by always attempting to choose *the cheapest reconfiguration option that also causes the least hurt to all applications that may be affected due to the reconfiguration.*

Figure 14 illustrates the steps and timeline of Merlin’s arbitration techniques. We use the term ‘Application-set (S)’ to denote the set of applications whose resource demands need to be arbitrated at any given time. At time instant (t), Merlin considers a requesting VP’s (App-1’s) resource re-allocation request. The application set (S) hence comprises of App-1 at time (t). Merlin first assesses availability of free compute capacity on the local node of the application. If there are available CPUs on the local node, Merlin reconfigures

the requesting VP's shares to include the available 'local' CPU core. By allocating a local CPU, Merlin also attempts to choose the 'cheapest' reconfiguration option thereby minimizing the 'hurt' caused to the set of applications in set (S). However, as seen in Section 3.1, choosing local CPU may be hurtful to corunning applications sharing the node LLC and memory controllers; particularly if App-1 is cache and/or memory-intensive. In order to monitor for potential 'hurt' caused to corunning applications, after every reconfiguration, 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 (Refer Figure 4) detected for any of the requesting VP's corunning applications are attributed to the reconfiguration actions performed at the beginning of the current epoch.

If there are no transitions, the reconfiguration is deemed successful, and Merlin proceeds to the next arbitration interval ($t+1$). However, in the event of state transitions which denotes 'hurt' caused to other corunners, Merlin adds those applications to the set (S) of the current epoch and reassesses its reconfiguration. The Merlin resource arbiter is aware of App-1 VP's mapping to the platform topology, as well as how its shared resources (such as caches and MC/IC) are used by other corunning applications and their VPs. Upon consulting the corunning VP monitors, Merlin derives relevant cache and MC/IC sensitivity information for the corunning applications, as well as for App-1. As seen in Figure 14, this sensitivity-related information along with knowledge of reconfiguration costs are used to find a better reconfiguration that meets the goal defined in Equation-(1) for applications in set (S).

As an example, in Figure 12, if App-2 is 'hurt' due to resource-reallocation for App-1, the set of applications (S) to be considered for reconfiguration is (App-1, App-2). Merlin attempts to use the next cheaper reconfiguration option: allocating remote CPU on a remote node if available (Node-b in the figure), which will also cause the least amount of hurt to applications in (S). Using sensitivity knowledge, Merlin chooses the MC/IC insensitive application from set (S) to migrate to remote node. If however, there is no remote CPU

available, Merlin next attempts to find an application on a remote node (e.g., App-3 in Figure 12) to which this cheaper reconfiguration option may be applied. If App-3 in this case is also MC/IC insensitive, Merlin ‘swaps’ App-3’s CPU resources with the appropriate local application in set (S) to create the next ‘ideal’ reconfiguration minimizing ‘hurt’ and ‘cost’ for all applications in (S). Finally, in the occurrence that none of the applications in set (S) are MC/IC insensitive, and hence unsuitable for VCPU migration alone, Merlin uses the next reconfiguration option: memory migration as well. 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 attempts to reduce overall ‘hurt’ and ‘cost’ incurred while migrating memory pages.

Memory Re-allocation. The need for memory-reallocation may arise due to insufficient memory for the application, or in the occurrence of an application’s VP comprising of ‘remote’ memory, which needs to be reconfigured to being ‘local’ because of the application’s high MC/IC sensitivity. In the second case, page migration is followed by memory copying as well which adds system overheads. In this thesis, we consider this second case where memory migration is needed to deal with applications’ sensitivity to remote latency. Memory re-allocation is the most expensive management option, and Merlin applies 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 VM memory needs to be migrated from one memory node to another in order to resolve contention at the memory controller resource between two or more applications, Merlin chooses the application VM with the smaller memory footprint to migrate. Second, Merlin executes VM 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 sensitive to the memory subsystem resources (MC/IC) as well. This technique may end up

hurting performance of the application being migrated, and Merlin resorts to using application priorities to decide which of the applications may be hurt. Finally, if priorities cannot resolve the conflict 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 data-center.

Next, we experimentally evaluate Merlin’s resource management methods, and their ability to choose more effective VP configurations for applications.

3.3 Experimental Evaluation

While evaluating the Merlin resource allocator, we define our evaluation goals as follows.

1) How do cost and sensitivity related metrics guide Merlin’s resource allocation methods to choose more ‘effective’ reconfigurations for applications compared to state-of-art techniques? What effects do they have on performance of applications?

2) Are platforms that are managed via Merlin’s resource allocation methods ‘better-managed’ systems i.e., managed with higher efficiency, low overheads and improved predictability?

3) Can Merlin provide feedback to higher-level schedulers (for e.g., at the cluster-level) highlighting the limits to reconfiguring resource shares at the node-level so as to notify the need to migrate workloads to other nodes?

3.3.1 Experiment Methodology

In order to evaluate Merlin’s allocation methods, we use application mixes comprising of Voldemort key value stores, mapreduce codes [80], and other web-search based workloads [74] collocated with data-mining benchmarks [64]. Our experimental platform is the same Westmere platform used in Chapter 2. We evaluate Merlin’s allocation methods by creating varying resource demands along multiple dimensions during application runtime, and comparing against ‘local’, ‘random’ and ‘cheap’ representative policies studied earlier in Section 3.1. We describe Merlin’s benefits and ‘effectiveness’ using observed

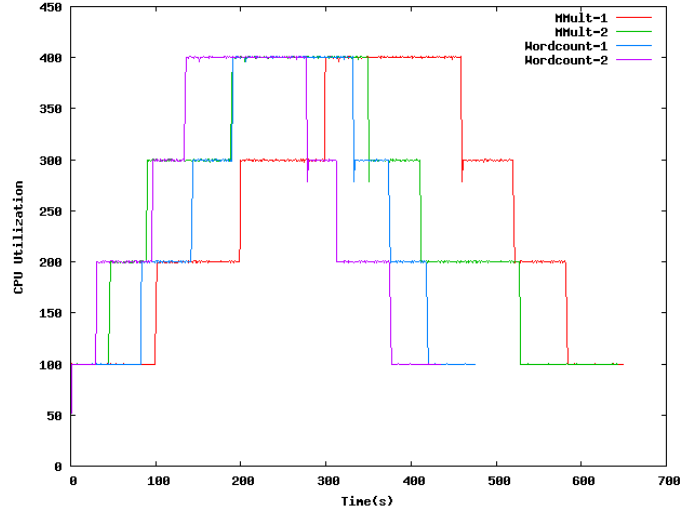


Figure 15: Variable CPU demand of Mapreduce codes

performance degradation for all applications in the workload mix as well as other end-level metrics such as observed CPU and memory utilization, number of VCPU and memory migrations undertaken and resources consumed, L3misses/1000inst and Memory Factor values of applications. Finally, we also report the *Platform Efficiency* metric for each allocation policy to demonstrate the efficiency and ‘effectiveness’ of each policy’s management operations.

3.3.2 Multi-dimensional sensitivity knowledge helps avoiding costly operations

In this experiment, we execute Voldemort key value store within two virtual machines, each with 2 GB of physical memory, and 2 VCPUs each. The client setup is similar to the experiment in Section 1.2 – where it is deployed on an external machine and queries Voldemort store for values given keys in a uniform distribution. This application mix further comprises of 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 1.2). We execute the application-mix choosing different startup order

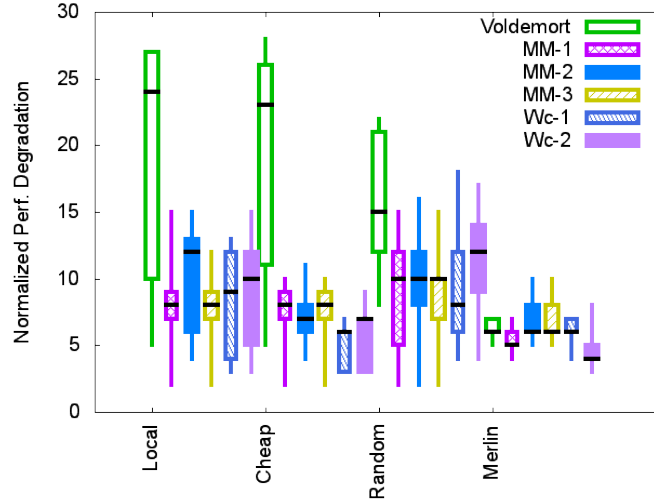


Figure 16: Reconfiguration of CPU resource shares for Matrix-Multiplication hurt Voldemort due to consequent increase in memory intensity

Table 2: Platform Efficiency (PE) and other metrics for each allocation policy.

Policy	CPU-Util	No. of VCPU Migrations	No. of Memory migrations	P.E.	L3miss / 1000inst	CPI
Local	2370	28	7	2.20	165	3.172
Cheap	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

of applications every run so as to not bias colocation on the same NUMA node.

Figure 15 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 VP monitor for the application will request an additional core. We explain next for each allocation policy, how extra cores are allocated to the applications. In this particular example, all applications have the same priority (weight in Equation-(1)).

Figure 16 shows the performance degradation experienced by the application mix depicted via the *h-spread* of the degradation amounts, and Table 2 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 2 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. The Platform Efficiency (P.E.) metric is calculated similar to Section 2.3 in Chapter 2. 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 L3miss/1000inst instructions 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) *Selfish policies 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 remote CPU (on remote node), however in addition also migrates the application's pages to the remote node. As seen in Table 2, 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 Wordcount 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 2), and increased variability (up to 13%) in performance degradation across runs for all Mapreduce codes as seen in Figure 16.

2) *Cheaper allocations may not always be most effective.* In the *cheap* allocation policy case, VP reconfigurations are performed based on their cost of allocation. As seen in Table 2, 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 *cheap* allocation policy incurs lower costs of reconfiguration (observed via lower CPU usage in Table 2 vs. local policy), and lower performance variability across applications in Figure 16), the degradation caused to Voldemort is substantial, and is caused due to disregard for application resource sensitivities.

3) *Multi-dimensional sensitivity and cost knowledge guides Merlin toward better re-configuration 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, it observes that Voldemort undergoes a state-transition to become 'highly' MC/IC sensitive due to increase in L3misses 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 16, 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 2 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 Chapter 2.), 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 2, this leads to slight increase in CPU usage for Merlin, however traded by improved performance, variability and platform efficiency.

Finally, as seen in Table 2, L3miss/1000inst 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. Summarizing, knowledge of costs of allocation and sensitivities to multiple resource types helps Merlin in eliminating the less effective configuration options and choosing the more effective ones toward improved management of platform resources.

3.4 Chapter Summary

In this chapter, we discussed the Merlin resource allocator for shared multicore platforms which has the following properties: (1) Merlin manages application VPs, i.e., its management is multi-dimensional along compute, memory and memory bandwidth subsystem resources (last-level caches, memory controllers and interconnects); and is aware of all VP mappings to platform topologies and shared resources. (2) When an application's VP needs to be reconfigured, Merlin's arbitration methods are guided by online models that capture both the cost of VP reconfigurations, and the sensitivities of other applications and their VPs to resource types affected by the reconfigurations. (3) Guided by these two metrics, Merlin's management methods seek improved platform efficiency and more 'effective' reconfigurations by choosing the 'cheapest' method of allocation which will also cause 'least' amount of hurt to all applications. This leads to improved management of server resources.

We continue to evaluate Merlin's resource allocation techniques to further understand the limits of reconfiguring a server's resource shares to provide the best possible allocation to all applications. It will be interesting to add notification interfaces between Merlin and higher-level cluster schedulers in the future to advise them of scenarios when application migration is needed to other server nodes in the data-center.

In the next chapter, this thesis explores supporting entire applications, their end-to-end performance levels and their VPs on future 'scale-out' platforms that may host multiple independent resource manager entities on the same platform (e.g., in heterogeneous computing systems).

CHAPTER IV

INTUNE: TOWARD FUTURE ISLANDS-BASED SYSTEMS

Multicore platforms are moving from small numbers of homogeneous cores to ‘scale out’ designs with multiple tiles or ‘islands’ of cores residing on a single chip, each with different resources and potentially controlled by their own resource managers. Applications and their VPs running on such machines, however, operate across multiple such *resource islands*, and this also holds for global properties like platform power caps. The *inTune* software architecture meets the consequent need to support platform-level application requirements and properties. It (i) provides the base *coordination abstractions* needed for realizing platform-global resource management and (ii) offers *management overlays* that make it easy to implement diverse per-application and platform-centric management policies. A Xen hypervisor-level implementation of *inTune* supports policies that can (i) pro-actively prepare for increased or decreased resource usage when the inter-island dependencies of applications are known, or (ii) re-actively respond to monitored overloads, threshold violations or similar. Experimental evaluations on a larger-scale multi-core platform demonstrate that its use leads to notable performance and resource utilization gains: such as a reduction in the variability across request response times for a three-tier web server by up to 40%, and completion time gains of 15% for parallel benchmarks.

4.1 Introduction

As systems integrate more heterogeneous resources and scale out to many cores, it is imperative that their diverse resource managers coordinate to achieve platform-global properties and/or to run applications efficiently. In this thesis, we present *inTune*: a set of system-level coordination abstractions and mechanisms that let resource managers interact with each other to achieve global properties and make it convenient for applications to interact

with them via standard interfaces.

To sustain 2x performance growth with every chip generation, many-core architectures are evolving beyond high core counts and complex memory hierarchies to ‘scale-out’ architectures, such as with the ‘tiles of cores’ approach [36], to directly address scalability concerns for hardware cache coherence and thermal design power (TDP) bounds [37, 109]. Similar scalability concerns with routing a single global clock across the chip within limited power bounds and at ever increasing core frequencies, may lead to emergence of multi-clock sets of cores mapped to independent clocks and voltage/frequency domains [41, 72].

Hardware designs with such on-chip networked tiles of independent, disaggregated nodes give rise to software architectures structured as clusters of multiple, independent resource managers on a single chip [28, 30]. With the emergence of core and memory heterogeneity [57, 6] supported by specialized runtimes or custom device drivers (e.g., GPGPU platforms), these independent managers may also be heterogeneous. Further, even without hardware heterogeneity, multiple functionally different resource managers (e.g., real-time vs. throughput-oriented CPU schedulers) may manage spatially multiplexed CPU core-sets on a single platform to meet the requirements of highly diverse applications, like high throughput web services vs. real-time services like VOIP [53].

