Technical Report
Number: NU-EECS-13-07

July 23, 2013

Dynamic Adaptive Resource Management in a Virtualized NUMA Multicore System for Optimizing Power, Energy, and Performance

Chang Seok Bae

Abstract

A Non-Uniform Memory-Access (NUMA) machine is currently the most deployed type of a hardware architecture for high performance computing. System virtualization on the other hand is increasingly adopted for various reasons. In a virtualized NUMA system, the NUMA attributes are transparent to guest OS's. Thus, a Virtual Machine Monitor (VMM) is required to have NUMA-aware resource management. Tradeoffs between performance, power, and energy are observable as virtual cores (vcores) and/or virtual addresses are mapped in different ways. For example, sparsely located vcores have an advantage in memory caching compared to densely located vcores. On the other hand, densely located vcores tend to save power. Such tradeoffs lead to an abstract question: how a VMM as a resource manager can optimally or near-optimally execute guests under a NUMA architecture. In my dissertation, I claim that it is possible to solve this problem in real time through a dynamic adaptive system. Workload-aware scheduling, mapping, and shared resource management are controlled by adaptive schemes. The user may demand one of three objectives: performance, energy, or power. My system also incorporates a new detection framework that observes shared memory access behaviors with minimal overheads, and includes models that estimate performance, power, and workloads' resource demands. The system uses a simple heuristic policy if it is sufficient to optimize the user demand objective.

My dynamic adaptive system, called NUMA-ware Virtualized Adaptive Runtime (NAVAR), was designed, implemented, and evaluated with benchmarks from PARSEC, SPEC OMP, and NAS suites. It can achieve more than a 10% performance and energy benefit over the default static policy. And, its performance is never worse than the default. When NAVAR includes developed offline models, these models are validated with the separate validation set from the training set. NAVAR is also deployed and tested in two test machines, and its overheads suggest that it will be highly extensible for scaled NUMA multicore machines.

This effort was made possible by support from the National Science Foundation (NSF) via grants CNS-0709168, and the Department of Energy (DOE) via grant DE-SC0005343.

Keywords: Virtualization, NUMA, Adaptation, Energy

NORTHWESTERN UNIVERSITY

Dynamic Adaptive Resource Management in a Virtualized NUMA Multicore System for Optimizing Power, Energy, and Performance

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Computer Science

By

Chang Seok Bae

EVANSTON, ILLINOIS

August 2013

Abstract

Dynamic Adaptive Resource Management

in a Virtualized NUMA Multicore System

for Optimizing Power, Energy, and Performance

Chang Seok Bae

A Non-Uniform Memory-Access (NUMA) machine is currently the most deployed type of a hardware architecture for high performance computing. System virtualization on the other hand is increasingly adopted for various reasons. In a virtualized NUMA system, the NUMA attributes are transparent to guest OS's. Thus, a Virtual Machine Monitor (VMM) is required to have NUMA-aware resource management. Tradeoffs between performance, power, and energy are observable as virtual cores (vcores) and/or virtual addresses are mapped in different ways. For example, sparsely located vcores have an advantage in memory caching compared to densely located vcores. On the other hand, densely located vcores tend to save power. Such tradeoffs lead to an abstract question: how a VMM as a resource manager can optimally or near-optimally execute guests under a NUMA architecture. In my dissertation, I claim that it is possible to solve this problem in real time through a dynamic adaptive system. Workload-aware scheduling, mapping, and shared resource management are controlled by adaptive schemes. The user may demand one of three objectives: performance, energy, or power. My system also incorporates a new

detection framework that observes shared memory access behaviors with minimum overheads, and includes models that estimate performance, power, and workloads' resource demands. The system uses a simple heuristic policy if it is sufficient to optimize the user demand objective.

My dynamic adaptive system, called NUMA-aware Virtualized Adaptive Runtime (NAVAR), was designed, implemented, and evaluated with benchmarks from PARSEC, SPEC OMP, and NAS suites. It can achieve more than a 10% performance and energy benefit over the default static policy. And, its performance is never worse than the default. When NAVAR includes developed offline models, these models are validated with the separate validation set from the training set. NAVAR is also deployed and tested in two test machines, and its overheads suggest that it will be highly extensible for scaled NUMA multicore machines.

Thesis Committee

Peter A. Dinda, Northwestern University, Committee Chair

Russ Joseph, Northwestern University, Committee Member

Nikos Hardavellas, Northwestern University, Committee Member

Dongyan Xu, Purdue University, Committee Member

Acknowledgements

I am very grateful for those who helped me to complete my dissertation.

I indeed appreciate my advisor, Prof. Peter A. Dinda. I cannot express in a few words for his enormous helps and guidance throughout my PhD study. He patiently encouraged and motivated me especially when I faced obstacles. I am also amazed by his enthusiasm and openness. It is my honor to finish my PhD under his guidance.

I also want to say many thanks to my thesis committee members and many faculty members. Prof. Russ Joseph and Prof. Nikos Hardavellas brought me broader aspects and knowledge of architecture and system. I appreciate Prof. Dongyan Xu for giving me priceless comments as an external committee member from my proposal. I had two chances to help Prof. Lawrence Henschen as a teaching assistant which also helped me to leverage my communication skill and I respect his ceaseless efforts on teaching.

Many colleagues in Evanston campus and also people I met through the V3VEE project gave invaluable feedbacks on my research. I was fortunate to have Lei Xia as my office-mate who is a good friend of mine. We had many discussions on various topics. Jack Lange is one of the smartest people I ever met and his broader and in-depth knowledge in systems helped my research in various ways. Kyle Hale, Maciej Swiech, and Madhav Suresh are good collaborators. Patrick Bridges, and Kevin Pedretti in the V3VEE project gave helpful feedbacks during project meetings.

Besides the Northwestern community, many people I met at Intel during my three internships gave incomparable advice and ideas. Especially, I must say my mentor Tayeb Jamel at Intel whom I worked with for almost an year. He is smart, professional, and even humorous. I miss many conversations we had at the lab. Stevan Rogers, Jisoo Yang, Kyungtae Han, and David Ott gave me various insights that result in my dissertation.

Most of all, I want to give my special thanks to my family who endlessly support me and pray for me.

Contents

List of Figures

Chapter 1

Introduction

System virtualization is popular in many different hardware contexts. Demands on HPC and cloud services in these days accelerates the deployment of system virtualization, because the virtualization technique provides great flexibility in system resource management concerns. On the other hand, a modern node or server features multiple sockets in one physical machine (a NUMA[1] multicore system), and each socket (a NUMA domain) has multiple cores and a separate memory space with its own memory controller. As more hardware threads are made available by the NUMA multicore machine, more parallel and concurrent workloads are available to be executed. Given that, the multiple choices of which cores and memory banks are mapped to an application (a guest) invoke various resource management optimization problems with regards to virtual core (vcore) and page mapping, and vcore scheduling.

I claim that these salient resource management optimization problems can be solved by an interference-based (black box based) dynamically adaptive system. To solve a resource management problem, in general it is important to know the resource demand and

[1]Non-Uniform Memory-Access

provisioning (supply), and typically it is hard to extract the resource demand information in advance. The white box approach has advantages in that it can capture the applications' resource demands more directly. However, the white box approach has scalability issue, because it tends to be customized for each application. When system optimizations lie in a coarse-grained level, the black box approach is particularly viable. If the system can be optimized by a black box approach or a simple heuristic, the scope of coverage can be smoothly extended, which is particularly useful for managing large scale nodes for high performance computing and backend servers. The system further needs to be dynamically optimized during runtime. Along with searching the best configuration, the overhead for system reconfigurations themselves should be managed.

1.1 Virtualized NUMA Multicore Machine

The virtualized NUMA multicore machine has two attributes: non-uniform memory-access with multicore, and resource virtualization. For resource virtualization and for system optimization, the virtual machine monitor (VMM) is central for driving inference-based resource management due to observability and controllability available from the VMM context.

The NUMA multicore machine exhibits differences in the memory hierarchy despite the uniform memory access. The physical memory in the system is divided into partitions, each of which has a different performance profile depending on the accessing core. Depending on the location of a core and the accessed page, memory-access distance varies, which produces tradeoffs in the resource mapping.

When the NUMA multicore machine is virtualized, more software layers are introduced. The virtualized system has a VMM on top of host OS. While pursuing the black box approach to address the tradeoffs in the resource mapping, the inference work can be made in one of the software layers: in the VMM, in the host OS, and even in the guest OS. When the observation is made in the guest OS context, the resource provisioning information of the physical machine should be exposed to the guest OS. Even with the physical hardware information, the guest OS still needs to make inferences for the application. The guest OS may be better able to extract application's information; however, the VMM can also obtain the context information of the guest OS. Also, the guest OS approach hurts the scalability and flexibility that the system virtualization can provide, because the optimization is very dependent on the guest OS version.