Systems with multiple distinct resource manager entities have been built to support heterogeneity in hardware resources [45, 71] and to enhance scalability [11, 30, 28]. Efficiently running applications on such systems, however, requires dealing with two fundamental issues.

I. *Maintaining global properties.* Applications may operate across multiple resource domains, and their resource managers schedule application components independently unaware of overlaying end-to-end application properties. How can applications obtain and maintain their desired levels of performance given the resource managers’ independent scheduling decisions? Coordinating independent schedulers has been shown important for high-throughput graphics codes, between CPU threads and their GPU activities [34], and

for parallel systems so that program threads avoid undue levels of timing variation for synchronization [30]. How to additionally maintain platform-level properties, like maintaining system-wide power caps across proposed independent voltage/frequency domains of cores [72] using coordinated power allocation [65, 67]? Stated in terms of systems functionality, what are the system-level coordination abstractions and supports needed to let independent managers interact to obtain these goals and maintain these properties?

II. *Diverse and independent managers.* Consider applications running on a single platform with multiple resource managers, with their own scheduling policies and maintaining their own threading or resource abstractions. How do they interact with these managers, via which interfaces, and/or can managers hide such complexity by coordinating with each other on an application's behalf?

Figure 17 shows an example that maintains global properties for a multicore platform on which different sets of software-partitioned cores are managed by independent and functionally different CPU scheduler entities. In this case, scheduler diversity serves to meet different functionality requirements, i.e., the need for real-time vs. throughput-centric core scheduling. Yet as typical for larger scale web applications, a single such code spans multiple of these core sets and their managers and so, it becomes difficult to maintain its desired end-to-end properties. From this example, we derive the following objectives for our research: (i) the need for standard abstractions and interfaces that provide applications or higher level policy managers the means to efficiently and conveniently interact with a platform's diverse resource managers, and (ii) the need for interaction – *coordination* – across different managers to provide to even a single application the composite levels of service needed to run it with the performance it requires.

The *inTune* systems software presented in this thesis provides an efficient framework for managing applications and their performance properties on many-core systems organized as multiple resource sets, termed *resource islands*, with their own *island managers*. Our research makes the following three key contributions.

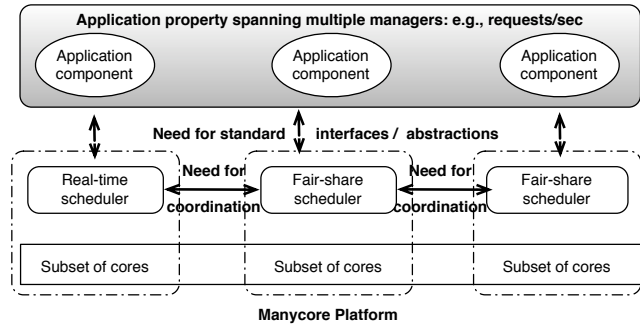


Figure 17: How to maintain global properties in multi-resource-manager manycore platforms?

1. inTune implements *resource islands*, as a system-level abstraction that places a subset of resources under the control of a specified resource manager. With islands, one can partition multicore hardware for reasons of scalability, e.g., the 'tiles' in the Intel SCC chip, for reasons of hardware heterogeneity, e.g., for commodity vs. accelerator cores, or to enable functionally different schedulers using software partitions. The example studied in this thesis are islands created as software partitions for multiple CPU sets managed by distinct scheduler functions, to suit the varied needs of server and parallel applications using a multicore platform.

inTune islands are dynamic partitions of multicore resources, so that within the limits defined by underlying hardware capabilities (e.g., cache coherence, or architectural incompatibilities), they can be flexibly sized to meet applications' elastic resource demands.

2. The *inTune framework* – offers system-level functionality that provides APIs and base mechanisms for inter-island coordination. Its mechanisms permit island managers to independently manage their resources, but then: (i) extend island-level resource managers, by pairing them with per-island *coordinators* that then interact with each other (and with their respective 'native' resource managers) using inTune APIs to achieve platform global properties, thus (ii) abstracting away from applications the heterogeneous resource management implementations of different islands.

3. *Management overlays* – are inTune's system-level representations of application and

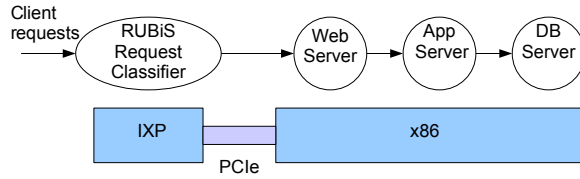


Figure 18: RUBiS Components on IXP and x86 systems and their Interactions on Receive Path.

platform properties. They (i) encapsulate and maintain state that defines the specific property of interest (e.g., performance requirement of an application or platform utilization cap), (ii) define the linkages across the inTune coordinators involved in carrying out actions that serve to obtain and maintain the property, and (iii) can use proactive or reactive methods for policy implementation by invoking inTune coordination mechanisms.

The inTune abstractions and methods in this work are implemented with the Xen hypervisor [10], for multicore platforms with independent sets of CPU islands managed by distinct scheduler functions. The hypervisor-based implementation serves the purpose of exploring potential platform enhancements without reliance on specific guest OS features or functionality. inTune’s methods and principles, however, can also be implemented in other system infrastructures, including in the operating system [56], or in island-based microkernels with island resources managed by library OSs [24].

We next present technical evidence of the importance and utility of the inTune approach for efficiently running applications on multi-island platforms (see Section 4.2). This is followed by Section 4.3 describing the inTune architecture and its components. Section 4.4 explains the inTune implementation, followed by experimental evaluations in Section 4.5. Section 4.6 summarizes prior related work with conclusions and future work appearing at the end.

4.2 Motivation for Islands and inTune

The most natural motivation for to view platforms as islands of resources concerns heterogeneous platforms – different types of cores, e.g., CPU vs. GPU or communication

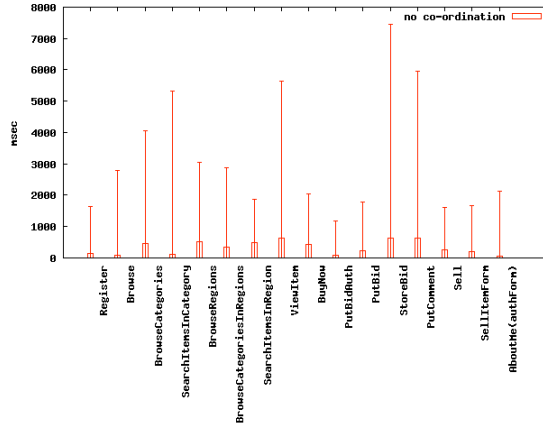


Figure 19: RUBiS: Variation in minimum-maximum response latencies.

cores, with different instruction sets, perhaps private caches and memories, etc. Here, differences in core type create natural island boundaries, and those are further highlighted by the presence of different runtimes for different cores. However to run applications efficiently on these platforms, islands need to coordinate. We next experimentally demonstrate how heterogeneous islands need to coordinate to achieve application properties. Consider a prototype heterogeneous platform comprised of a general purpose set of x86 processors connected over PCIe to an IXP Network processor [1]. The platform is used to run the RUBiS web application, which is an eBay-like auction website benchmark (see Figure 18). The x86 processors are managed by the Xen hypervisor [10], where RUBiS is run by placing its three major components, namely the Web, Application and database servers, into separate virtual machines. Requests issued by external clients are handled by the IXP platform component, which acts as a programmable network interface that sends and receives RUBiS traffic between our prototype host and clients. Previous work [9, 87, 2] has shown that the resource usage of multi-tier applications is governed by incoming client requests and their types. Exploiting this fact, a request classification engine performing deep packet inspection and running on the IXP processor can be used to better manage the CPU resource allocations given to individual RUBiS components running on the x86 processors.

Needless to say, the performance improvements sought in this fashion cannot be realized unless there are well-defined and efficient interfaces between the message-centric

resource management methods existing on the IXP (e.g., the priorities used for servicing different message queues) and the process- or VM-centric management methods used on the x86 platforms. This is demonstrated in Figure 19, which shows the minimum and maximum end-to-end response time latencies for various RUBiS request types, as observed by the client in this setup. These measurements show substantial variation in the minimum and maximum response time latencies of requests, which are due to the fact that there is no coordination between the IXP’s queue-centric and the x86’s VM-centric resource management actions. Using coordination between the IXP and host islands, we are able to alleviate peak response time latencies, by almost 30%.

Next, in order to highlight the use of islands and inter-island coordination to efficiently run an application across systems comprising multiple resource islands on a single multi-core hardware platform, Figure 20 demonstrates the performance effects seen for a representative parallel application, Fluidanimate from the PARSEC application suite when run in different configurations as follows. Fluidanimate, like other parallel codes, operates in multiple CPU-intensive phases implemented via explicit barrier synchronization. For this code, it is important to minimize its performance variability in each phase, so that all of its threads simultaneously reach the barrier.

On a 12-core hyperthreaded x86 Westmere processor platform, we execute copies of Fluidanimate codes inside each of three virtual machines with 512MB memory and configured with different number of VCPUs for the following two scenarios: (1) when the platform is underloaded and there is no CPU contention i.e., each VM is configured with 4 VCPUs (the left set of bars in the figure), and (2) when the platform is overloaded, such as in the event of a higher degree of workload consolidation i.e., each VM is configured with 8 VCPUs (the set of bars on the right). For each scenario, we compare mean application completion times along with observed deviation in (i) the default – the *NoIslands* case running native Xen credit scheduler, to (ii) a case where CPU resources are partitioned into smaller cells – the *Islands* case (with six cores each) – and (iii) a case where

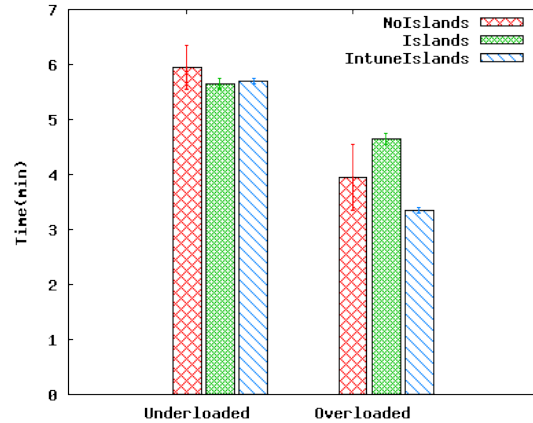


Figure 20: Islands reduce performance variability and at increased loads, coordination helps in island right-sizing.

islands are elastic because there is explicit coordination across their individual schedulers, the goal being to achieve island right-sizing during an application’s CPU-intensive phases – the *IntuneIslands* case.

The figure shows performance improvements with a higher number of application threads in the *overloaded* scenario for all three cases. However, we observe high amounts of variability in Fluidanimate completion times in the *NoIslands* case, particularly in the *overloaded* configuration. Use of partitioned CPU-sets in the *Islands* case lowers these levels of performance variability – which suggests that even at these moderate scales, *islands are important*. However, in the *overloaded* configuration in the *Islands* case, due to insufficient CPU resources or due to lack of knowledge of workloads’ worst-case requirements, application performance may actually deteriorate due to infeasible static right-sizing of islands (as denoted by higher completion times). This implies that *coordination across islands is critical* to more flexibly size islands by trading resources, in order to meet application performance needs. This is shown via both improved completion times and lower variability values in the *IntuneIslands* case.

In general, Section 4.5 shows that performance advantages gained from using inTune abstractions can be striking – leading to a reduction in the variability across request response times for a three-tier web server by up to 40%, and completion time gains of 15%

for parallel benchmarks.

4.3 *inTune Architecture*

The *inTune* software architecture is an efficient framework for implementing platform-level and application-centric management for future many-core systems comprised of multiple, distinct *resource islands*. Platform-global properties are met using the following key design components of the *inTune* architecture:

4.3.1 Resource Islands

Figure 21 illustrates the main *inTune* components for a representative four-island, general-purpose, multi-core NUMA platform. A resource island is a system level representation of a subset of the resources (e.g., CPU, memory) present on the multicore platform in Figure 21, which in this case, can be initialized to naturally match the multi-socket (4 sockets) nature of this machine. Each socket however can be configured to host multiple islands. The CPU resource within each island is managed independently by a scheduler (e.g., credit-based, best effort, or real-time), determined at the time of island creation, and is not visible to other islands.

For island creation, a boot kernel initially performs discovery of resources, namely, enlisting the CPUs, memory, and devices in the global namespace of the entire platform. A single hypervisor image booting on all cores performs this task. Using this globally available shared namespace, CPU islands are created by grouping subsets of CPUs, assigning a single CPU scheduler to each, via a global platform manager entity (e.g., the privileged domain in our hypervisor-based model – not shown in the figure for clarity). The global manager also updates the shared namespace with mappings between islands and resources they manage. Memory islands may be defined by assigning NUMA memory node mappings to an island in a NUMA multicore platform. The global platform manager also instantiates a *coordinator* thread entity per resource island, primarily to interface with other islands in the system.

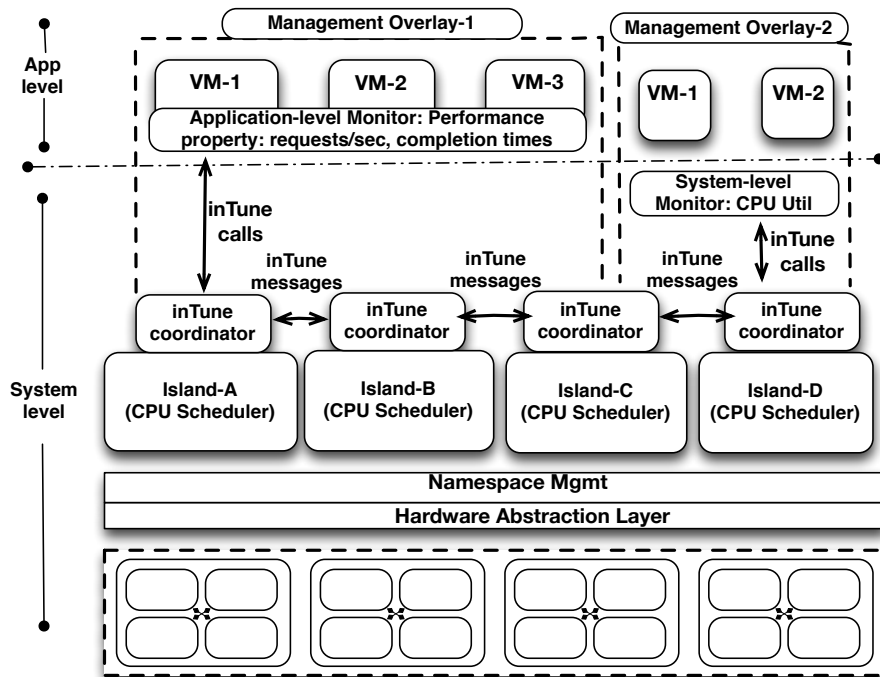


Figure 21: inTune architecture components.

4.3.2 Management Overlays

Achieving platform-level properties, like power caps, and application properties, like requests/sec for multi-tier web services, requires actions that span multiple resource islands. Island resource managers, however, are unaware of those, and so, will schedule application components in an isolated and independent manner. This encourages scalable operation, but can detrimentally affect application performance (see Section 4.2) and prevent meeting platform needs. Responding to this issue, *management overlays* (i) encapsulate and maintain the state that defines desired global platform or application properties, at system-level, and in addition, they (ii) *actuate* inter-island coordination methods to achieve their specific properties.

In our hypervisor-based implementation, application-centric overlays (see Overlays-1 and 2 in Figure 21) comprise (i) the collection of component VMs used by a single application, (ii) the performance property (e.g., requests/sec, response times) specified at overlay-creation time, (iii) performance monitor threads implemented at the system-level and/or application-level that periodically monitor resource management state pertaining to

the performance property, and finally (iv) system interfaces that the monitor threads use to request resource (CPU, memory) reconfigurations in order to maintain this property. These requests are then translated into inter-island coordination actions at the system-level.

Creating overlay components. A global platform manager creates an application-centric overlay by first creating the application VMs and mapping them to resource islands. Overlay VMs are *registered* using a unique overlay identifier with each constituent island. Further, each constituent island is made aware of the mapping between overlay VMs and other peer islands of the overlay. Affinity of application VMs to islands is determined by island properties, including available number of cores, memory size, and scheduler function. Islands are queried for these properties to decide initial VM-island mappings. An individual VM within an overlay may not span island boundaries to preserve isolation of the VM and the island resource management. An island's resources, however may be shared by multiple distinct overlays (see Island-C in Figure 21 hosting VMs of Overlay-1 and -2). The overlay monitors are then started at the system level (Overlay-2 in Figure 21) or application-level (Overlay-1 in Figure 21) to periodically monitor performance state, and to request reconfiguration of resources for component VMs. Finally, the platform manager also creates the necessary inter-island communication channels among islands within an overlay (e.g., amongst Islands-A, B and C in Overlay-1 and between Islands C and D in Overlay-2 in Figure 21).

The global platform manager is hence primarily responsible for creating the various components of overlays and passing relevant overlay state to constituent islands. inTune's current design hence delegates the runtime overlay management overhead of (i) interfacing with the overlay monitor for its resource re-adjustment requests, and (ii) translating those requests into subsequent actuations of inter-island interactions, to island coordinators instead of a centralized global manager. We make this design choice favoring scalability of overlay management for future manycore platforms, however inTune's architecture does not prohibit more centralized implementations.

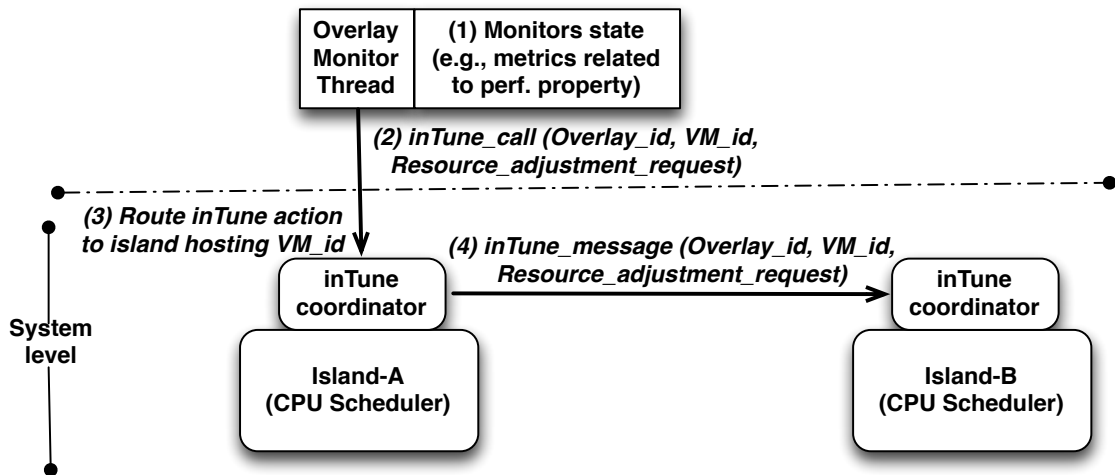


Figure 22: Overlay management actions using inTune API.

Example overlay properties and their management. Overlay-1 in Figure 21 is comprised of the three VMs representing Web (VM-1), Application (VM-2), and Database (VM-3) tiers, each running in separate islands - A, B and C. The property implemented by this web-server overlay is to maintain the 99th percentile of its response time latencies within a pre-defined upper bound. In its current realization, the overlay relies on the web-server VM (VM-1) to use intelligent application-level request classification [9], and to monitor/detect changes in request traffic patterns. Based on such data, the overlay's internal policy then advises *tuning* of resource allocations for the backend server VMs (VMs 2 and 3), or it may request to increase its own resource allocations. Under heavy request loads (such as during request spikes) and/or with workload consolidation, driven by the web server's observations, it may also request for *urgent, boosted* scheduling of the backend server VM to avoid undue request processing lags.

Figure 22 shows how overlay resource allocation requests are propagated to its constituent islands. Upon observing events pertaining to the maintenance of its property, the overlay monitor initiates an inTune API call to tune or urgently adjust resource allocations that are then executed entirely at the system-level. Application-level overlay monitors specify this request in the form: *inTune_call(Overlay-id, VM-id, resource_adjustment_request)*

(specific inTune calls are further explained in Section 4.3.3). This assumes that the overlay monitor knows how the application components are mapped to overlay VMs and their identifiers. Overlay monitors however need not be aware of underlying islands, as these requests will always be directed to the local island, which will further actuate inter-island coordination on behalf of the overlay, if necessary. An overlay monitor's resource reconfiguration request of the form *(Overlay-id, VM-id, resource_adjustment_request)* is then translated into an inter-island coordination message of the form: *(Overlay-id, Island-id, VM-id, resource_adjustment_request)* at the island-level by the island's local inTune coordinator using knowledge of island -to -overlay VM mapping initiated at overlay creation time.

The application-centric overlay described above uses application-level monitoring to implement proactive policies for maintaining properties expressed in terms relevant to the application. More generally, overlay policies can also be based on system-level monitoring, and they can use reactive vs. proactive methods for actuation. Overlay-2 in Figure 21, for instance, uses observation-based monitors in the Xen hypervisor, which we have added to more conveniently gather per-VM performance and resource utilization state (i.e., using performance counters).

We next describe the inTune coordination API and the inTune coordinator design.

4.3.3 Coordination with inTune

In order to obtain and maintain global overlay properties distributed across multiple distinct islands, island-local events in an overlay must be able to influence resource management actions in remote islands within the overlay, and external events must be able to influence local island management. With inTune, any actuated coordination action directed toward a remote island is propagated via explicit communications. Such message-based coordination ensures that inTune can operate both across strongly (via shared memory) and weakly (via PCI) connected resource islands.

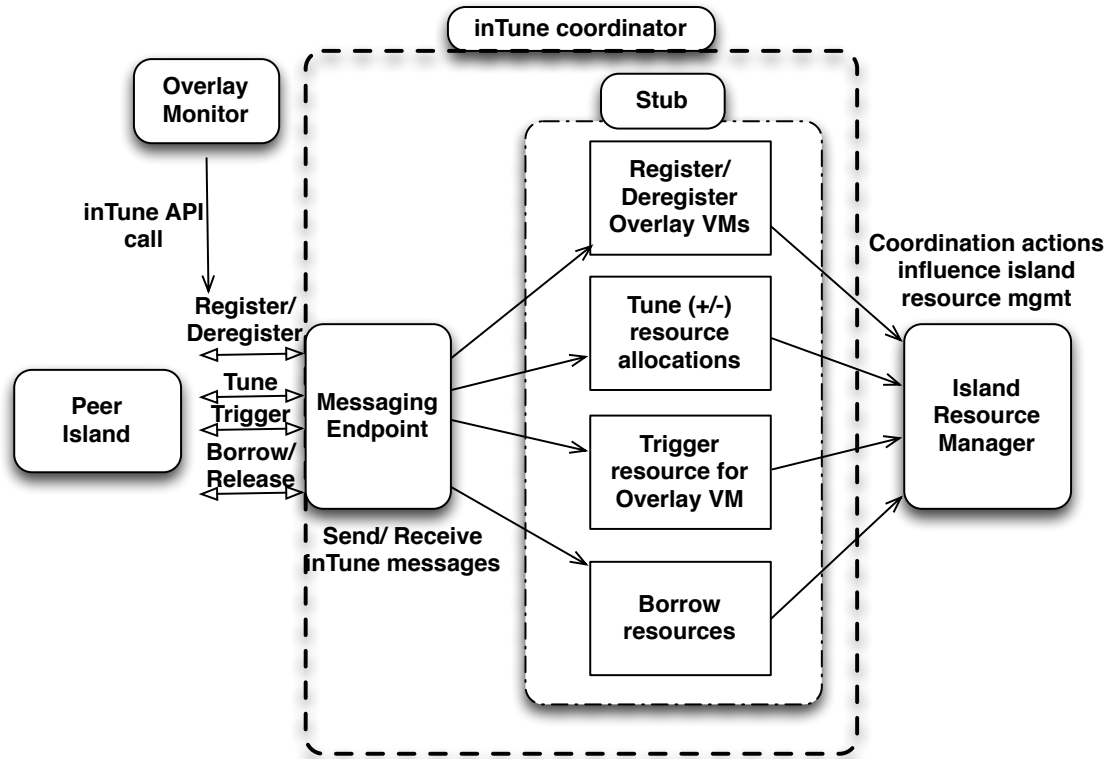


Figure 23: inTune coordinator and inTune messages.

Message channels are used for communication across island coordinators. Their platform-dependent realization may use message queues implemented in shared memory regions or DMA operations across the PCIe interconnect (e.g., as for the prototype x86-IXP platform used in our prior work [91]). Each active overlay uses an independent set of transmit and receive message channels (see Section 4.3.2) between pairs of all its constituent islands, the purpose being to cleanly isolate different overlays' coordination actions. These message channels are created at overlay creation time by the global platform manager. Each constituent island is also initiated with routing state to reach other islands in the overlay, and differentiates between sets of message channels by their overlay identifier.

An **inTune coordinator** (see Figure 23) is responsible for (1) interfacing with the overlay monitor that requests resource re-adjustments to maintain platform- and/or application-level policies leading to inter-island coordination actions. (2) carrying out the necessary explicit communications across islands, to coordinate *advisory* coordination actions across islands, and (3) implementing stub-like functionality for an island, so as to translate remote

messages into local resource management actions. To achieve this, each island's coordinator implements the following set of abstractions, illustrated in Figure 23:

1. **inTune coordination messages** – implement the inter-island coordination mechanisms supported in inTune, summarized in Table 3. As seen in Section 4.3.2, the occurrence of certain events in an overlay (e.g., a peak in client requests at the front-end web server, or when an overlay exhausts its CPU resource) leads to resource re-adjustment requests from overlay monitors. These requests are translated into inTune coordination messages at the island-level. *The subsequent coordination resulting from sending/receipt of these messages amongst islands of an overlay along with their self-reconfiguration serves to accomplish the overlay's resource adjustment requests, and hence the overlay performance property.* inTune supports the following messages – *tune*, *trigger*, *borrow* and *release* to let an island convey coordination actions that may influence resource allocation decisions in peer islands, in response to its local overlay requests.

Motivation behind inTune API. Inter-island coordination APIs in future islands-based systems should possess the following properties: (i) The API needs to be *expressive* enough to successfully translate resource adjustment requests from application and/or platform overlays, so as to meet their properties. (ii) The API should be *generic* and should support a *unifying* set of abstractions to be applicable to heterogeneous resource managers of varied resource types (e.g., CPU, memory). (iii) The API actions should be implemented with low overheads for those resource types.

In choosing to support the specific messages – *tune*, *trigger*, *borrow*, *release* in the current inTune coordination API, we account for the range of resource adjustment request types that overlays may use (for ease of expression), the properties of resource-types managed by islands (for generality) and the overheads of implementing them. We classify resource adjustment requests as those that pertain to (i) increase/decrease or *tuning* of resource shares for overlays over subsequent time periods of application runtime and, (ii) urgent provisioning or *triggering* of resource shares for time-critical use. *Tuning* of resource

shares is limited by island resource capacity. Additional resource adjustment beyond an island's capacity may necessitate *borrowing*, and then *releasing* of resources from other islands. Urgent provisioning of resources may be requested in a time-shared preemptible system, and is translated into a *trigger* message. We believe the API comprising of *tune*, *trigger*, *borrow*, *release* is also generic and can be applied to varied resource types such as CPU and memory. We next discuss coordination messages in further detail and consider their overheads and generality when applied to different resource types.