Instead of exposing the NUMA attributes to the guest, the VMM can create a virtual machine (VM) with all flat (SMP) topology of virtual cores (vcores). The fact that the VMM has the guest system information makes itself to be different from the host OS. Technically, the VMM has the same privilege as the host OS; therefore, the inference work in the VMM does not affect the system scalability. Furthermore, the VM context information is important when multiple VMs run with multiple potentially overlapping vcores. The commodity host OS is designed to schedule processes with a fairness objective. Sometimes, the vcores (kernel processes) of one VM needs to be executed at the same time; otherwise, the waiting time for synchronization would be unnecessarily increased. This issue happens for any virtualized multicore systems.

1.2 Energy and Power Optimization

In the abstract, providing more resources is expected to boost performance. More resources here refers to more cache space and memory bandwidth, for example, to decrease memory-access time. Since NUMA multicore physical machine become available, significant efforts have been made and are still under investigation for minimizing overall memory-access time. However, the **over-provisioning** cases have been more often disregarded. The degree of resource demand is a characteristic of the application; thus, depending on the workload, more resources do not necessarily help to make it faster. The **withdrawal** of the over-provisioned resource helps to reduce energy, which is operational cost. Therefore, one way to optimize energy is to power-off or power-down the hardware module that is not currently used or does not help the execution speed. On the other hand, power optimization is different due to its orthogonality the execution time. One extreme case for power optimization is consolidating all vcores in one physical core, that minimizes power, but at potentially high energy and low performance. The problem for power optimization involves performance constraints.

System virtualization allows us to migrate VMs to a different physical machine seamlessly during runtime. This feature gives an opportunity to consolidate multiple VMs to power-off an entire machine. This coarse-grained VM mapping strategy works if the resource demand is predictable; otherwise, the VM migration cost is not commensurable with the energy saving. In particular, when resource demand is fluctuating and the VM is not yet consolidated, energy and power optimization in a single node (one physical machine) has considerable opportunity.

1.3 Thesis Statement

In a virtualized NUMA system, there are tradeoffs between performance, power, and energy given where vcore and/or virtual addresses are placed and when vcores are scheduled. Sparsely placed vcores take better advantage of cache and memory channels compared to densely placed ones which conversely save power with resource contention. Furthermore, specific memory-access patterns generated by some workloads result in opposite tradeoffs. These rules of thumb are aspects of a broader problem: how a VMM can optimally or near-optimally execute guests under a NUMA architecture. I claim that it is possible to solve this problem in real time through a dynamic adaptive system. I have developed a system, NAVAR (NUMA-aware Virtualized Adaptive Runtime), that performs workload-aware scheduling, mapping, and shared resource management in pursuit of the objectives that the user demands. The basis of NAVAR is a framework that detects shared memory-access patterns with minimal overheads and minimal machine dependencies, and then predicts resource demands and estimates contention due to sharing.

1.4 Related Works

The matter of thread mapping and scheduling is a hot topic and a very traditional problem. In virtualized systems, the VMM needs to be extended with adaptive resource provisioning, since resource attributes are transparent to the guest OS and parallel workloads are becoming mainstream. Virtual cores (vcores) tend to be numerous when multiple parallel applications are executing at the same time. In this context, co-locating and co-scheduling threads have been addressed recently in the context of space sharing for native systems [90, 22, 49]. Moreover, for concurrently executing workloads, synchronization is also important. Co-scheduling has been successfully demonstrated to this end [92, 105, 50]. Some recent papers [116, 17, 115, 107, 72] argue for selective co-scheduling policies for virtualized systems to mitigate the side effects of co-scheduling. However, there are still research opportunities in these areas. For space-sharing concerns, mapping vcores has not been addressed thoroughly with the constraints of virtualization, NUMA characteristics, and power-proportionality. Most of previous studies are conducted in native systems and executed without energy-awareness. Furthermore, tradeoffs indeed exist as a matter of performance, energy, and power. Energy proportionality becomes one of the crucial constraints in system design as well. Meisner et al. [87] investigate opportunities in energy savings across different hardware components. They conclude memory systems need further improvement. Recently significant efforts were made in the hardware [41, 42, 34, 63, 12]. Although these new hardware designs are not yet deployed, the benefit of these new features can be investigated. The resource management problem for the NUMA multicore machine is still a hot issue [28, 40]. Mostly, performance optimizations are discussed in a native environment; therefore, power-proportionality and optimization in a virtualized system are still open research problem in the context of NUMA.

1.5 Outline

In Chapter 2, three major problems are formulated individually. The vcore mapping problem (**`vcore-mapping`**) addresses tradeoffs in cache contention and energy consumption. Mapping vcores across different sockets takes more space in caches with increased power as sockets are active. Accordingly, the location of pages in multiple NUMA domains is related to the cores that execute application threads, which affects memory-access traffic (**`page-mapping`**). The more pages that are spread across multiple NUMA domains, the more bandwidth is utilized. Scheduling vcores in a hardware thread is another problem (**`vcore-scheduling`**). This issue appears when multi-threaded workloads are run concurrently. More delays on lock waiting can occur in a virtualized system than in a native system if the system scheduler is not aware of the runtime characteristics of the multiple vcores.

Regardless of the problem formulation, resource management problems do not occur independently in a real world; therefore, the problem must be addressed in a different way. The **`vcore-scheduling`** problem especially occurs only when multiple vcores are mapped to one hardware thread. In this situation, system resources are fully utilized; thus, there is little chance to power-off a hardware sub-system. Conversely, when the system resources are underutilized due to the small number of VMs that are used, it becomes more important to keep power-proportionality for energy efficiency. And also, in this case, the system tends to have more configuring options. Overall, resource demanding situations are classified into three different cases. The three classes are under-subscription, full-subscription, and over-subscription, which are addressed in separate chapters. First, the overall system design approaches are discussed in the Chapter 3. The chapter also describes the three resource management mechanisms, the common experimental setups, and the classification of the three types of resource demanding cases. Chapter 4 and 5 are about the under-subscription cases. The following Chapter 6 and 7 describe the full-subscription and over-subscription cases separately. Besides the differences in the situation, each chapter discusses different system designs particularly from policy design perspectives. Chapter 8 wraps up the overall different policy designs by discussing the generalization. The combined policy and testing of the designed system in a different machine are also presented in the chapter. It also includes discussions for the scalability of the policy. Chapter 9 discusses relevant articles, and Chapter 10 summarizes the whole chapters, lists major contributes, and presents possible future works.

Chapter 2

Problem Statement and Possible Approaches

This chapter introduces key resource management problems and their formulations. It also includes brief discussions about possible approaches to solve them. The purpose of this chapter is to comprehensively illustrate the problems. The problem statement is on the three main problems: **`vcore-mapping`**, **`page-mapping`**, and **`vcore-scheduling`**. They are described here as strongly decoupled with each other so that each problem is clearly identified. The problems are a type of an optimization problem, which is comprised of objective functions, constraints, and metrics. The objectives are performance, power and energy in virtualized systems. One of them can be chosen. Once the commonly used notations and variables are defined, the three problems are formulated succinctly.

After formulating problems and addressing possible approaches to solve them, the situations that raise the resource optimization problem are classified. The classification may help to solve the problems efficiently.

2.1 Notations and Variables

These notions and variables are commonly used in the key problem formulations.

- $vcore \in V = \{0, ..., v\}$: Number of distinct vcores across VMs in a machine. V is a set including all vcores. Note that it uniformly counts vcores across different VMs on a physical machine.
- $vm \in VM = \{0, ..., max\_vm\}$: Number of VM. VM is a set including all VMs in a machine.
- $max\_vcore(vm)$: Maximum number of vcores in a VM vm.
- $vcore_{vm} \in V_{vm} = \{0, ..., max\_vcore(vm) - 1\}$: Number of vcores in the VM vm. V_{vm} is a set including all vcores of the VM vm. $|V_{vm}|$ is equal to $max\_vcore(vm)$.
- $(vcore_{vm}, vm) \in VVM = \{(0, 0), ..., (max\_vcore(0) - 1, 0), ..., (0, max\_vm), ..., (max\_vcore(max\_vm) - 1, max\_vm)\}$: The distinct vcore $vcore_{vm}$ of the VM vm.
- $lcore \in L = \{0, ..., l\}$: Number of logical cores[1] in each physical core. L is a set including all logical cores in a physical core.
- $pcore \in P = \{0, ..., p\}$: Number of physical cores in each socket. P is a set including all physical cores in a socket.
- $socket \in S = \{0, ..., s\}$: Number of sockets[2] in a physical machine. S is a set including all sockets in a physical machine.

[1]A distinct logical core represents one hardware thread.
[2]A socket and a NUMA domain are used interchangeably.