When an island uses *tune* coordination messages to influence a remote island's management, it permits one island to non-preemptively request dynamic adjustment – *tuning* – of the resource shares assigned to a virtual machine in an overlay in a remote island for a specific resource. This makes it straightforward to increase/decrease a particular resource allocation given to a remote VM, without specifying actual 'amount' parameters. This also enables *tune* to non-intrusively integrate external events with an island's resource management decisions, without complete knowledge of the algorithm used by the island's resource manager. A successful *tune* action for the CPU resource at the remote island involves increasing/decreasing CPU shares of a particular overlay VM along with modifying CPU scheduler state in the remote island. Similarly, *tuning* of memory resources may add/release memory pages for the overlay VM using *ballooning* techniques [101], along with appropriate changes to the VM's page tables. A successful *tune* call may be followed by additional such calls from the overlay monitor, if allocations are still insufficient. The remote island sends a *negative* acknowledgement if it fails to carry out the *tune* action.

trigger, on the other hand, has pre-emptive semantics. If permitted, a *trigger* requests immediate execution of some overlay VM in the remote island. *trigger* may be used to deal with time-critical runtime decisions, where island run-queues are reordered to boost a VM's VCPUs. Implementing *trigger* for the CPU resource involves possibly pre-empting the current VCPU task, saving its state and reordering scheduler run-queues to boost the target VM's VCPUs. Preempting memory is a heavy-weight operation which may involve

saving and restoring a VM's address space. *Triggering* memory resource hence has practical limitations.

tuning a VM's resource shares is limited by the island's resource capacity. *borrow* and *release* are further used to flexibly re-size islands, i.e., to request resources from an external island and add them to the pool of the local island, and to then either release them voluntarily or upon request. When resource borrowing is caused by some specific management overlay, the first attempt is to borrow resources (e.g., CPU cores) from other islands in the same overlay (e.g., in the same multi-tier web application). If a 'borrow' request is not satisfied within an overlay, current policy is to borrow resources beyond the overlay from the island that has the maximum available capacity. To determine idle capacity, previous research offers many design alternatives, including continued distribution of resource freelists across all islands [17]. To avoid incurring the overheads of such periodic updates, our current design resorts to explicit resource utilization queries to other islands when local utilization exceeds some limit. *Borrowing* resources to re-size islands is limited by hardware properties such as cache coherence and core architecture. In our current platforms where islands are software partitions of homogeneous cores, borrowing of cores is a relatively low overhead operation that involves (i) freeing up cores by migrating running tasks to other cores in the lender island, and then further (ii) updating borrower island state to reflect addition of cores to its capacity. We evaluate inTune overheads in Section 4.5.2.3. Finally, the *register* and *deregister* actions are used to notify the inTune co-ordinators of an overlay component's entry into or exit from an overlay.

2. Messaging endpoint – via which the coordinator communicates with peer coordinators. A messaging endpoint is set up during island creation. Routing information about the other island coordinators in an overlay is made available to the endpoint at the time of overlay creation (see Section 4.4). The endpoint drives the sending and receipt of coordination messages on the messaging channels connecting coordinators. Since an island can be part of multiple overlays, the messaging endpoint also implements an *arbitration algorithm*

Table 3: inTune Coordination messages.

tune (resource_type, increase_request/decrease_request, overlay_id, island_id, vm_id) request increase/decrease of resources allocated to a given overlay VM; the actual change to resource allocation is determined locally at the remote island
trigger (resource_type, overlay_id, island_id, vm_id) request immediate boost to resources allocated to a given overlay VM;
borrow_preferred (resource_type, overlay_id, island_id, request_amount) request to temporarily gain exclusive access to resources currently managed by target remote island.
borrow_any undirected borrow request to multiple islands and then choose based on available utilization.
release (resource_type, overlay_id, island_id) release resource to lender island.

when de-queuing coordination messages from multiple overlays (round robin and priority-based in current design).

3. **Scheduler stub** – is a set of functions that translate the information carried in inTune messages to appropriate actions in the target island, and vice versa. For instance, a *tune* call pertaining to the CPU resource will make the stub adjust the CPU shares of VMs according to its own credit system. Such actions will be in accordance with the management state and constraints in the target island.

Arbitration. When islands are part of multiple overlays, an island’s inTune coordinator may receive requests with conflicting control actions, e.g., that compete with other overlay components for an island’s limited resources, or conflict with the island’s platform-centric objectives (e.g., its CPU caps). The current inTune coordinator supports two methods for arbitrating across conflicting requests: (1) a simple round-robin method to execute incoming requests, and (2) a priority-based method that executes requests in order of decreasing overlay priority. Based on these arbitration methods at the island-level, the coordinator executes a ‘winner’ request and sends back negative acknowledgements to conflicting requests. Arbitration can be further optimized, for instance by including the use of historical information to deal with repeated oscillations (e.g., core ping-ponging between two islands due to borrow-release). Under certain conditions, like repeatedly declined (i.e., ‘nack’d’) requests (five successive ones in the current prototype), arbitration decisions are forwarded and handled by the global platform manager. Finally, an overlay’s use of one resource in an

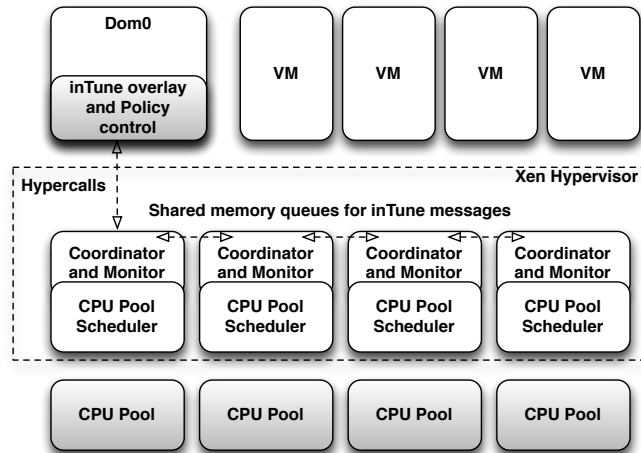


Figure 24: inTune components in Xen hypervisor.

island may need to be arbitrated against other resource managers. For instance, a CPU borrow request may need to be arbitrated against the overlay’s potential memory bandwidth use, as it may interfere with other overlays sharing the memory bandwidth. Arbitration across two application overlays competing to borrow a third island’s CPU resources is described in Section 4.5.3.

4.4 Implementation

Islands and inTune have been implemented in the Xen hypervisor, on two different platforms: 8- and 12-core x86-based machines. The hypervisor is extended to construct multiple scheduling islands (see Figure 24) comprised of pools of CPU, each with a separate, possibly functionally different, CPU scheduler to manage the allocation of the island’s cores. The initial configuration of islands, and scheduler assignment is performed by a global manager running as a user-level process in the privileged domain, Dom0. Beyond island creation its actions include defining inTune overlays by creating and mapping application VMs to islands. After overlay initialization, the Dom0 manager is needed only for (infrequent) global arbitration across multiple overlays. All of its actions, initiated from Dom0, are implemented as hypercalls.

Two periodic threads, implemented as Xen *softirq* contexts on the first CPU in every

island, are responsible for per-island monitoring and as the per-island inTune coordinator respectively. A monitor thread periodically (every 30ms – which is the Xen scheduling epoch) calculates island CPU utilization as part of system-level observations of overlay progress (Section 4.3.2). Any *inTune coordinator* implementation is inherently manager-specific, because it must translate inTune messages into actions understood by the island’s resource manager. One of our current implementations is a coordinator for the Xen Credit scheduler. Regarding the inTune messages described in Section 4.3.3, a *tune* message translates to tuning the CPU cap allocation, which is a parameter that is already supported by the credit scheduler. A *trigger* results in boosting a VCPU to the front of a CPU runqueue, but only if it has sufficient credits to spare in the current scheduling epoch. We added this functionality to the Xen Credit scheduler. For resizing island resources via the *borrow* mechanism, the implementation sends resource utilization queries to all islands within its overlay, and then to other islands, and borrows from the one having maximum available capacity. The coordinator thread polls for incoming inTune messages every 30ms and, based on feedback from the local monitor maintaining island utilization state, applies external control actions locally, if possible. A negative acknowledgement message is sent back to the messaging island in case the control action cannot be applied due to local constraints.

Management overlays. Management overlays encapsulate the platform-level or application-centric properties and run the methods used for obtaining these properties. When creating an overlay, the global manager in Dom0 first retrieves island information (e.g., current load in islands and available resource capacity). It then performs a basic matching of overlay resource requirements to underlying island resources (i.e., admission control) and updates state regarding the overlay’s mapping to islands. The constituent island coordinators are then notified by *registering* the overlay and overlay VMs, establishing pairwise message channels among the corresponding islands, and initiating the island messaging endpoints with routing state (shared memory pointers) to reach remote islands in the overlay. Message channels are implemented as producer-consumer message rings. The global manager

also initiates hypervisor-level monitors that may be distributed across islands and maintain island-local monitoring state like island utilization, but it assumes monitoring within the application to be initiated by the application itself. A combination of such hypervisor-level monitors and/or application-level monitoring periodically updates and checks overlay state pertaining to the overlay property to be attained. An example application-level monitor keeps track of incoming request streams for a web-server overlay, and in reaction to observed traffic patterns, requests resource re-allocations to backend components. This particular monitor implementation is simple and attributes to an additional 50 lines of code to the existing web-server code used in Section 4.5.2.1.

To trigger overlay operations in response to changes to application- or platform-level property, we add inTune coordination hypercalls to overlay monitors that invoke local island coordinators, as was shown in Figure 22. Xen hypercalls to local islands will include the inTune directive (tune, trigger, borrow) and an additional overlay VM identifier, for which resource allocation needs to be adjusted. In our simplified current overlay implementation, we assume that the overlay monitor exactly knows its resource type needs while specifying an inTune request. Our application-level monitors are implemented in the same language runtime as the application (see Section 4.5.2). However, we then use language-specific extensions to invoke the xen-level ‘C’ inTune hypercall API.

An overlay implementation need not be island-aware, as further translation of a VM identifier to an island hosting the VM is performed at the local island, using initial state known at overlay creation time. Inter-island interactions are performed directly by island coordinators.

Since an island may be a part of multiple overlays, arbitration may be needed to mediate across multiple policy overlays; we currently use round-robin and priority-based arbitration. Upon successive negative acknowledgements the global manager is invoked for veto control.

4.5 *Experimental Evaluation*

Jointly, inTune resource islands, inter-island coordination, and management overlays serve to realize application- and system-level policy objectives on scale-out multicore systems. This is demonstrated with experimental evaluations performed on two platforms: (1) a 8-core hyperthreaded dual socket Nehalem x86 host and (2) a 12-core hyperthreaded dual socket Westmere x86 host, both running the Xen 4.0 hypervisor. Experiments use the RUBiS enterprise benchmark, PARSEC and Sequoia parallel benchmarks, Apache Web servers, and Map-Reduce codes. A *platform efficiency* metric is used to evaluate inTune's resource management mechanisms. The metric measures application performance against resource utilization, where a higher *platform efficiency* value using inTune signifies its benefits.

4.5.1 Meeting Global Platform Properties

Consider a platform CPU cap, as an example of a specific platform level property. We achieve this global property as a sum of per-island CPU caps,

Experiment setup. We run the inTune prototype on a 12-core hyperthreaded Intel X5660 Westmere machine, which for this example, is initially configured into four islands of three cores each. We use parallel benchmarks from the PARSEC suite to generate CPU load within each island. In three islands, we deploy one PARSEC application per VM; we leave the first island for Dom0 execution. Given our virtualized environment, three possible relationships may hold for the total number of VCPUs($nV\text{CPU}$) and total number of PCPUs($n\text{PCPU}$): (i) $nV\text{CPU} < n\text{PCPU}$, (ii) $nV\text{CPU} = n\text{PCPU}$, or (iii) $nV\text{CPU} > n\text{PCPU}$. For brevity of presentation, we report a case in which the number of PCPUs in an island is kept constant and equal to 6, changing the number of VCPUs for each deployed VM as 4,6, and 8 for the three cases. The number of PARSEC application threads inside the VM is always equal to the number of VM VCPUs, to avoid oversubscribing within a VM. Every CPU island is managed by a separate Xen credit scheduler and has its own inTune coordinator. Given that the platform has 24 threads, the maximum CPU utilization is 2400. The global

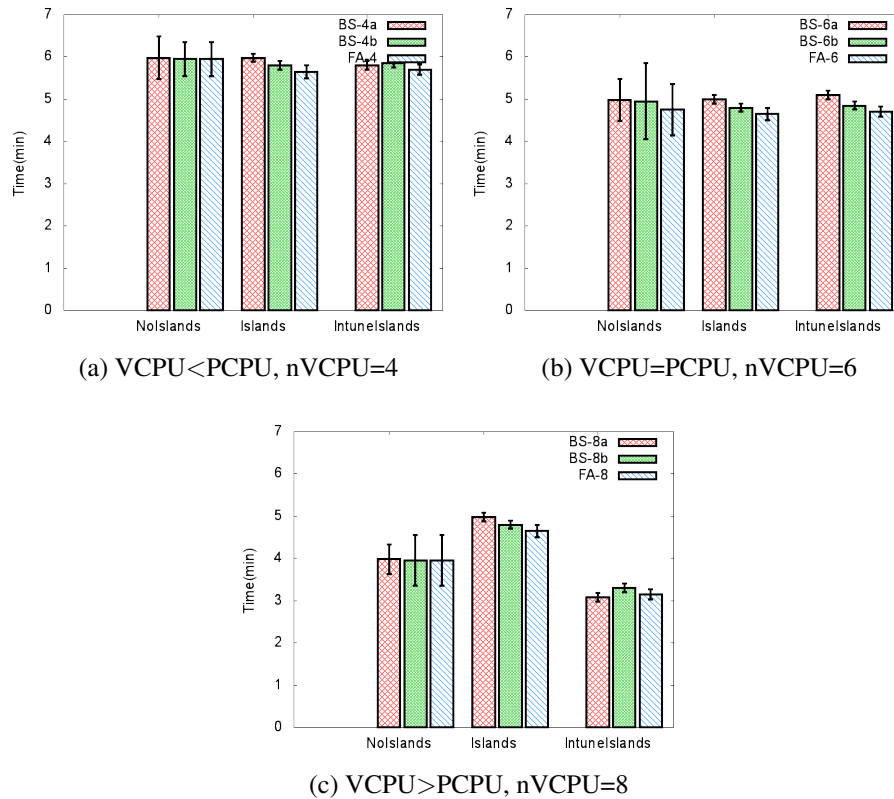


Figure 25: PARSEC: average completion times. BS: Blackscholes FA: Fluidanimate property is to maintain the CPU cap at 2000, by specifying an initial platform-wide overlay with a per-island CPU cap of 500.

Coordination is needed to achieve platform properties like CPU caps. We first compare the *coordinated* management approach (labeled *IntuneIslands*) with the native Xen scheduler (*NoIslands*) with respect to their abilities to maintain platform-wide properties. The *NoIslands* case is the default Xen case, where a single Xen scheduler manages the entire platform. We observe that with inTune coordination, we can continuously ensure the platform-wide CPU cap of 2000, whereas with the default case, this property is violated. In this experiment, coordination among islands is triggered in *reaction to events*. The platform overlay in this case creates local monitors for each island, to track its local CPU cap (refer to Section 4.3.2). When a local monitor detects a violation of its local cap, this indicates that the island has exceeded its resource capacity, and so, the monitor actuates the local inTune coordinator's *borrow* mechanism to borrow a core from a peer island. Overlay

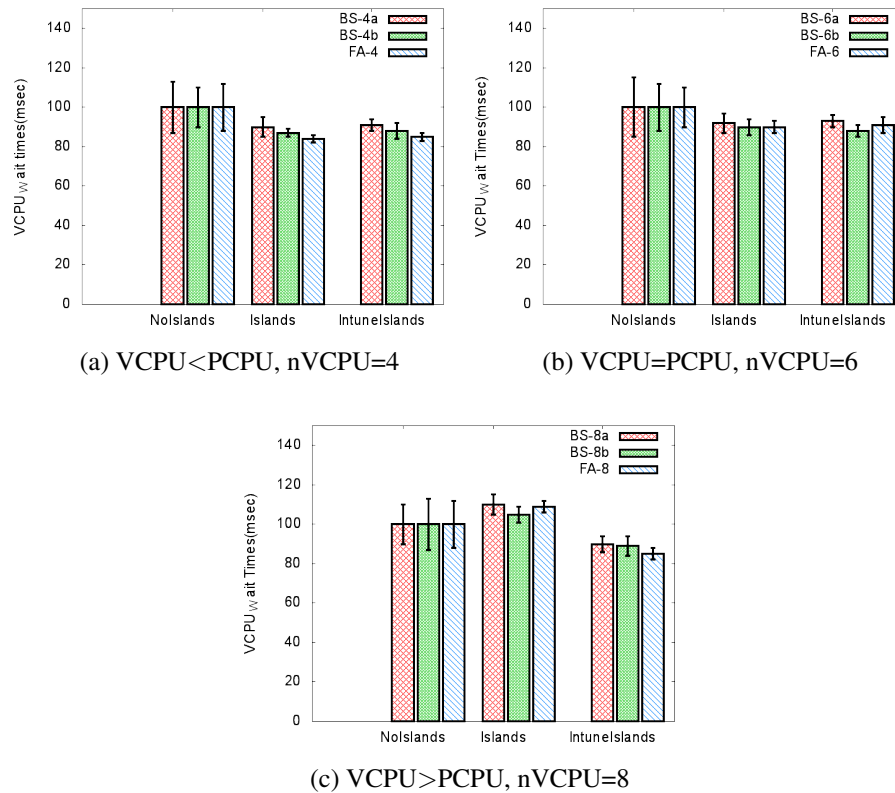


Figure 26: PARSEC: average VCPU_wait times.

monitors in both lender and borrower islands, then, recalculate their local CPU caps without any central controller intervention. The cap enforcement is subject to availability of idle capacity on the platform. Note that the parallel applications used in this experiment exhibit collective phases of computation, followed by relatively idle phases. With inTune, even without knowledge about such application behavior, solely by monitoring CPU utilization, we are able to detect the presence of idle capacity on islands, so that this capacity can then be made available to islands with greater CPU demand. As stated earlier, we conclude from the experiment that *coordination is a necessary means for obtaining desired platform-level properties*.

Utility of island elasticity. Figures 25a, 25b, 25c show the mean completion times of the three benchmark applications along with standard deviation bars for all three nVCPU:nPCPU configurations. The first and third set of bars in each figure correspond to the *NoIslands*

and *IntuneIslands* cases described above. The second set of bars shows times for ‘uncoordinated islands’ (*Islands*), where no coordination takes place across the islands’ CPU managers. The following observations can be made about the results shown in the figure. When $nVCPU \leq nPCPU$, in Figures 25a and 25b, we observe that island creation reduces the level of performance variation seen in the default *NoIslands* case by close to 15%, and it also shows small gains in average completion times. *IntuneIslands* shows similar performance as *Islands*, as the use of *borrow* is never triggered for the case of $nVCPU \leq nPCPU$. However, when $nVCPU > nPCPU$ (see Figure 25c), the use of uncoordinated islands may result in performance degradation, since we are *statically* partitioning the platform, thereby restricting the CPU resources available to applications. In fact, as seen in the *IntuneIslands* case in Figure 25c, with the use of inTune’s mechanisms for *borrowing* and *releasing* resources, flexibly resizing islands reduces performance variation from 10% to less than 0.2% and achieves close to 16% improvements in mean completion time compared to the centralized *NoIslands* case. From Figure 26, we see that with inTune, these gains in performance variation are primarily because of reduced VCPU_wait times for individual VMs in the *IntuneIslands* case compared to the *NoIslands* case, sometimes by up to 20%, as when $nVCPU > nPCPU$, indicating that re-adjustment of island size reduces runqueue lengths, thereby scheduling VCPUs in a more timely manner.

In Table 4, we show the aggregate performance information, and the resulting platform efficiency for the $nVCPU > nPCPU$ case. With inTune, average CPU utilization increases equivalent to one hardware thread, but this increased resource utilization at the appropriate time(s) also delivers lower completion times. The outcome is a higher platform efficiency value ($1 / (\text{CPU-Utilization} * \text{Completion-time})$), thus meeting our global policy objective with application performance benefits.

Table 4: PARSEC – platform efficiency.

	NoIslands	inTune
Completion Times(min)	11.88	9.98
Avg CPU Utilization	938	1040
Platform Efficiency	8.97	9.63

4.5.2 Application Overlays

We next show how inTune’s management overlays and system-level coordination abstractions help achieve performance properties for applications spanning across multiple islands, e.g., in scale-out platforms.

4.5.2.1 Ensuring Predictable Response Times for Multi-tier Web Applications

As a representative multi-tier web application, we use RUBiS, a well-studied auction website prototype modeled after eBay, comprised of an Apache webserver frontend, a Tomcat Servlets application server, and a MySQL database server backend, all deployed in separate Xen hardware virtual machines. Each component VM with three VCPUs is deployed in its own separate island of 2 cores each. Dom0 is deployed in a dual-core island of its own. A RUBiS client is deployed on a separate x86 dual-core host, with 2 GB physical RAM. The standard RUBiS benchmark client has two workload profiles: browsing (read) mix and bid/browse/sell (read-write) mix. From previous work [9] and our own profiling, we know that the browsing mix results in a large amount of web-server and application server processing with practically no database processing. For the read-write mix, database-intensive processing is initiated at the server end. We use this analysis to drive coordination policies that use inTune’s methods.

Overlay setup and coordination methods. This particular realization of the RUBiS application overlay uses proactive policies to actuate inTune coordination by exploiting the application knowledge gained from offline profiling (Overlay-1 in Section 4.3.2). We modify the Apache webserver to classify incoming client requests into bid and browse-type requests, and to also count the numbers of requests of each type every 30 seconds.

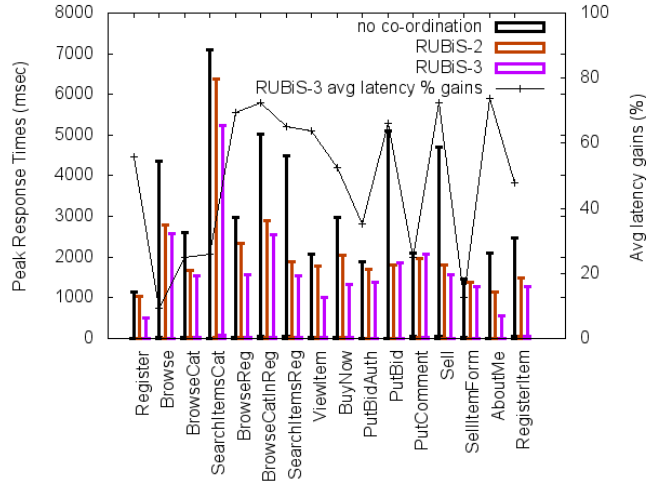


Figure 27: RUBiS Min-Max Response Times. Coordination helps in peak response latency alleviation.

Table 5: RUBiS – Throughput Results.

	Base (req/s)	RUBiS-2 (req/s)	RUBiS-3 (req/s)
Throughput	75req/s	110req/s	126req/s
Avg CPU Utilization	385	475	492
Avg session time(s)	473	454	440
Platform Efficiency	0.19	0.23	0.26

Depending on the request traffic pattern in the previous sample, the webserver propagates this information via a *_tune* hypercall to the hypervisor-resident inTune coordinator on its island. The coordinator then sends a *_tune* coordination message either to the application or the database island, as appropriate. Resource re-allocation within islands due to actuation of *_tune* progresses between three states: (1) no-coordination – a case when there is no coordination and resources are allocated under a *RUBiS-3/2* policy with all three application VMs capped at 1.5 cores within their 2-core islands, (2) *RUBiS-2* – a case when only inTune’s *_tune* messages are used, and the initial allocation changes from 1.5 to 2 cores for the application or the database server; and (3) *RUBiS-3* – a case when both *_tune* and *_borrow* messages are used, and the CPU allocation further changes to a *RUBiS-3* one, with 3 cores for either the application or the database server island, depending on the incoming request pattern. As a result of the receipt of the *_tune* message, the remote island CPU

scheduler may change its CPU allocations from *RUBiS-3/2* to *RUBiS-2*, and vice versa, or go a step further, and make a *_borrow* request, so its allocation can be changed to *RUBiS-3*. The *RUBiS-3/2* policy is representative of cases where web-server resource usage may be capped initially for lower client request rates, and subsequently adapted based on varying load requirements [19].

Elasticity: from ‘tuning’ to ‘borrowing’. Figure 27 shows the min-max response times for different types of RUBiS requests in a read-write browsing mix workload with 480 simultaneous client sessions and the normalized average latency benefits of the *RUBiS-3* configuration over *RUBiS-3/2*. We experiment with varying numbers of simultaneous sessions, starting from 180. On our testbed, the baseline *RUBiS-3/2* configuration can sustain up to 360 sessions, beyond which a *_tune* coordination reconfigures the application and database resource allocations to *RUBiS-2* state. Upon increasing the number of sessions to 480, the application and database server reach maximum CPU utilization in their islands under a *RUBiS-2* allocation. Tuning works only up to capacity limits, so this is when a *_borrow* request is used to get another core to go to *RUBiS-3* allocation.

We also observe that *RUBiS-2* and *RUBiS-3* exhibit reduced variation for almost all request types, sometimes by up to 60%, as in case of the ‘PutBid’ request type. The average response latency gains, also presented in Figure 27 (see right-side Y axis) for the *RUBiS-3* allocation as a percentage value decrease over the baseline *RUBiS-3/2*, show similar trends of decreased response times when the application and database servers receive appropriate CPU allocations for the corresponding request types, sometimes by up to 70%.

Note that for two of the request types, ‘PutBid’ and ‘PutComment’, response time latencies for *RUBiS-3* case are higher than that of *RUBiS-2*. We attribute this increase to the slight overheads experienced when borrowing and releasing cores (Borrow_preferred and Borrow_any mechanisms take 291 and 494 nanoseconds to complete), and the fact that our classification engine is not very sophisticated. However, even with this simple classifier, we are able to see substantial benefits due to the flexible resource allocation achieved via

Table 6: MPI collectives overlay.

	Uncoordinated		inTune	
	Time(ms)	%Var	Time(ms)	%Var
Completion Times (mpi_barrier)	1580	±6.0	1361	±2.93
Completion Times (BS_32_1)	4122	±0.2	4197	±3.2
Completion Times (BS_32_2)	4117	±0.1	4207	±3.1

inTune’s mechanisms. Table 5 shows additional performance metrics for the RUBiS benchmark, where the use of inTune-based coordination clearly results in improved performance and more efficient utilization of platform resources. The platform efficiency metric in Table 5 justifies the resulting higher CPU utilization with increased application throughput *and* lowered response time, thus demonstrating the importance of using inTune mechanisms for coordinated management. We conclude that *inter-island coordination plays an important role in regulating and optimizing the performance of web and datacenter applications.*

4.5.2.2 Latency-sensitive Codes

It is not just the aggregate resource availability that determines application performance, but in addition, it is important ‘how’ and ‘when’ applications’ use of island resources is scheduled. We demonstrate this via an inTune-realized *coscheduling* policy for applications running across multiple islands, making use of inTune’s *trigger* mechanism.

Overlay setup. We use the MPI Barrier benchmark, part of the Sequoia parallel benchmark suite [85], which times MPI_barrier operations for a specified number of iterations across all processes in its communication world. We divide 16 MPI processes among 2 VMs, each with 8 VCPUs and 256M memory. In *inTune* parlance, this MPI world of 2 VMs becomes our application level overlay to which policy objectives may be applied. Each VM is assigned to a separate 4-core island on a 12-core hyperthreaded Intel Westmere machine. The remaining 4 cores are reserved for Dom0. Each island’s CPU resources are

managed by separate credit schedulers. We launch two compute-intensive Blackscholes benchmarks in separate 8 VCPU VMs, and assign each one to the created islands, so that every island has two VMs, one running MPI and the other running Blackscholes. Table 6 shows the barrier performance results of this experiment.