- $(lcore, pcore, socket) \in C = \{(0,0,0), ..., (l,p,s)\}$: A distinct hardware thread as the logical core $lcore$ of the physical core $pcore$ of the socket $socket$. C is a set including all logical cores in a machine.

- $(pcore, socket) \in C' = \{(0,0), ..., (p,s)\}$: A physical core, $pcore$ and its the socket $socket$. C' in a physical machine.

- $(lcore, pcore) \in C'' = \{(0,0), ..., (l,p)\}$: A distinct hardware thread, $lcore$ and its phyiscal core $pcore$, in a certain socket. C'' is a set including all hardware threads in a socket. Note that a Symmetric Multi-Processor (SMP) model is assumed, thus every socket should contain the same core topology and the same number of logical and physical cores.

- $pfn \in P = \{0, ..., p\}$: A distinct page frame number in the guest physical address space, representing a page that should be mapped to one physical page frame number. P is a set including all page frame numbers. Note that it uniformly counts across different VMs.

- $Exectime_{vcore,(lcore,pcore,socket)}$: Execution time is the time taken by executing a task tied to the vcore $vcore$ mapped to the hardware thread, $(lcore, pcore, socket)$

- $Makespan_{vm}$: Makespan, by definition, is the time difference between the start of the first task and the end of the last, particularly in a multi-task execution; therefore, each VM vm has its own makespan $Makespan_{vm}$.

- $sMakespan_{vm}$: $Makespan$ of vm when there is only one VM vm runs with a balanced mapping, which refers to the mapping that vcore siblings (vcores from the same VM) are mapped in different hardware threads.

- $cMakespan_{vm}$: $Makespan$ of vm when there are other max_vm VMs running, holding the balanced mapping. Note that $cMakespan_{vm}$ and $sMakespan_{vm}$ should be the same when $|VM| = 1$; however, $Makespan_{vm}$ and $sMakespan_{vm}$ may be different, because $Makespan_{vm}$ is not declared with any restrictions for the vcore mapping.

- $PageAccessDelays_{pfn}$: DRAM access delays for the page frame number pfn.

- $x_{vcore \rightarrow (lcore,pcore,socket)} \in \{0, 1\}$, $x_{(vcore_{vm},vm) \rightarrow (lcore,pcore,socket)} \in \{0, 1\}$: A vcore mapping status to a hardware thread. The mapping status is binary, either mapped (1) or not (0).

- $x_{vcore \rightarrow socket} \in \{0, 1\}$: A vcore mapping status to a socket. The mapping status is binary, either mapped (1) or not (0).

- $w_{pfn \rightarrow vcore} \in \{0, 1\}$: A page mapping status to a vcore. The mapping status is binary, either mapped (1) or not (0).

- $w_{pfn \rightarrow socket} \in \{0, 1\}$: A page mapping status to a socket. The mapping status is binary, either mapped (1) or not (0).

- $y_{(pcore,socket)}$: Power state of the physical core $pcore$ of the socket $socket$.

- y'_{socket}: Power state of the DRAM controller of the socket $socket$.

- $p_{vcore,(lcore,pcore,socket)}$: Power of the distinct hardware thread, $(lcore, pcore, socket)$, when the vcore $vcore$ is mapped.

- p_{cpu_max}: CPU power cap that a user provides or is unbounded.

- $p_{dram\_active}(PageAccessDelays_{pfn})$: Active power spent on accessing DRAM. It is linear to the input, $PageAccessDelays_{pfn}$.

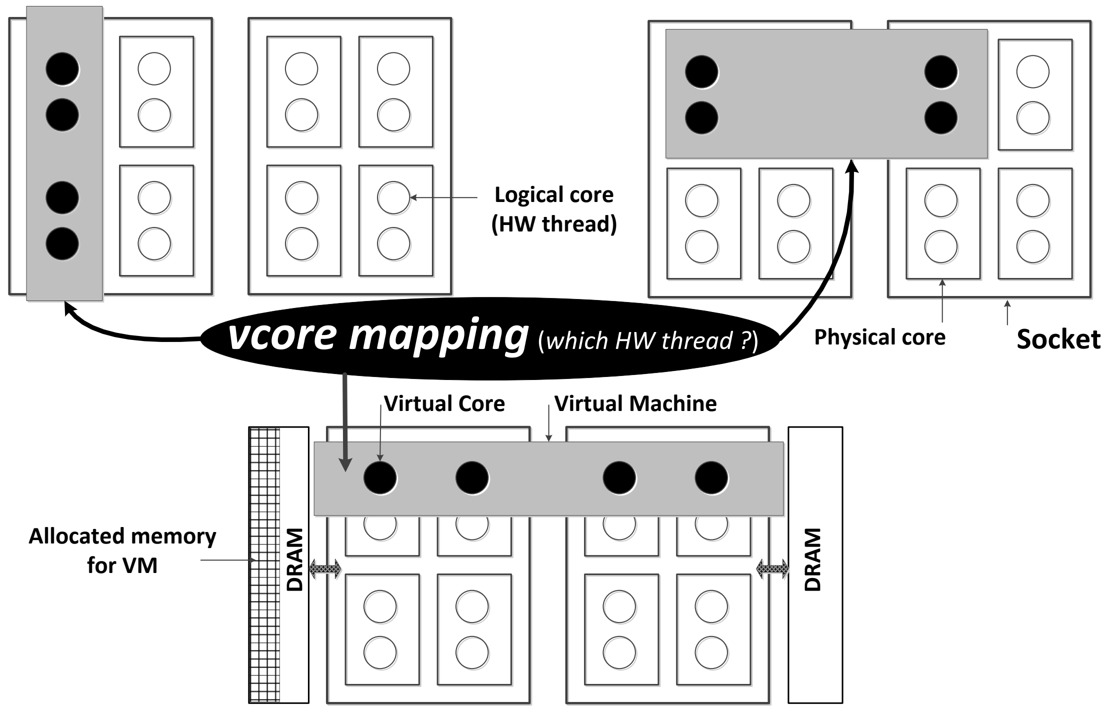


Figure 2.1: Illustration of **`vcore-mapping`** problem.

- p_{dram_static}: Static power in DRAMs.
- p_{dram_max}: DRAM power cap that a user provides or is unbounded.

2.2 Virtual Core Mapping (`vcore-mapping`)

The problem is about a selection of one configuration of the vcore mapping (Figure 2.1) with constraints for all objectives: performance, power, and energy. Shmoys et al. [102] and Khuller et al. [71] formulate a similar problem of thread-to-core mapping with power constraints. Similarly, the objectives and constraints that are related to the **`vcore-mapping`** can be formulated. The problem formulation incorporates abstract metrics such as an execution time and CPU power. At first, two assumptions should be noted. They decouple **`vcore-mapping`** from **`page-mapping`** and **`vcore-scheduling`**.

- **under-subscription[3] and balanced scheduling**: In the case of over-subscription[4], multiple vcores may be mapped to the same hardware thread, which potentially invokes the scheduling problem depicted in Figure 2.3. Even with the under-subscription assumption, a VMM or a host OS possibly schedules multiple vcores, even from the same VM, on the same hardware thread; therefore, the constraint of balanced scheduling is necessary to completely decouple this problem from the scheduling concerns. Balanced scheduling refers to the configuration where vcore siblings are located in different hardware threads [107].

- **Pages are allocated in one specific NUMA domain**: This constraint decouples the problem from page mapping choices.

Objective One of three objectives can be chosen:

1. Performance objective:

$$Min \sum_{vm \in VM} Makespan_{vm}$$

2. Power objective:

$$Min \sum_{(lcore,pcore,socket) \in C} \sum_{vcore \in V} \left(p_{vcore,(lcore,pcore,socket)} \cdot x_{vcore \rightarrow (lcore,pcore,socket)}\right)$$

[3]The number of all vcores is less than the number of all available hardware threads

[4]The number of all vcores is more than the number of all available hardware threads

3. Energy objective:

$$Min \sum_{(lcore,pcore,socket)\in C} \sum_{vcore\in V} (Exectime_{vcore,(lcore,pcore,socket)} \cdot p_{vcore,(lcore,pcore,socket)}$$
$$\cdot x_{vcore\rightarrow(lcore,pcore,socket)})$$

Subject to

$\forall(lcore, pcore, socket) \in C,\ \sum_{vcore\in V} x_{vcore\rightarrow(lcore,pcore,socket)} \leq 1$: Each hardware thread should serves at most one vcore for the balanced scheduling assumption. This constraint conveys the under-subscription assumption as well.

$\forall vcore \in V,\ \sum_{(lcore,pcore,socket)\in C} x_{vcore\rightarrow(lcore,pcore,socket)} = 1$: Each vcore should be mapped to only one hardware thread.

$\sum_{(pcore,socket)\in C'} (\sum_{lcore\in L} \sum_{vcore\in V} x_{vcore\rightarrow(lcore,pcore,socket)} \cdot p_{(lcore,pcore,socket)}) \cdot y_{(pcore,socket)}$ $< p_{cpu\_max}$: If overall CPU power cap is given, total power should be limited to it.