Coordination: coscheduling using trigger. It is well-known that the performance of applications using collective operations like barriers is improved when participating processes are coscheduled. To achieve this, we permit the MPI application overlay (or the communication stacks they use) to explicitly initiate coordination, by providing them with hypercalls directed at the inTune coordinator. Using these hypercalls, the application can inform its island’s coordinator to *trigger* all VMs in the MPI overlay. The local coordinator then sends appropriate *trigger* messages to islands that contain the other VMs to be coscheduled; using the routing information distributed at the time of overlay creation (Section 4.3).

As seen from Table 6, with such coordination, barrier completion time is reduced by 13%, compared to the uncoordinated case, and we also observe reduced variation across consecutive barrier runs. Such *trigger*-based coordination could negatively affect other workloads, but the two Blackscholes VMs experience tolerable performance degradation (less than 4%), which is less than the cumulative advantage seen for the coscheduled MPI VMs. The slight variation for MPI VMs in the *inTune* case can be attributed to ‘advisory’ control design principle used by the coordinator; this means that an attempt to coschedule VCPUs may not always succeed for reasons of fairness within the credit scheduling algorithm. Note, however, that even the worst case performance remains better than the uncoordinated case. We conclude that *inTune interfaces and coordination support help performance-critical codes define relevant policies to improve their performance.*

4.5.2.3 *inTune* overheads

The overheads of our *inTune* implementation consist of: (i) messaging overheads – i.e., time to create an *inTune* message by assembling all relevant parameters, send the message to the target island by copying to the appropriate shared memory channel, and then receive the message, again by copying from the shared memory channel, in our implementation; (ii) arbitration overheads – i.e., time to arbitrate among multiple requests; (iii) scheduling of software interrupt at the target island to schedule the *inTune* coordinator thread, and finally (iv) executing the specific coordination action.

Considering messaging overheads and specific coordination action overheads, a *tune* call needs 212ns, *trigger* needs 293ns, *borrowing* in a two-island overlay requires 381ns, and from among all six islands (of two cores each) needs 594ns (due to additional message routing decision costs). By comparing to a ‘null’ message, which takes 150ns, we observe the enqueueing and dequeueing of *inTune* messages on the shared memory channel represent a dominant factor of total cost. These overheads can be reduced by leveraging future hardware support (e.g., like fast messaging support for control exchanges on Intel’s SCC chip) or by better mapping *inTune*’s explicit message-based communications to the underlying hardware interconnects [11]. Scheduling of software interrupts on our Westmere platform incurs additional overheads between 200-400ns. This only accounts for dispatch overhead, not potential cache misses, etc.

A potential variable cost is incurred due to arbitration across multiple overlays that may queue up within one scheduling epoch of the *inTune* coordinator (30ms). The total number of simultaneous resource readjustment requests at a coordinator is the sum of requests sent by each overlay mapped to an island within the previous 30ms interval. These may also include previously ‘nack’d’ requests. In order to place a bound on this sum, a limit may be applied to the total number of outstanding requests an overlay sends in a time-interval, while using techniques like exponential back off [86] to delay sending subsequent requests. We are still exploring such optimizations to better scale our arbitration methods

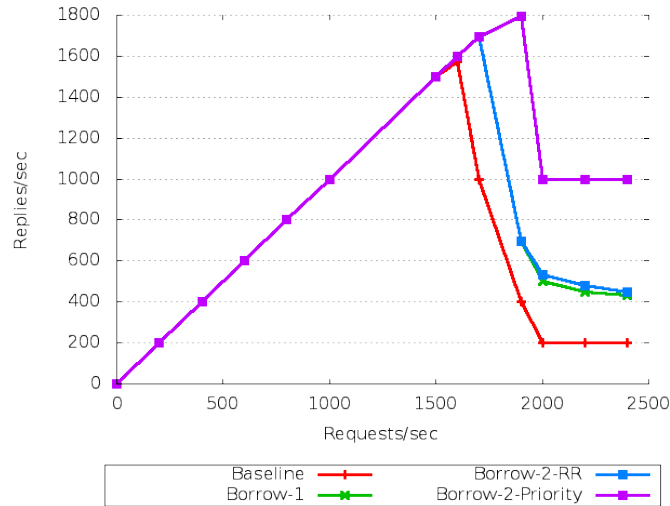


Figure 28: Apache Web server throughput scales with increasing number of cores, especially when its ‘borrow’ request is prioritized over other overlays.

as the number of overlays and islands scale.

4.5.3 Arbitration among Policy Overlays

In the presence of multiple management overlays, there may be multiple control requests to an island’s coordinator that (i) compete for the island’s available resources, or (ii) conflict with the island’s local objectives (e.g., local CPU caps). Such situations require arbitration, as demonstrated by the next use-case in which inTune arbitrates across competing application overlays attempting to borrow a third island’s CPU resources.

Arbitrating between two competing overlays. Consider the inTune prototype running on the 12-core Westmere system (with disabled hyperthreading), initially configured into 4 islands. Dom0 occupies the first island of 4 cores. The second island of 4 cores hosts one VM running the Blackscholes application (of 2 threads) from the PARSEC suite. The third and fourth islands have 2 cores each and host a 4-VCPU Apache v2.4 Web server application and a 4-VCPU Phoenix Map-reduce Word-count [80] application VM, respectively. Each application VM is configured with 2GB of memory, and runs Ubuntu Linux kernel version 3.0.2. The Apache Web server VM services incoming requests for a 14KB static image from an external client system running *htperf*, while the Word-count application

VM processes a 1GB text file using four Map and Reduce threads. At the beginning of the experiment, the 4-core Blackscholes island is least utilized and has an available capacity of 3 cores. The Apache Web server island starts capping its CPU resource, and hence experiences a throughput degradation when the *httperf* request rate starts exceeding 1500 requests/second, as seen in the ‘Baseline’ configuration in Figure 28. The Word-count island also caps its CPU, and both the Web-server and Word-count islands attempt to borrow a CPU core each from the Blackscholes island. The Blackscholes island satisfies both requests, reconfiguring itself to 2 cores, and lending one core each to both requesting islands. ‘Borrow-1’ in Figure 28 shows the Web-server throughput scaling with an additional core – by 10% compared to ‘Baseline’ and sustains almost double the ‘Baseline’ throughput at higher request rates.

A further increase in the *httperf* request rate causes the web-server island to again start capping CPU, matched with a similar capping of CPU in the Word-count island, causing both islands to again send ‘borrow’ requests to the Blackscholes island. However, as the Blackscholes island has only one core available, its inTune coordinator needs to arbitrate between the two requests. With *round-robin* arbitration, the Blackscholes island satisfies the Word-count island’s request, denying a core to the Webserver island (see ‘Borrow-1-RR’ case in Figure 28). However, we noticed that the Word-count application does not gain much with an additional core (completion time reduces by 20% with the first borrow request, but improves by only an incremental 5% with the second borrowed core). Hence, we next experiment with *priority-based* arbitration, which prioritizes the Web-server overlay over the Word-count overlay, thereby lending the available core to the Web-server island, instead (see throughput gains and scaling for ‘Borrow-2-Priority’ case in Figure 28 – sustaining almost four times the throughput at higher request rates). We conclude that *arbitrating across competing overlay requests is a necessary and important function of the inTune coordinator*, enabling it to make better decisions while maintaining application and platform properties, especially in resource-constrained, consolidated systems.

Discussion. In the current implementation of our inTune prototype, the arbitration policy supported by the inTune coordinator (round-robin vs. priority-based) is configurable and can be chosen via the global platform manager at the time of coordinator initialization. As seen in Section 4.3, if an overlay’s requests are successively nack’d over five intervals of 30ms by an island’s inTune coordinator due to insufficient resources, the inTune coordinator invokes the global platform manager for overall arbitration. Current policy of the global platform manager is to initiate capping CPU usage of another overlay chosen at random, and to re-allocate available CPU resource to the ‘starved’ overlay. We are exploring more sophisticated arbitration policies for the platform manager that provide weighted performance degradation for all applications [63]. In the event of insufficient resources to provide acceptable performance levels to all applications, necessary interactions with higher-level schedulers and load balancers (at the data-center level) should take place.

Finally, the overall fairness and stability of the inTune coordinator can be further enhanced by accounting for the *history* of previous decisions.

4.5.4 Summary of Evaluation

Evaluation use-cases demonstrate the utility of inTune, and address the two issues in managing the resources of scale-out platforms introduced in Section 4.1.

Maintaining platform-centric and application performance properties. Section 4.5.1 shows for multi-island many-core systems, a platform-wide property of CPU cap can be implemented, using inTune policy overlays and its *borrow* and *release* mechanisms. Sections 4.5.2.2 and 4.5.2.1 demonstrate the necessity of *trigger*, *tune*, and *borrow* for implementing application-specific overlay properties. In all such cases, *trigger* and *tune* are the initial mechanisms of choice, with *borrowing* and *realizing* used when capacity limits are reached. We can also claim, based on the results seen in these use-cases, that coordination using inTune (i) is important for managing platforms and meeting application performance requirements (supported by the consistently improved performance and lower performance

variation we see with coordinated islands compared to the centralized Xen scheduler) and (ii) is sufficiently versatile for representing a variety of higher level policies for island-based platforms. In Section 4.5.3, we additionally demonstrate the ability of the inTune coordinator to arbitrate amongst multiple consolidated overlays. Arbitration will be particularly important in consolidated data center systems, to ‘better’ allocate resources when realizing multiple properties with resource-constrained platforms.

Standard interfaces for diverse resource managers. With the coexistence of multiple, functionally different and/or heterogeneous resource managers becoming the norm in data center systems, we argue that rather than trying to design one scheduler to deal with all platform resources [15], using multiple scheduler islands and adding inter-island coordination interfaces is preferable. With this approach, custom interfaces and interactions exist only to ‘translate’ control actions between specific resource managers (e.g., real-time vs. credit-based schedulers) and the controllers with which they interact, whereas controllers interact via well-defined and -proven techniques based on the principles of online system control [61] and elasticity.

4.6 Related Work

Resource islands. The concept of resource islands used in Helios [71] has its roots in earlier work that includes Cellular-Disco [30], Hive [21], and K42 [50]. While Helios [71] uses satellite kernels to account for heterogeneous runtimes, Hive [21] uses resource-partitions for fault-containment, and K42 [50] uses them to exploit locality. The implementation of islands via hypervisors in our work is similar to the cell-partitions approach followed in the Cellular-Disco system [30], with the key additional contribution of general system-level support for abstractions and methods to coordinate these islands for achieving management goals. These abstractions permit arbitrary interactions amongst coordinating islands compared to fixed gang-scheduling in the Cellular-Disco system. Also, inTune accommodates

cross-VM application dependencies and hence, coordinates VCPUs across VMs, not limited to a single VM. inTune complements micro- and exo-kernels [24], by building resource management abstractions on top of a minimal kernel using message passing and its contributions related to resource management apply to systems software in general: including island-based microkernels with island resources managed by library OSs [24]. Recent work includes the Tessellation OS [56] that creates space-time partitions (STPs) encapsulating applications and OS services within ‘resource containers’ [8] for performance isolation. STPs are complementary to our resource-level abstraction - ‘resource-islands’ where STPs may be time-multiplexed by OS-level island schedulers, similar to our hypervisor-based islands scheduling application VMs. In addition, applications need not be aware of the island abstraction, as STPs are exported to applications.

Messaging and coordinated scheduling. Communication primitives and messaging abstractions for resource management have been studied for grids and clusters – CCL [7] and Condor [27] being two widely-studied examples. In recent research, the use of message passing and RPC for inter-core communication has seen renewed interest in Barrelfish [11]. inTune adopts some lessons from such prior work in terms of its loose coupling across resource managers, but it extends the messaging based system of Barrelfish [11] with additional semantics focused on resource management. Also relevant to our work are the two-level scheduling problems discussed in previous research using scheduler activations [4], specifically in terms of our approach of exchanging only relevant and limited information about resource islands via relaying selective control, a similar approach is used by scheduler activations between user and kernel-level schedulers.

From multi-agent systems to inTune. The choice of the abstractions and coordination primitives supported by the inTune architecture is not accidental. Instead, since inTune is intended to enable and support coordination, its features are defined based on a substantial body of work in system controls and multi-agent systems. In particular, in reference to distributed control theory and multi-agent systems for power engineering [61], inTune

coordinators are similar to intelligent agents in that they are: (1) *proactive*, initiating their own local control actions, (2) *reactive*, such that they can react to local monitored events; (3) *social*, such that they communicate management actions with other coordinators; (4) *autonomous*, such that they have local control and treat external control as ‘advisory’.

4.7 Chapter Summary

A key problem for future large-scale and heterogeneous multicore platforms is how to manage applications and their performance properties spanning across their multiple ‘resource islands’, each with their own resource managers and methods. InTune realizes an approach and framework for developing and evaluating future management methods, using as a guiding theme, coordinated management in which there are explicit interactions between different islands’ resource managers taken on behalf of applications to maintain their properties. inTune lets islands cooperate using standard interfaces and mechanisms on behalf of applications hence abstracting away their implementation details.

inTune is implemented as an extension of the Xen hypervisor, and its management overlays using message-based coordination primitives are shown to operate on the Intel Nehalem x86-based manycore machine used in this thesis. Experimental evaluations for both web and parallel applications demonstrate the utility and importance of the coordinated islands approach in meeting applications’ end-to-end needs (with improved throughput, predictability, and lower response times) than when using a single homogeneous Xen hypervisor.

CHAPTER V

PREVIOUS WORK

Previous chapters delineated the related work relevant to the thesis components described in the chapter. We summarize some of the previous work here.

A unifying theme of this thesis has been to enable shared multicore servers to provide the levels of performance, predictability and efficiency sought for applications and platform resource management. Virtual Platforms leverage parallel research in distributed systems for storage resources [63, 103], and extend our contributions to include multi-dimensionality in compute, memory and memory bandwidth resources. The specific challenges that arise out of intricate memory and interconnect designs in hardware platforms to provide isolation are also studied in previous work that deals with interference in the memory subsystem [49, 111, 13, 48, 66]. Such methods aim to increase overall system throughput, but because they do not account for application-specific sensitivities, they have limited ability to run datacenter applications at the levels of predictability and timeliness they require. The Bubble-up framework [60, 108] deals with one important configuration of consolidated datacenter workloads in which latency-sensitive applications receive performance guarantees when improving overall platform utilization. Virtual Platforms, in comparison recognize sensitivities of all running applications, costs of reconfiguration methods and account for NUMA platforms as well. In order to leverage possible advancements in hardware to create shares of cache and memory bandwidth [20, 5, 84, 69, 42, 79], VPs lay the foundation for better hardware/software interfaces (by making use of topology knowledge and hardware performance counters), and further deals with dynamic nature of applications using flexible higher-level policies.

Finally, recent work in operating systems [11, 71] focuses on redesigning OS primitives to better support new hardware trends of increasing cores on chip, and the advent of heterogeneity. inTune and the islands approach extend the message-based primitives of Barrelfish to include semantics pertaining to managing the various diverse resources on such platforms. Exploration of such standard interfaces that provide some visibility into hypervisor resource management, and exposed to higher-level policies is inTune's novel contribution, which has not yet been addressed in previous work.

CHAPTER VI

CONCLUSIONS

This thesis presents a rigorous approach to managing shared multicore server platforms efficiently by facilitating abstractions, and representative methods at the system-level that improve their performance and isolation properties. The concrete contributions toward better management of shared multicore platforms are as follows.

6.1 Contributions

1) New abstractions – *Virtual Platforms (VP)* – make commitments of resource shares to entire applications and manage shared server resources in ways that maintain isolation as a first-class management principle. Virtual Platforms are multi-resource dimensional abstractions, and incorporate knowledge of application-specific characteristics such as resource intensities and sensitivities to various resource types, as well as performance properties of applications.

2) Application properties and VPs share platform resources, and there need to be methods that monitor and mediate their usage of platform resources so as to meet performance objectives of possibly multiple applications, and also to isolate their resource shares for better performance predictability. The methods that facilitate this management of VPs need to be efficient in terms of their costs and overheads, and consider application-specific sensitivities to manage them effectively. Toward these goals, this thesis introduces *Merlin*: a resource allocator that manages elastic resource shares of highly dynamic application mixes. Merlin is guided by online models and metrics which capture (a) interference effects seen for applications by assessing their sensitivity to each of the the platform’s shared resource points and their resource usage; (b) costs of reconfiguration methods to manage ‘elasticity’, and how they might ‘hurt’ other applications’ resource shares and properties.

Using these metrics, Merlin enables better management of resources by improving the efficiency of platform management, and also choosing more ‘effective’ VP configurations that improve performance and predictability for all applications.

3) These management methods are implemented at the hypervisor layer in a platform-agnostic manner and within modest overhead bounds. The software methods shown and evaluated present opportunities for improving the hardware/software interfaces and functionality of future server systems used in consolidated and cloud data-centers. Implemented in the Xen hypervisor using black-box monitoring, the VP approach does not require guest operating system or application inputs. Evaluations with various application mixes demonstrate the ability of VP-enabled hypervisors to improve isolation for consolidated server loads, i.e., to strongly reduce the variability in performance seen for consolidated applications (5-10%), in addition to significant improvements in their worst-case performance degradation (40-50%).

4) Finally, as single-node multicore platforms evolve further from small numbers of homogeneous cores toward multiple sets or islands of potentially heterogeneous cores residing on a single chip, such platforms will have multiple resource managers managing their respective ‘islands’ of resources. Though geared toward improved scalability and functionality, for applications spanning across multiple diverse resource islands to realize such opportunities, systems software must make it easier for them to interact with the island managers; and also help islands based systems achieve end-to-end performance properties via joint coordination amongst their island managers. In order to meet the challenges in maintaining performance objectives on future ‘scale-out’ platforms, this thesis contributes *inTune*: a framework for inter-island operation, offering APIs and mechanisms that permit applications (and their virtual platforms) to interface with resource islands and their resource managers to jointly achieve application performance guarantees and global platform-level properties.

6.2 Concluding Remarks

The technical contributions of this thesis enable platforms to meet performance objectives of multiple applications and improve the isolation needs sought for their properties, and the efficiency sought for platforms. Coordination amongst multiple resource managers of various resource types will be increasingly important in the future as server hardware complexity and their interconnections evolve. inTune interfaces that let resource managers interact with each other coupled with Virtual Platforms that encapsulate application resource shares and their properties help build systems software that can better engage with evolving hardware and let applications utilize their scalable and diverse functional properties.

CHAPTER VII

FUTURE WORK

This thesis presents several interesting avenues for future work. The Virtual Platforms abstraction and their resource management using Merlin in particular can be extended to other resource types such as storage and networking resources; and the entire approach of monitoring and mediating the use of shared resources is easily extensible to these resource types to improve their isolation properties. As future hardware evolves and integrates more heterogeneity in memory systems (e.g., non-volatile memories, 3-D stacked DRAM and deeper hierarchies), sensitivities that applications may exhibit to different resources will be more intricate; as well as understanding memory latency and its effect on application performance will be important. The Virtual Platforms infrastructure provides the basic hooks and interfaces to understand these effects and the costs involved in managing them. It would be interesting to further adapt these hardware-software interfaces to leverage new support in hardware and/or cater to more complex hardware designs. Furthermore, it would be interesting to apply higher-level policy models for pricing the usage of these resources for applications to construct more sophisticated ‘pay-as-you-go’ models in cloud infrastructures.

A near-term future direction for inTune coordination is exploring its use for meeting platform-level objectives like power budgets [67], while continuing to evaluate its base coordination mechanisms and arbitration techniques with respect to scalability as hardware platforms evolve. We also believe that the two underlying themes of this thesis namely mitigation of interference and coordination across multiple diverse resource managers to

attain global properties is also applicable to maintaining platform power and thermal properties. Input from guest VMs and applications that may employ more sophisticated monitoring techniques or the data layouts of application processes within the VM can be used to further enhance VP management methods. Finally, it would be interesting to extend the approach of 'islands' and 'Virtual Platforms' to operate across multiple node platforms like those present in high end parallel machines or in the cloud data centers.

REFERENCES

- [1] ADILETTA, M., “The Next Generation of Intel IXP Network Processors,” *Intel Technology Journal*, 2002.
- [2] AGARWALA, S., ALEGRE, F., SCHWAN, K., and OTHERS, “E2EProf: Automated End-to-End Performance Management for Enterprise Systems,” in *DSN*, 2007.
- [3] AMAZON. <http://aws.amazon.com/ec2/instance-types/>. ”Amazon EC2 cloud Instance Types”.
- [4] ANDERSON, T. E., BERSHAD, B. N., LAZOWSKA, E. D., and LEVY, H. M., “Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism,” *ACM TOCS*, 1992.
- [5] AWASTHI, M., NELLANS, D. W., SUDAN, K., BALASUBRAMONIAN, R., and DAVIS, A., “Handling the Problems and Opportunities Posed by Multiple On-chip Memory Controllers,” in *PACT*, 2010.
- [6] BAILEY, K., CEZE, L., GRIBBLE, S. D., and LEVY, H. M., “Operating System Implications Of Fast, Cheap, Non-volatile Memory,” in *Proceedings of the 13th USENIX conference on Hot topics in operating systems*, 2011.
- [7] BALA, V., BRUCK, J., MEMBER, S., CYPHER, R., ELUSTONDO, P., HO, A., TIEN HO, C., KIPNIS, S., and SNIR, M., “CCL: A Portable and Tunable Collective Communication Library for Scalable Parallel Computers,” *IEEE Transactions on Parallel and Distributed Systems*, 1995.
- [8] BANGA, G. and DRUSCHEL, P., “Resource Containers: A New Facility For Resource Management in Server Systems,” *OSDI ’99*.
- [9] BARHAM, P., DONNELLY, A., ISAACS, R., and MORTIER, R., “Using Magpie for Request Extraction and Workload Modelling.,” in *OSDI*, 2004.
- [10] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., and WARFIELD, A., “Xen and the Art of Virtualization,” in *SOSP*, 2003.
- [11] BAUMANN, A., BARHAM, P., DAGAND, P.-E., HARRIS, T., ISAACS, R., PETER, S., ROSCOE, T., SCHÜPBACH, A., and SINGHANIA, A., “The Multikernel: A New OS Architecture for Scalable Multicore Systems,” in *SOSP*, 2009.
- [12] BIENIA, C., KUMAR, S., and OTHERS, “The PARSEC Benchmark Suite: Characterization and Architectural Implications,” in *PACT*, 2008.

- [13] BLAGODUROV, S., ZHURAVLEV, S., FEDOROVA, A., and KAMALI, A., “A case for NUMA-aware Contention Management on Multicore Systems,” in *Usenix ATC*, 2011.
- [14] BODIK, P., FOX, A., FRANKLIN, M. J., JORDAN, M. I., and PATTERSON, D. A., “Characterizing, Modeling, and Generating Workload Spikes for Stateful Services,” in *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC ’10, 2010.
- [15] BOYD-WICKIZER, S., CLEMENTS, A. T., MAO, Y., PESTEREV, A., KAASHOEK, M. F., MORRIS, R., and ZELDOVICH, N., “An Analysis of Linux Scalability to Many Cores,” in *OSDI*, 2010.
- [16] BRUNO, J., GABBER, E., OZDEN, B., and SILBERSCHATZ, A., “The Eclipse Operating System: Providing Quality of Service via Reservation Domains,” in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATC ’98, 1998.
- [17] BUTT, A., ZHANG, R., and HU, Y. C., “A Self-Organizing Flock of Condors,” in *SC ’03*, 2003.
- [18] CECCHET, E., MARGUERITE, J., and ZWAENEPOEL, W., “Performance and Scalability of EJB Applications,” in *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA ’02, (New York, NY, USA), ACM, 2002.
- [19] CHANDRA, A., GONG, W., and SHENOY, P., “Dynamic Resource Allocation for Shared Data Centers Using Online Measurements,” in *IWQoS 2003*, pp. 381–398, Springer, 2003.
- [20] CHANG, J. and SOHI, G. S., “Cooperative Cache Partitioning for Chip Multiprocessors,” in *SuperComputing*, 2007.
- [21] CHAPIN, J., ROSENBLUM, M., DEVINE, S., LAHIRI, T., TEODOSIU, D., and GUPTA, A., “Hive: Fault Containment for Shared-memory Multiprocessors,” *SIGOPS Oper. Syst. Rev.*, 1995.
- [22] CHEN, F., KODIALAM, M., and LAKSHMAN, T., “Joint Scheduling of Processing and Shuffle Phases in Mapreduce Systems,” in *INFOCOM, 2012 Proceedings IEEE*, IEEE, 2012.
- [23] DASHTI, M., FEDOROVA, A., FUNSTON, J., GAUD, F., LACHAIZE, R., LEPELERS, B., QUEMA, V., and ROTH, M., “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.
- [24] ENGLER, D. R., KAASHOEK, M. F., and OTHERS, “Exokernel: An Operating System Architecture for Application-level Resource Management,” in *SOSP*, 1995.

- [25] FEDOROVA, A., SELTZER, M., and SMITH, M. D., “Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler,” *PACT*, 2007.
- [26] FERDMAN, M., ADILEH, A., KOCBERBER, O., VOLOS, S., ALISAFABEE, M., JEVDJIC, D., KAYNAK, C., POPESCU, A. D., AILAMAKI, A., and FALSAFI, B., “Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware,” in *Asplos*, 2012.
- [27] FREY, J., TANNENBAUM, T., LIVNY, M., FOSTER, I., and TUECKE, S., “Condor-G: A Computation Management Agent for Multi-Institutional Grids,” *Cluster Computing*, 2002.
- [28] GAMSA, B., KRIEGER, O., APPAVOO, J., and STUMM, M., “Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System,” in *OSDI*, 1999.
- [29] <http://tinyurl.com/pps7ej>, 2009. Michael Jackson Death: An inside look at how Google handled an extraordinary day in search.
- [30] GOVIL, K., TEODOSIU, D., HUANG, Y., and ROSENBLUM, M., “Cellular Disco: Resource Management using Virtual Clusters on Shared-memory Multiprocessors,” *ACM Trans. Comput. Syst.*, 2000.
- [31] GOVINDAN, S., CHOI, J., and NATH, A. R., “Xen and Co.: Communication-Aware CPU Management in Consolidated Xen-Based Hosting Platforms,” *IEEE Transactions on Computers*, 2009.
- [32] GULATI, A., AHMAD, I., and WALDSPURGER, C. A., “PARDA: Proportional Allocation of Resources for Distributed Storage Access,” in *Proceedings of the 7th conference on File and storage technologies*, FAST '09, USENIX Association, 2009.
- [33] GUO, F. and SOLIHIN, Y., “A Framework for Providing Quality of Service in Chip Multi-Processors,” in *MICRO*, 2007.
- [34] GUPTA, V., SCHWAN, K., TOLIA, N., TALWAR, V., and RANGANATHAN, P., “Pegasus: Coordinated Scheduling in Virtualized Accelerator Systems,” in *Usenix ATC*, 2011.
- [35] HONG, S. and KIM, H., “An Analytical Model for a GPU architecture with Memory-Level and Thread-Level Parallelism Awareness,” in *ISCA*, 2009.
- [36] HOWARD, J., DIGHE, S., HOSKOTE, Y., VANGAL, S., FINAN, D., RUHL, G., JENKINS, D., and OTHERS, “A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS,” *ISSCC*, 2010.
- [37] HUANG, W., RAJAMANI, K., STAN, M., and SKADRON, K., “Scaling with Design Constraints - Predicting the Future of Big Chips,” in *IEEE Micro*, 2011.