$\forall vcore \in V,\ \forall(lcore, pcore, socket) \in C,\ x_{vcore\rightarrow(lcore,pcore,socket)} \leq y_{(pcore,socket)}$: When a physical core is powered-down, the physical core cannot be assigned to any vcore.

$\forall pfn \in P,\ \sum_{socket\in S} w_{pfn\rightarrow socket} = 1$: Each page should be assigned at most one socket.

$\exists socket \in S,\ \forall pfn \in P,\ w_{pfn\rightarrow socket} = 1$: Each page should be assigned to memory space in one specific socket (in this case, socket $socket$).

$x_{vcore\rightarrow(lcore,pcore,socket)} \in \{0, 1\},\ y_{(pcore,socket)} \in \{0, 1\}$: Mapping status is binary, either mapped (1) or not (0). Also, power state is either active (1) or idle (0) to fit the problem into an integer linear programming problem by relaxing constraints.

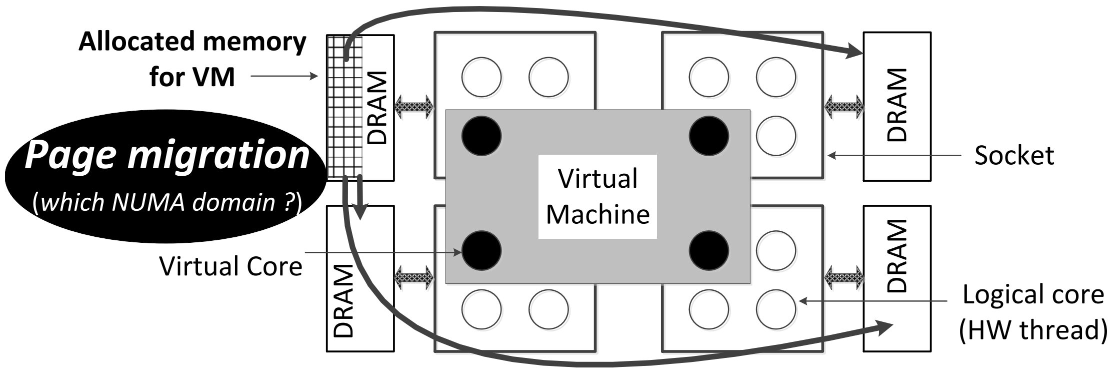


Figure 2.2: Depiction of **`page-mapping`** problem.

2.3 Page Mapping (`page-mapping`)

This problem addresses the trade-offs between the locations of pages in NUMA domains. The problem is illustrated in Figure 2.2 and is another configuration concern. The optimal configuration may vary according to the selected objective. The objectives and constraints are formluated with abstract metrics such as DRAM power and page access delays. Before the formulation, three assumptions should be noted. The first two assumptions decouple the problem from other problems, **`vcore-mapping`** and **`vcore-scheduling`**. The third assumption specifies the configurability on memory allocations in a hardware-level. The last assumption explains the reason why power and energy objectives are formulated in the same manner.

- **Virtual core mapping to the physical core is fixed**: This assumption excludes variations from **`vcore-mapping`**; thus, DRAM access time ($PageAccessDelay$) indeed determines the performance, and the degree of power is coupled with the latency of the DRAM access pathway.

- **Balanced mapping for all vcores with under-subscription conditions** : Similar to `vcore-mapping`, scheduling effects are excluded from this problem formulation.

- **Page allocation is interleaved across channels and DIMMs within a socket**: The following formation ignores the option for allocating a page into a specific region in a NUMA domain; therefore, the tradeoffs between interleaved and non-interleaved allocations in a socket are not considered. Nonetheless, these trade-offs are still important and promising options for the better energy proportionality.

Objective

1. Performance objective:

$$Min \mid \bigcup_{pfn \in P} PageAccessDelays_{pfn} \mid$$

2. Power objective:

$$Min(\sum_{socket} y'_{socket \in S} \cdot p_{dram\_static} + p_{dram\_active})$$

3. Energy objective:

$$Min(\sum_{socket} y'_{socket \in S} \cdot p_{dram\_static} + p_{dram\_active} \cdot (\mid \bigcup_{pfn \in P} PageAccessDelays_{pfn} \mid))$$

Subject to

$\forall(lcore, pcore, socket) \in C$, $\sum_{vcore \in V} x_{vcore \rightarrow (lcore, pcore, socket)} \leq 1$: Each logical core serves at most one vcore, excluding **`vcore-scheduling`**.

$\forall pfn \in P,\ 0 \leq \sum_{vcore \in V} w_{pfn \rightarrow vcore} \leq |V|$: Each page is touched by none or less than the total number of vcores.

$\forall pfn \in P,\ \sum_{socket \in S} w_{pfn \rightarrow socket} = 1$: Each page should be assigned at most one socket.

$\sum_{socket \in S} y_{socket} \cdot p_{dram\_static} + p_{dram\_active}(\sum_{pfn \in P} PageAccessDelays_{pfn}) \leq p_{dram\_max}$: If overall DRAM power cap is given, total power should be limited.

$\forall pfn \in P,\ \forall socket \in S,\ w_{pfn \rightarrow socket} \leq y'_{socket}$: (Re)-allocating the page with the page frame number pfn to a DIMM in a socket $socket$ is not allowable, when the DIMMs in the socket are powered-off.

$w_{pfn \rightarrow vcore} \in \{0, 1\}, w_{pfn \rightarrow socket} \in \{0, 1\},\ y'_{socket} \in \{0, 1\}$: Mapping variables, $w_{pfn \rightarrow vcore}$, $w_{pfn \rightarrow socket}$, have a binary value, either mapped (1) or not (0), and so does power state of memory controller of socket $socket$, y'_{socket}. The power state is either active (1) or idle (0). It is intended to relax the problem fitting into a integer linear programming model.

2.4 Virtual Core Scheduling (`vcore-scheduling`)

While **`vcore-mapping`** and **`page-mapping`** are related to the space-sharing issue, Figure 2.3 describes the scheduling problem in allocating vcores in time slices in the time-sharing. When many VMs share the same hardware thread, these vcores out of different VMs take separate time quantums; thus, the configuration is on sequencing vcores in time slices. Given that vcore and page mappings are fixed, vcore-scheduling does not affect power. Energy consumption is linear to the performance; thus, the performance and energy objectives follow in the same direction. Jiang et al [64] formulate this scheduling problem, and the problem is formulated similarly here. The problem formulation is drawn with four

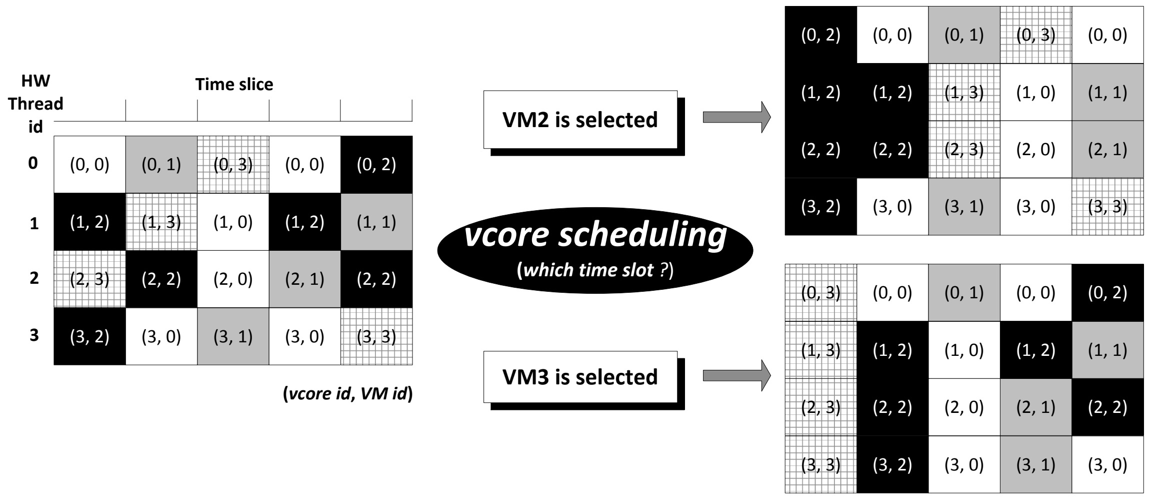


Figure 2.3: **`vcore-scheduling`** configures the execution sequences of multiple VMs' vcores in each hardware thread.

assumptions. While the last two assumptions make **`vcore-scheduling`** decoupled with the other problems, the first two clarify the vcore mapping conditions.