- [38] IBM, “A Wire-speed Power Processor: 2.3 GHz 45 nm SOI with 16 cores and 64 threads,” *ISSCC*, 2010.
- [39] INTEL. <http://goo.gl/UjX9q>. ”Intel Software Developers’ Manual Vol 3C.”.
- [40] INTEL. <http://goo.gl/5lpY8>, 2010. ”Intel 1000 Core Chip”.
- [41] IYER, A. and MARCULESCU, D., “Power and Performance Evaluation of Globally Asynchronous Locally Synchronous Processors,” in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, 2002.
- [42] IYER, R., ILLIKKAL, R., TICKOO, O., ZHAO, L., APPARAO, P., and NEWELL, D., “VM3: Measuring, Modeling and Managing VM Shared Resources,” *Computer Networks*, 2009.
- [43] KAMBADUR, M., MOSELEY, T., and OTHERS, “Measuring Interference Between Live Datacenter Applications,” in *SC*, 2012.
- [44] KASERIDIS, D., STUECHELLI, J., and OTHERS, “A Bandwidth Aware Memory Subsystem Resource Management using Non-invasive Resource Profilers for Large CMP Systems,” in *HPCA*, 2010.
- [45] KAZEMPOUR, V., KAMALI, A., and FEDOROVA, A., “AASH: An Asymmetry-aware Scheduler for Hypervisors,” in *VEE*, 2010.
- [46] KESAVAN, M., GAVRILOVSKA, A., and SCHWAN, K., “Differential Virtual Time (DVT): Rethinking I/O Service Differentiation for Virtual Machines,” in *SoCC*, 2010.
- [47] KIM, Y., HAN, D., and OTHERS, “ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers,” in *HPCA*, 2010.
- [48] KNAUERHASE, R. C., BRETT, P., and OTHERS, “Using OS Observations to Improve Performance in Multicore Systems.,” *IEEE Micro*, 2008.
- [49] KOH, Y., KNAUERHASE, R., BRETT, P., BOWMAN, M., WEN, Z., and PU, C., “An Analysis of Performance Interference Effects in Virtual Environments,” in *ISPASS*, 2007.
- [50] KRIEGER, O., AUSLANDER, M., ROSENBERG, B., WISNIEWSKI, R. W., XENIDIS, J., DA, D., OSTROWSKI, S. M., BUTRICO, M., MERGEN, M., WATERLAND, A., and UHLIG, V., “K42: Building a Complete Operating System,” in *EuroSys*, 2006.
- [51] KUMAR, S., TALWAR, V., and OTHERS, “vManage: Loosely Coupled Platform and Virtualization Management in Data Centers,” in *ICAC*, 2009.
- [52] KUMAR, V., SCHWAN, K., and OTHERS, “A State-space Approach to SLA based Management,” in *NOMS*, 2008.

- [53] LEE, M., KRISHNAKUMAR, A. S., KRISHNAN, P., SINGH, N., and YAJNIK, S., “Supporting Soft Real-time Tasks in the Xen Hypervisor,” in *VEE*, 2010.
- [54] LINGJIA, T., MARS, J., ZHANG, X., HAGMANN, R., HUNDT, R., and TUNE, E., “Optimizing Google’s Warehouse Scale Computers: The NUMA Experience,” in *The 19th IEEE International Symposium on High Performance Computer Architecture*, IEEE, 2013.
- [55] LINKEDIN. <http://project-voldemort.com/>. ”Project Voldemort: A Distributed Key-Value store”.
- [56] LIU, R., KLUES, K., BIRD, S., HOFMEYR, S., ASANOVIC, K., and KUBIATOWICZ, J., “Tessellation: Space-Time Partitioning in a Manycore Client OS,” in *HotPar*, 2009.
- [57] LOH, G. H., “3D-Stacked Memory Architectures for Multi-core Processors,” in *ISCA*, 2008.
- [58] MAGENHEIMER, D., “Memory Overcommit - Without the Commitment,” in *Xen-Summit*, 2008.
- [59] MAJO, Z. and GROSS, T. R., “Memory Management in NUMA Multicore Systems: Trapped between Cache Contention and Interconnect Overhead,” in *Proceedings of ISMM’11*, 2011.
- [60] MARS, J., TANG, L., HUNDT, R., SKADRON, K., and SOFFA, M. L., “Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers Via Sensible Co-locations,” in *MICRO*, 2011.
- [61] MCARTHUR, S., DAVIDSON, E., CATTERSON, V., DIMEAS, A., HATZIARGYRIOU, N., PONCI, F., and FUNABASHI, T., “Multi-Agent Systems for Power Engineering Applications Part I: Concepts, Approaches, and Technical Challenges,” in *IEEE Transactions on Power Systems*, 2007.
- [62] <http://lse.sourceforge.net/numa/>, 2002. Linux Scalability effort.
- [63] MERCHANT, A., UYSAL, M., PADALA, P., ZHU, X., SINGHAL, S., and SHIN, K., “Maestro: Quality-of-Service in Large Disk Arrays,” in *Proceedings of the 8th ACM international conference on Autonomic computing (ICAC)*, 2011.
- [64] <http://cucis.ece.northwestern.edu/projects/DMS/MineBench.html>, 2006. The Minebench Application Suite.
- [65] MISHRA, A. K., SRIKANTIAH, S., KANDEMIR, M. T., and DAS, C. R., “CPM in CMPs: Coordinated Power Management in Chip-Multiprocessors,” in *SuperComputing*, 2010.
- [66] NATHUJI, R., KANSAL, A., , and GHAFFARKHAH, A., “Q-clouds: Managing Performance Interference Effects for QoS-aware Clouds,” in *EuroSys ’10*, 2010.

- [67] NATHUJI, R. and SCHWAN, K., “VPM Tokens: Virtual Machine-aware Power Budgeting in Datacenters,” *Cluster Computing*, 2009.
- [68] <http://bit.ly/5eNDKy>, 2009. Intel Nehalem Processors.
- [69] NESBIT, K. J., LAUDON, J., and SMITH, J. E., “Virtual Private Caches,” *SIGARCH Comput. Archit. News*, 2007.
- [70] NESBIT, K. J., LAUDON, J., and SMITH, J. E., “Virtual Private Machines: A Resource Abstraction,” tech. rep., In University of Wisconsin - Madison, ECE TR, 2007.
- [71] NIGHTINGALE, E. B., HODSON, O., MCILROY, R., HAWBLITZEL, C., and HUNT, G., “Helios: Heterogeneous Multiprocessing with Satellite Kernels,” in *SOSP*, 2009.
- [72] NIYOGI, K. and MARCULESCU, D., “Speed and Voltage Selection for GALS Systems Based on Voltage/frequency Islands,” in *Proceedings of the 2005 Asia and South Pacific Design Automation Conference, ASP-DAC '05*, 2005.
- [73] NOVAKOVIC, D., VASIC, N., NOVAKOVIC, S., KOSTIC, D., and BIANCHINI, R., “DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments,” in *Usenix ATC*, 2013.
- [74] <http://nutch.apache.org/#What+is+Apache+Nutch>, 2005. Nutch Apache Web Crawler Project.
- [75] ONGARO, D. and COX, A. L., “Scheduling I/O in Virtual Machine Monitors,” in *VEE*, 2008.
- [76] <http://bit.ly/8oe8uU>, 2007. AMD Opteron six-core processors.
- [77] PETER, S., SCHÜPBACH, A., BARHAM, P., BAUMANN, A., ISAACS, R., HARRIS, T., and ROSCOE, T., “Design Principles for End-to-End Multicore Schedulers,” in *HotPar'10*, 2010.
- [78] PU, C., SAHAI, A., and OTHERS, “Observation-Based Approach to Performance Characterization of Distributed n-Tier Applications,” 2006.
- [79] QURESHI, M. K. and PATT, Y. N., “Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches,” in *MICRO*, 2006.
- [80] RANGER, C., RAGHURAMAN, R., PENMETSU, A., BRADSKI, G., and KOZYRAKIS, C., “Evaluating MapReduce for Multi-core and Multiprocessor Systems,” in *HPCA*, 2007.
- [81] RAO, D. and SCHWAN, K., “vNUMA-mgr: Managing VM Memory on NUMA Platforms,” in *HiPC*, 2010.

- [82] RAO, J., WANG, K., ZHOU, X., and XU, C.-Z., “Optimizing Virtual Machine Scheduling in NUMA Multicore Systems,” in *HPCA*, pp. 306–317, 2013.
- [83] ROSU, M.-C., SCHWAN, K., and FUJIMOTO, R., “Supporting Parallel Applications on Clusters of Workstations: The Virtual Communication Machine-based Architecture,” *Cluster Computing*, 1998.
- [84] SANCHEZ, D. and KOZYRAKIS, C., “Vantage: Scalable and Efficient Fine-Grain Cache Partitioning,” in *ISCA*, 2011.
- [85] <https://asc.llnl.gov/sequoia/benchmarks/#phloem>, 2008. ASC Sequoia Benchmark Codes.
- [86] SONG, N.-O., KWAK, B.-J., SONG, J., and MILLER, M., “Enhancement of IEEE 802.11 Distributed Coordination Function With Exponential Increase Exponential Decrease Backoff Algorithm,” in *Vehicular Technology Conference, 2003. VTC 2003-Spring. The 57th IEEE Semiannual*, IEEE, 2003.
- [87] STEWART, C., KELLY, T., and OTHERS, “A Dollar From 15 cents: Cross-Platform Management for Internet Services,” in *ATC’08: USENIX Annual Technical Conference*, 2008.
- [88] STUECHELI, J. and JOHN, L. K., “Cache Capacity and Memory Bandwidth Scaling Limits of Highly Threaded Processors,” in *ISPASS*, 2009.
- [89] SULLIVAN, D. G. and SELTZER, M. I., “Isolation with Flexibility: A Resource Management Framework for Central Servers,” in *USENIX Annual Technical Conference*, 2000.
- [90] TANG, L., MARS, J., VACHHARAJANI, N., HUNDT, R., and SOFFA, M. L., “The Impact of Memory Subsystem Resource Sharing on Datacenter Applications,” in *ISCA*, 2011.
- [91] TEMBEY, P., GAVRILOVSKA, A., and SCHWAN, K., “A Case for Coordinated Resource Management in Heterogeneous Multicore Platforms,” in *Workshop on the Interaction between Operating Systems and Architecture, WIOSCA*, 2010.
- [92] UHLIG, V. http://l4ka.org/publications/2005/uhlig_phd-thesis_scalability.pdf, 2005. Scalability of Micro-kernels.
- [93] URGAONKAR, B., SHENOY, P., and ROSCOE, T., “Resource Overbooking and Application Profiling in Shared Hosting Platforms,” in *OSDI*, 2002.
- [94] URGAONKAR, B., SHENOY, P. J., CHANDRA, A., and GOYAL, P., “Dynamic Provisioning of Multi-tier Internet Applications,” in *ICAC*, IEEE Computer Society, 2005.
- [95] VAQUERO, L. M., RODERO-MERINO, L., and BUYYA, R., “Dynamically Scaling Applications in the Cloud,” *SIGCOMM Comput. Commun. Rev.*, 2011.

- [96] <http://tinyurl.com/lehv3dl>, 2009. Best Practices for Para-Virtualization Enhancements using EPT and Vt-D.
- [97] VERGHESE, B., GUPTA, A., and ROSENBLUM, M., “Performance Isolation: Sharing and Isolation in Shared-Memory Multiprocessors,” in *Asplos*, 1998.
- [98] VISHAKHA, G., ROB, K., and KARSTEN, S., “Attaining System Performance Points: Revisiting the End-to-End Argument in System Design for Heterogeneous Many-core Systems,” in *SigOps OSR*, 2011.
- [99] <http://bit.ly/efd07H>, 2007. Vsphere: VMWare Cloud OS.
- [100] <http://tinyurl.com/6oln9bw>, 2006. Intel Hardware Assisted Virtualization Support.
- [101] WALDSPURGER, C., “Memory Resource Management in the VMWare ESX Server,” in *OSDI*, 2002.
- [102] WALDSPURGER, C. A. and WEIHL, W. E., “Lottery Scheduling: Flexible Proportional-Share Resource Management,” in *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, USENIX Association, 1994.
- [103] WANG, A., VENKATARAMAN, S., ALSPAUGH, S., STOICA, I., and KATZ, R., “Sweet Storage SLOs with Frosting,” in *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Computing*, HotCloud’12, USENIX Association, 2012.
- [104] WENTZLAFF, D. and AGARWAL, A., “Factored Operating Systems (fos): The Case for a Scalable Operating System for Multicores,” *SIGOPS Oper. Syst. Rev.*, 2009.
- [105] WICKIZER, S. B., CHEN, H., and OTHERS, “Corey: An Operating System for Many Cores,” in *OSDI*, 2008.
- [106] WIKIPEDIA. http://en.wikipedia.org/wiki/Exponential_backoff. ”Exponential back off”.
- [107] XU, H., FENG, C., and LI, B., “Temperature Aware Workload Management in Geodistributed Datacenters,” in *Proceedings of the ACM SIGMETRICS/international conference on Measurement and modeling of computer systems*, ACM, 2013.
- [108] YANG, H., BRESLOW, A., MARS, J., and TANG, L., “Bubble-Flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers,” in *ISCA*, 2013.
- [109] ZHAI, B., BLAAUW, D., SYLVESTER, D., and FLAUTNER, K., “Theoretical and Practical Limits of Dynamic Voltage Scaling,” in *DAC*, 2004.
- [110] ZHU, X., YOUNG, D., and OTHERS, “1000 islands: Integrated capacity and workload management for the next generation data center,” in *ICAC*, 2008.

- [111] ZHURAVLEV, S., BLAGODUROV, S., and FEDOROVA, A., “Addressing Shared Resource Contention in Multicore Processors via Scheduling,” in *Asplos*, 2010.

VITA

Priyanka Tembey was born in Pune, India on December 14, 1984. She completed her Bachelors' in Computer Science from Cummins College of Engineering affiliated with University of Pune, India in 2006. She then traveled to Atlanta, USA to complete her Masters' in Computer Science at the College of Computing, Georgia Institute of Technology. After completing her Masters' she continued onwards toward her Ph.D. in Computer Science at Georgia Tech.

Priyanka completed her doctoral dissertation titled "Virtual Platforms: Improving Performance and Isolation Properties of Shared Multicore Servers" in October 2013 working with Dr. Karsten Schwan and Dr. Ada Gavrilovska as her advisors. Her primary research interests are scalable system designs, heterogeneous multicore and many-core platform resource management, performance isolation and system virtualization.