- **Contention in vcore mapping (>1 vcore per HW thread)**: The scheduling issue emerges only if hardware threads are mapped to more than one vcore.
- **Balanced scheduling in each VM**: Virtual cores from the same VM should be spread out across different hardware threads. Without this restriction, vcores with the same VM may be scheduled on the same hardware thread. The concept of balanced scheduling is discussed in [107].
- **Fixed vcore mapping**: Similar to the assumptions **`page-mapping`** has, this condition decouples **`vcore-scheduling`** from **`vcore-mapping`**. Although scheduling in general covers thread migrations, controlling vcore sequence in a hardware thread is the focus here.

- **Pages are mapped in one specific socket**: Similarly, page mapping issues are excluded to make the scope of the problem clear.

Objective The objective is reducing the concurrency overhead.

$$Min \sum_{n} \frac{cMakespan_{VM_n} - sMakespan_{VM_n}}{sMakespan_{VM_n}}$$

Subject to

$\exists (lcore, pcore, socket) \in C,\ \sum_{vcore \in V} x_{vcore \to (lcore,pcore,socket)} > 1$: There is at least one logical core mapped with two more vcores to comply the contention assumption.

$\forall vm \in VM,\ \forall (lcore, pcore, socket) \in C,\ \sum_{vcore_{vm} \in V_{vm}} x_{(vcore_{vm},vm) \to (lcore,pcore,socket)} \leq 1$: Each logical core should be mapped with at most a vcore from one specific VM. (balanced scheduling within a VM)

$\forall vcore \in V,\ \sum_{(lcore,pcore,socket) \in C} x_{vcore \to (lcore,pcore,socket)} = 1$: No more than one logical core should be mapped for one vcore.

$|V| = |VVM|$: Virtual cores are exclusive to the different VMs.

$\forall pfn \in P,\ \sum_{socket \in S} w_{pfn \to socket} = 1$: Each page should be assigned at most one socket.

$\exists socket \in S,\ \forall pfn \in P,\ w_{pfn \to socket} = 1$: Each page should be assigned to memory space in one specific socket (in this case, socket $socket$).

$x_{(vcore_{vm},vm) \to (lcore,pcore,socket)} \in \{0, 1\},\ x_{vcore \to (lcore,pcore,socket)} \in \{0, 1\}$: Mapping is binary value.

2.5 Approach for the Solutions

In this section, possible options will be discussed for solving the three problems. Two possible approaches can be considered: Integer Linear Programming (ILP) and heuristic approaches are those options.

The procedure for ILP-based approach starts from the problem formulation in Section 2.2, 2.3, and 2.4. The problems are described in a canonical form that matches very well with ILP. By incorporating models, the problem formulation is describable with measurable metrics; then, the problem formula can be solved at runtime using an ILP solver. One upside of the ILP-based approach is that solutions are very deterministic and proven to be optimal as long as formulations are complete, whereas the heuristic approach does not guarantee optimality as a complete problem formulation is not taken into account. Nonetheless, there are still tradeoffs between them.

2.5.1 Heuristic Approach

The heuristic approach turns out to be considered as an alternative to the ILP-based approach. The heuristic approach has flexibility in tackling specific problems in terms of system design. It does not require a complete formulation for a ILP solver. Instead, the heuristic approach allows us to design a policy component, illustrated in Figure 3.1, with different structures, control algorithms and collected metrics. This section will address some options in the structure at a high-level. All other parts of policy design are rather case-specific. This is a real distinction (the flexibility) compared to the ILP-based approach. The cases are detailed along with experimental tests. These types of approaches and structures are considered:

- Brute-force approach with closed-loop control

- Model-based prediction with open-loop control
- Model-based prediction with semi-closed-loop control

Structure: open-loop vs. closed-loop The terminologies of closed-loop and open-loop are taken from feedback control systems. A feedback control system is a typical structure for a control system. The open-loop and closed-loop structure are two extreme types. The loop is a feedback loop that delivers measured metrics after configuration is complete. The main differences between the two structures are based on measurements that feed back to the front-end decision components. If there is no direct measurability of system outputs, a corresponding model might be responsible for providing estimated values. This is virtual substitution for the feedback loop, which is called a semi-closed-loop.

Execution time, power and energy are the main output measures. Power and energy objectives are clearly easy to measure, however, it is not clear how to capture speed or performance in the middle of execution when the objective is makespan. Makespan is determined after the workload is completed. CPI, for example, provided by hardware monitor is reported to have significant mismatches with the execution time in concurrent and parallel workloads [48, 8]. A performance model may need to drive a semi-feedback loop in terms of that. Nevertheless, it is not always the case that no metric is directly correlated to execution time here. Overall, the point is that there are two structures in this control system, because not all controlled quantities are directly measurable.

2.6 Classes of Resource Demands

So far, the three problems are formulated individually. In the real world, however, multiple or all of problems become an issue. They always depend on situations, so the context in which problems occur needs to be profiled. One way to do this is identifying the type of situation or class.

Three classes There are several extreme cases between vcore mapping and scheduling mechanism. Only when multiple VMs share one hardware thread and that is the case for all hardware threads, the scheduling mechanism is available (over-subscription). Other extreme cases are when a single VM has the number of vcores less than the number of hardware threads (under-subscription). In such cases, there is no need to consider scheduling concerns between VMs, but coarse-grained vcore remapping is important to address differences in cache contention between mapping strategies.

These examples show the case-sensitivities for the problems. This type of situation is actually coupled with resource demands by workloads. Therefore, we believe that that three classes should be defined: under-subscription, full-subscription, and over-subscription. One assumption here is that total guest memory of all VMs is less than host physical memory, so all concerns related to swapping are excluded. This is also aligned with in-memory system design, which has become popular.

In defining our classes, two methods are possible: a naive version and a strict version of definition. There is a rule-of-thumb that when a multiple socket machine is given, the system manager tends to partition the machine per socket. This is particularly reasonable for a NUMA machine, as each memory controller resides in each socket, which implicitly separates resources per NUMA domain. Loosely, when a two-socket NUMA machine is given, under-subscription happens only when a single VM is running. However, when two VMs are running, full-subscription occurs. When more than two VMs run, it is over-subscription. This model defines classes based on the number of VMs over the number of NUMA domains. The reason why this is naive is that it does not strictly take into account the number of vcores. One extreme case is when each VM has only one vcore. Supposing each socket has four hardware threads and two VMs – each of which has a vcore, it is not reasonable to classify this as full-subscription. This is under-subscription as only a quarter of hardware threads are active. Another way to define load of resource demands is through comparing the number of vcores and the number of hardware threads. Particularly, over-subscription is the case when overall number of vcores are more than the number of hardware threads that the machine provides. When the number of vcores are less than overall hardware threads divided by the number of NUMA domains, under-subscription is the case. Any other situation than this would be full-subscription. This is not yet a complete classification since there are potential corner cases. The purpose of this is clarify what problems belong to what class – that is what type of situation. Accordingly all the rest of the work tries to tackle the problems based on classes, as defined formally below:

- under-subscription:

$$\frac{number\_of\_overall\_hardware\_threads}{number\_of\_NUMA\_domains} \geq \textit{number of overall vcores}$$

- over-subscription:

$$\textit{number of overall hardware threads} \leq \textit{number of overall vcore}$$

- full-subscription: all cases other than above

Notice that an assumption for vcore mapping under under-/over-subscription is no overlapping; in other words, any scheduling concerns are excluded in those circumstances. To do so, balanced mapping is the assumption.

2.7 Conclusion

This chapter presents a problem definition for each of **`vcore-mapping`**, **`page-mapping`**, and **`vcore-scheduling`**. The idea is to clarify each problem; however, the problems may interact with each other. Virtual core mapping and page mapping are clearly the case where one configuration affects the other. As we combine problems, these problem statements may need to reformulated and expanded.

This chapter also discuss possible approaches to solve the problems. For the combined problems, the ILP-based approach demands a high accuracy in the problem formulations. Whereas, the heuristic approach over the ILP-based optimization approach has great flexibility in solving the problems; therefore, the heuristic approach is selected to solve the optimization problems.

Lastly, classes of different resource demand situations are formed and illustrated in this chapter. They are divided particularly with respect to the resource demand perspective. That classification actually affects system design, especially for policy and detection component design. There are three classes: under-subscription, full-subscription, and over-subscription.

Chapter 3

Design of NAVAR Adaptive System

As claimed in the thesis statement in Chapter 1, the system should automatically select the configuration that enables optimal or near-optimal execution of applications in virtualized environments for the user selected objectives. Obviously, this automatic reconfiguration is conducted during a runtime. In fact automatic reconfiguration is the goal for the design of this adaptive system. To reach the goal, the system incorporates *adaptation mechanisms*. The problems described in last section imply potential mechanisms that include 1) moving vcores to different physical cores (vcore migration mechanism), 2) migrating pages to memory addresses in a different NUMA domain (memory migration mechanism), and 3) gang-/co-scheduling vcores taken by concurrent guest threads in the same time slots, or alternatively assigning a priority to a set of vcores that exhibit high concurrency (gang-/co-scheduling mechanism). A policy component drives these mechanisms. The *policy component* will determine the best configuration and mechanisms. To do so, the policy component needs a set of accurate information. The *detection framework* provides the assorted *metrics* to the decision maker in a real time. Additionally, it needs to be mentioned that Palacios VMM is used for the base framework. Palacios is an OS-independent,

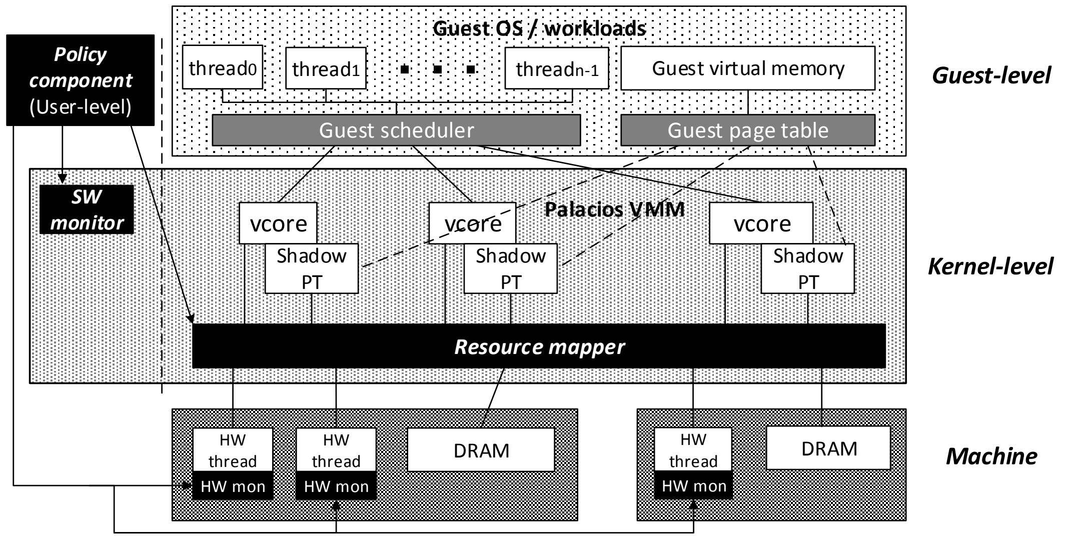


Figure 3.1: Illustration of overall adaptive system. The framework is the same regardless of the three resource demand cases: under-subscription, full-subscription, and over-subscription. Policy component includes control loop and algorithm; resource mapping contains three mechanisms: vcore migration, memory migration, and gang-/co-scheduler. Two types of monitors – software monitors and hardware monitors – comprise the detection framework.

open source, BSD-licensed, publicly available embeddable VMM designed as part of the V3VEE project (`http://v3vee.org`). Detailed information about Palacios can be found in various articles [78, 76, 77]. Palacios' OS-agnostic design allows itself to be embedded into a wide range of different OS architectures. I use Palacios embedded in the Linux kernel.

In this chapter, each section introduces a component design, including viable options and the direction which I have chosen for NAVAR. Mechanisms are described more thoroughly here when implementations of them have not been revised during experimental tests. Details of mechanism, which are very coupled with experimental setups, will be

	R410	R415
Processors (2)	Intel Xeon E5620 2.4 GHz Num. of Cores: 4 Num. of Hardware Threads: 8 (2-way SMT per core) Max TDP: 80 W	AMD Opteron 4122 2.2 GHz Num. of Cores: 4 Max TDP: 115 W
Processor Sockets	2	2
Cache	L1: 64KB x 4 (32KB L1 Data, 32KB L1 Inst.) L2: 256KB x 4 L3: 12MB	L1: 128KB x 4 (64KB L1 Data, 64KB L1 Inst.) L2: 512KB x 4 L3: 6MB
Memory	32GB (8GB x 4) 1066 MHz (DDR3)	16GB (4GB x 4) 1333 MHz (DDR3)
Power Supply	480W	480W

Figure 3.2: Features of test machines (Dell PowerEdge R410 and R415).

described in the following chapters. Particular cases here are the policy and detection frameworks.

Each class is boiled down to some specific case on which experimental tests are based; nonetheless, test machines and workloads are the same regardless. They are also described in the chapter. Lastly, one hardware mechanism is proposed near to the end of this chapter. It is the functionality to power-off multiple DIMMs in one NUMA domain if necessary. To get a sense of the benefit of the proposed hardware mechanism, it is assumed that the system has the feature and estimations based on actual measurement data are included in the final results. Details can be found in Section 3.4.

3.1 Testbed and Workloads

First, we now describe the testbeds and workloads used in overall work.

3.1.1 Hardware

Testbeds Figure 3.2 describes the two test systems used in my work – Dell PowerEdged R410 and R415 machines. Both have two processor sockets – two NUMA domains. The R410 has Xeon E5620 processors with four physical cores, each of which has two hardware threads. The R415 has AMD Opteron 4122 processor with four physical cores (single hardware threads per processor). They are both a small scale Non-Uniform memory-access (NUMA) architecture, in that each socket is preferentially associated with half of system memory. The machine is configured for performance according to Dell's recommendations in [82]. Specifically, node interleaving is turned off for memory allocation to make the effect of distance in accessing memory clear. Also, the system is to minimize variations on the performance, energy and/or power consumption due to DVFS, by setting a static frequency and voltage and turning off turbo mode. For the idle state in each core, the C-state option, including enhanced mode, is enabled with the idea being to maximize the dynamic power reduction when the socket is idled. Note that R410 machine is used for most experimental results. All results throughout the manuscript are from R410 unless specified. Chapter 8 discusses about R415 results. The R415 is used to demonstrate that the same kinds of tradeoffs and accordingly online adaptivity are valid on other hardware.

Energy/power measurement To measure energy consumption, an external power meter, a Watts Up PRO, is connected to power source. It measures system energy and power, and its serial output is fed into a separate machine that is dedicated for logging. The monitoring machine records time-stamped cumulative energy from the beginning and the end of execution. Since the precision of instantaneous power measurement is a bit low, a sample per second, average power (Watts) is taken from measured energy (Wh) and spent time (seconds). However, occasionally power value is read directly from the meter to get a shape of power over time. It is important to know an assumption that when multiple VMs are running, system power is supposed to evenly distributed by each VM. This assumption is valid when CPU- and memory-bound workloads are running in VMs.

3.1.2 Software

Three layers of software stack comprise the overall system. Host and guest OSes are Linux and the Palacios VMM is used for the virtualization framework. The host Linux distribution is off-the-shelf Fedora 15 with kernel 2.6.38. Two versions of Linux guest kernels are used. The work appeared in Chapter 4 was done earlier and kernel 2.6.30.4 was used. All the rest of work are on top of kernel 2.6.39. Other than the kernel version, the guest kernel image is built with 64-bit mode, while guest images used in Chapter 4 are built for 32-bit mode. Application images on top of that are all kept unchanged, each of which is compiled for 32-bit mode. Additionally, the baseline Palacios versions are different between the two set of works. One is from earlier commit, hash: #b8759fe01196884bea04eb9a1dd09781d0 605d47, that is used in Chapter 4, while #bae592fa0e54e3ce5933e84e4b15815ba0501618 is the other baseline version used for the rest of the work. Notice that the structure of shadow page table in Palacios is aligned with that in the guest; so, 32-bit mode and 64-bit mode page tables are used respectively.

Multithreaded parallel benchmarks are run using OpenMP and pthreads for parallelism. Specifically, the benchmarks are from the SPEC OMP [13], NAS [66] and PARSEC [26] suites. They are all CPU- and memory-bound workloads. Source codes are compiled using gcc or Intel compilers.

3.2 Detection Framework

Demand for detection framework To solve the problems defined in Chapter 2, it is important to measure objective metrics and also to figure out constraints to doing so. In fact, objective values such as performance (makespan), energy and power, are drawn from direct measurements, indirect model base estimations, or a metric that is strongly coupled with the objective value. Discussions here focus on the three objective metrics: power, execution time and energy. Power is dynamically measurable with an external meter or can be estimated by models; nonetheless, a power model is more powerful for its predictability. Addressing **`vcore-mapping`**, measuring power on every mapping case is sometimes inefficient or even an intrusive behavior. For the performance measurements, makespan is unknown until execution ends. Workload execution is repeatable and deterministic under a fixed configuration; however, measurability in the middle of execution is a different matter. If there is any metric that is highly correlated with performance in makespan and also measurable during the execution, the metric can be used. Lastly, energy is highly coupled with performance, as long as factors affecting the performance do not result in significant fluctuations on instantaneous power across the timeline. Sometimes turning on/off DIMMs in a certain NUMA domain may or may not change execution time. The changes in the memory power state affect power at the same time. When comparing the two cases for energy consumption in one execution, the energy metric is not always in the same line with the execution time. Sometimes, regardless of the number of active NUMA domain memories, performance is drawn to be equal or with minimal changes. In this situation, turning off some DIMMs takes less energy, but execution time does not show differences. Except for these cases, however, mostly execution time tends to be correlated with energy.

The detection framework comprises two types of monitors: software monitors and hardware monitors[1], based on the level where the monitor is implemented.

3.2.1 Hardware-assisted Software Monitors

A software monitor is a viable option to be considered; however, what is different from hardware monitor is that software monitor needs to be defined. Before specifying feature of software monitor, what is demanded should be clarified. The demand is mostly about workload characterization in terms of shared memory traffic, which is most significantly coupled with the workload performance. A model should be usable to estimate the best mapping for vcores and pages. With that, page tables are immediately a critical source of data. Page tables are commonly accessible data structures and especially effective from a VMM perspective.

Usable mechanism for software monitor Now, to move on to specifics of design of the software monitor, some key mechanisms need to be mentioned and described. We make use of the following elements:

- The x86 PMU to demarcate intervals of memory-accesses and memory store [2].
- The x86 shadow or nested page table entries' accessed and dirty bits to partition the guest physical address space's pages into sets of pages that have been accessed or written in an interval
- Periodic synchronization across the hardware threads to get a global view of the accessed and written sets, and compute jointly accessed pages across two or more vcores.

[1]Hardware performance counters, known as the Performance Monitoring Unit (PMU), are available in most modern processors.

[2]This PMU feature is currently more Intel processor specific.

Hardware monitor defined intervals The PMU allows us to trigger exceptions (and hence VM exits) after a certain number of events have occurred, such as instruction retirements or memory references. This facility is used to produce VM exits after a specified number of memory-accesses or stores have occurred; thus, the PMU creates the measurement windows over which the metrics are collected. This incorporation compensates for the coarse granularity of software monitor. This is an important point that distinguishes a hardware-assisted software monitor from a typical software monitor. Without a hardware-defined interval, the framework would be the same as a typical software monitor.

Page access information A page table update mechanism is incorporated into hardware component and is already used by a commodity OS; What is additionally implemented here is a page table scanner. The x86 architecture incorporates a detailed model of paging that includes *accessed* and *dirty* bits on the page table entries (PTEs). The hardware will ensure that the accessed bit is set on the first read or write of a given page, and that the dirty bit is set on the first write. These bits instrument accesses to the memory pointed to by the shadow page tables, which contain the combined intent of the guest virtual to guest physical mapping and the guest physical to host physical mapping. Because the VMM controls these page tables and the latter mapping, it can easily determine the guest physical pages that are being accessed. In addition, it can manipulate the accessed and dirty bits as much as needed so long as it projects the expected hardware behavior to the guest, using ancillary information it keeps. Alternatively the nested page tables can be instrumented.

Mechanism to collect page access profile Using these two mechanisms, at the beginning of a measurement interval for an individual vcore, the VMM clears the accessed and dirty bits on all valid shadow or nested page table entries, keeping ancillary information about the real values of these bits for shadow page tables.[3] It then sets the PMU to produce an exception after the desired interval of memory-accesses or writes. Execution then proceeds as normal, with the saved bits used in paging-related exits. Eventually, the PMU raises the exception, inducing an exit to the VMM. The VMM then walks the page tables and records information about pages with the accessed bit set, store bit set, or no bit set. These sets are stored as bit vector indices over the guest physical address space. Therefore, multiple sampling constitutes bitmaps as many as the number of samples. A couple of logical operators, AND and OR operations, are taken to merge them. This will be revisited later in this section.

Aggregation of information All of above mechanisms are executing per each vcore context; so, the information collected by now is all local so far. Now, a separate thread, named the *aggregator*, runs on a distinct hardware thread that is not used by any vcore. This thread periodically collects and aggregates the information collected by each vcore. As depicted in Figure 3.3, the *aggregator* determines interval between collection operations. Each information for write or read is stored in bitmap.

Parameters in detection framework Although the described mechanism is summarized above, there are parameters to determine the different kinds of metrics and characteristics that are computed. The four parameters are:

1. The frequency of scanning page tables and which hardware monitor event is used to define the page scanning interval – the number of memory operation or the number of store operation.

2. Which page table information is considered – access bit or dirty bit

[3]Note that in Palacios each vcore has a distinct shadow or nested page table, even if it is running a thread that shares a guest page table with some other thread.

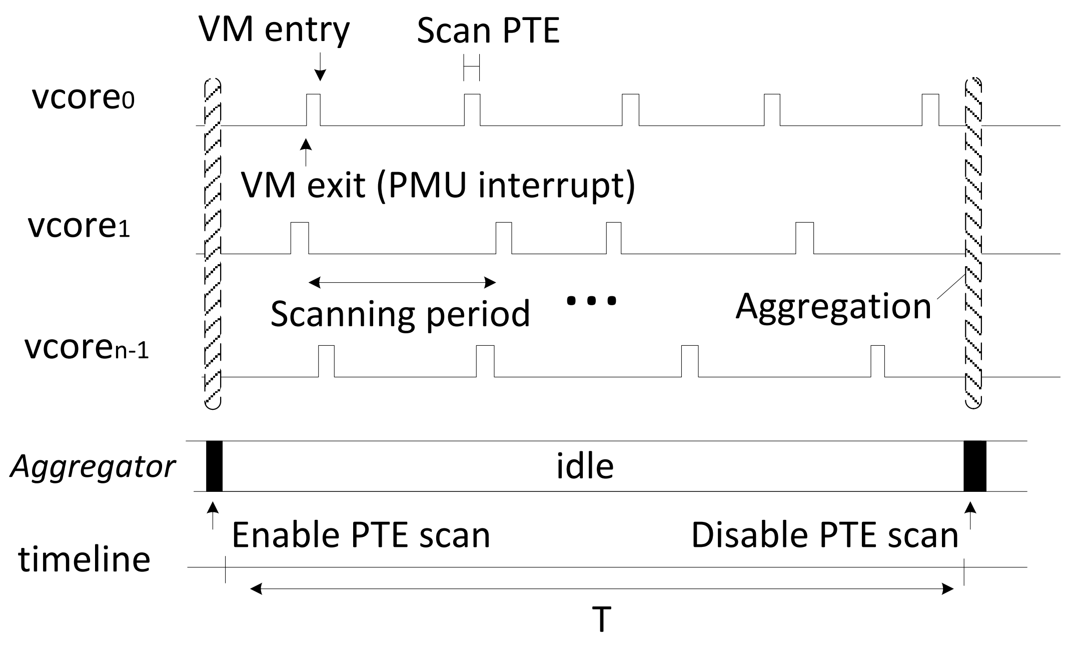


Figure 3.3: Illustration of probing on a timeline. During the probe interval T, each virtual core scans its page table independently. At the end of the probing, information on accessed (or written) pages is collected from all vcores and the metrics are computed from it.

3. How to merge the bitmaps on each vcore – AND (Intersect) or OR (union) operation
4. How to aggregate bitmaps across vcores – sum of all bitmaps' weight, union of all bitmaps, and so on[4].
5. How long consecutive page table scanning interval

Access and write operations have different implications. Memory-accesses capture the volume of interaction with the memory system, while writes are related to amount of invalidation traffic. Hence, per-access and per-write as well as access-info and write-info in

[4]Chapter 4 uses different ways of bitmap aggregation across vcores.

page table are important parameters – *1* and *2*. In merging bitmaps in a vcore, the AND operation filters pages that are repeatedly accessed during the period of time. That captures hot pages. The OR operation, in contrast, counts the overall number of pages that at least touched. This quantity is highly correlated with the working set size (WSS). Parameter 3 is quite significant. Collection of bitmaps across vcores (parameter 4) gives different views to characterize shared memory traffic, such as how many pages are accessed by vcores, and how many owners (vcores) for each page. The probe duration (parameter 5) allows us to profile different degrees of hot pages. Besides its implication on the metrics, the idea for setting period of time is measuring memory-access traffic sparsely. Even within this probe duration, page table entries are only scanned on intervals that hardware monitor defines.

Choice of metrics and limits Besides the underlying meaning of the metrics, the limit of each metrics is important to understand. Metrics are generally collected at the page granularity. A weakness here, compared to cache-line granularity, is potential false sharing; nonetheless, most application-level inter-thread sharing is at the page granularity and this makes up the vast majority of shared page accesses, especially in parallel codes. Also, the selection of the metric is very case-sensitive and also important for correct decisions. Here, two methods can be taken to choose key metrics. One is to check the correlation of the metrics to the objective metric – power, energy and execution time. The other is a cross-correlation check between the metrics. This allow us to filter out to metrics which are substitutable. Chapter 4, 5, and 6 include details on metrics and pseudocode for collecting the selected metrics.

3.3 Adaptation Mechanisms

This section describes the implementation of adaptation mechanisms to address resource management problems. Each mechanism is influenced by one of our three problems. Along with implementation details, design perspectives are also discussed here. The mechanisms have been implemented in Palacios, but could be ported into other VMMs.

Two coarse-grained mapping strategies for vcores and memory (pages) Before moving on, two common mapping strategies need to be mentioned as they are very commonly used throughout this and the following chapters.

- *Consolidated* - Given multiple NUMA domains, this mapping style consolidates all vcores or pages that belong to one VM into one domain. Sometimes, this mapping style is called *local*, in case when both vcores and pages are consolidated in the same NUMA domain.

- *Interleaved* - Mapping vcores or pages across multiple domains is the principle. The number of vcore or pages allocated per domain should be the same; thus, this mapping strategy tries to evenly spread out vcores or pages. Although it is mentioned that the mapping style is supposed to spread across entire domains in a machine, sometimes it includes cases that spread them across some selected NUMA domains. This variant will be discussed in Chapter 8. The spreading method is taking modulo-n, when there are many pages, n is the number of destination NUMA domains. Details on the page mapping specifics will be discussed in Section A.4.

Summary of mechanisms Here is a brief summary of mechanism implementations in Palacios. The vcore migration mechanism is essentially migrating a kernel thread to a different hardware thread, which is used to address the `vcore-mapping` problem. For

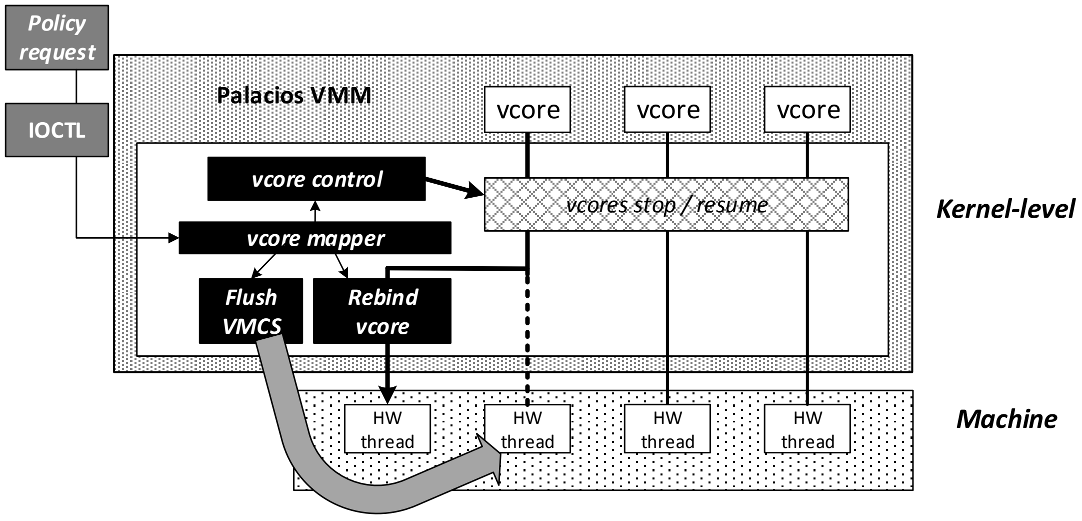


Figure 3.4: Illustration of the vcore migration mechanism. It shows migrating one vcore to different hardware thread. The major cost lies on flushing the cache contents (and the VMCS) as well as suspending/resuming all vcores of the VM.

a page migration mechanism for **`page-mapping`**, the memory map for each page of a guest needs to be extended to support multiple regions. Palacios allocates guest physical memory space via a contiguous host physical memory space. To support multiple contiguous memory regions, the memory map of the guest physical memory was extended with a hash table. Lastly, to address **`vcore-scheduling`**, some scheduling features need to be in VMM context. Most of the generic scheduling mechanisms can be provided by the host OS. Linux, for example, has generic CPU scheduler targeted for fairness. The objective of the host scheduler may not be sufficient in scheduling vcores of VMs due to the lack of context. The VMM, in a contrast, has context information on what vcores are from what VMs; thus, it is more effective for VMM to have control over scheduling, particularly in regards to, gang-/co-scheduling needs.

3.3.1 Virtual Core Migration

Palacios' Multicore supports Palacios supports a multicore VM that appears to the guest to be a physical machine that is compatible with the Intel Multiprocessor Specification. The guest sees an MP table describing the processors, APICs, IOAPICs, buses, and interrupt routing in the machine, and virtual versions of standard APIC/IOAPIC interrupt controller hardware.

Process of virtual core migration Palacios backs each virtual core with a host OS kernel thread that is bound to a specific physical core at VM startup time, and that can be remapped at any point. The mapping of virtual core threads to physical cores does not change except in response to explicit requests, which can be invoked from a user-space tool on the host. The call specifies a new mapping of all or some of the virtual cores. To handle the request, Palacios first uses physical IPIs to force all the virtual cores to exit to the VMM and synchronize. It follows this by rebinding their host kernel threads, and handing the relevant VT or SVM state to the new physical core. The threads synchronize again, and then reenter the guest. The process of migrating a virtual core thread to a different physical core functions is as follows, which is depicted in Figure 3.4:

- Detache the VM control structure (VMCS in the Intel VMX or VMCB in the AMD SVM implementation) associated with the virtual core from its source physical core (for example, by VMCLEAR on a VMX machine). This is to ensure that the VM control structure data that may be cached by the processor is flushed to memory.
- Unbind the virtual core thread from its source hardware thread and bind it to the destination physical core. This tells the host OS to de-schedule the thread from its source physical core and reschedule it on the destination physical core.

Benchmark	Page Table Scanning	VCore Remapping	Memory Remapping	VCore Scheduling
CANNEAL	1.51	5.24	913.34	0.0008
STREAMCLUSTER	0.78	5.27	915.0	0.001
EQAUKE	0.82	5.25	887.0	0.0012
SWIM	2.34	5.08	912.3	0.0012
RAYTRACE	0.39	5.24	799.33	0.0004
MGRID	0.61	5.27	892.86	0.0017
FLUIDANIMATE	0.58	5.25	891.6	0.0009
ART	1.30	5.30	892.44	0.001
APSI	4.61	5.27	920.57	0.0009

Table 3.1: Page table scanning time (average per each scan), vcore remapping time (moving four vcore threads), memory remapping time (moving guest memory to different NUMA domains), and vcore scheduling time (reconfiguring vcore scheduling policy) in milliseconds. Note that the overhead for the memory mapping mostly comes form copying pages. The memory mapping mechanism is implemented to copy pages unconditionally whether touched or not.

- Once the virtual core thread gets to run on the destination hardware thread, it loads and attaches the VM control structure data (for example, by performing a VMPTRLD on the destination hardware thread in VMX implementation).

Performance overhead The physical core-specific costs of migration consist of the very low fixed costs of changing a tiny number of VMX or SVM-specific control registers, and the costs of refilling cached state (cache, TLB, control structure caches, page hierarchy caches, etc) on the destination physical core. Additionally, there is the cost of synchronization among the physical cores. Each migration needs to suspend vcore execution in between detaches and attaches. If remapping is very frequent, this migration mechanism significantly worsens performance. However, the adaptive system is designed to take overhead into account.

Power overhead Power in the whole system should not be higher than normal execution for vcore migration. Migrating cache contention sometimes includes bus traffic in cases where migration happens across NUMA domains; nonetheless, all suspended vcores, as long as only one vcore mapped to a hardware thread, are supposed to power-down the physical core. Overall, reduced core power mitigates intra-/inter-core power overheads.

Support for general remapping: multiple remapping The migration mechanism described above is for each vcore. When the decision is made to migrate multiple vcores, it would be rather efficient to detach and attach multiple vcores at the same time; then, what is expected is suspending/resuming all vcores for each migration can be reduced to one from the number of vcores to be migrated. The idea is supporting transitions between *interleaved* and *consolidated* mapping strategies. To support multiple vcore remapping, a new interface exposed to user-level policy component has been added. The interface for single migration contains source vcore, destination physical core information, and target VM pointer. For multiple vcore migrations, the destination bitmap and target VM information are provided through the interface.

3.3.2 Page Migration

Palacios' memory mapping Palacios was originally designed to allocate guest memory in a contiguous chunk of physical memory. Here, one contiguous physical memory chunk may be called a *region*. The idea behind this contiguous memory design is to expose attributes of physical memory to high performance workloads for its optimization in the guest context. The current implementation at the time of forking from the main repository uses offline memory functionality that the host Linux provides. While launching Palacios, enough memory blocks are offlined to support multiple guests. Once memory is hidden

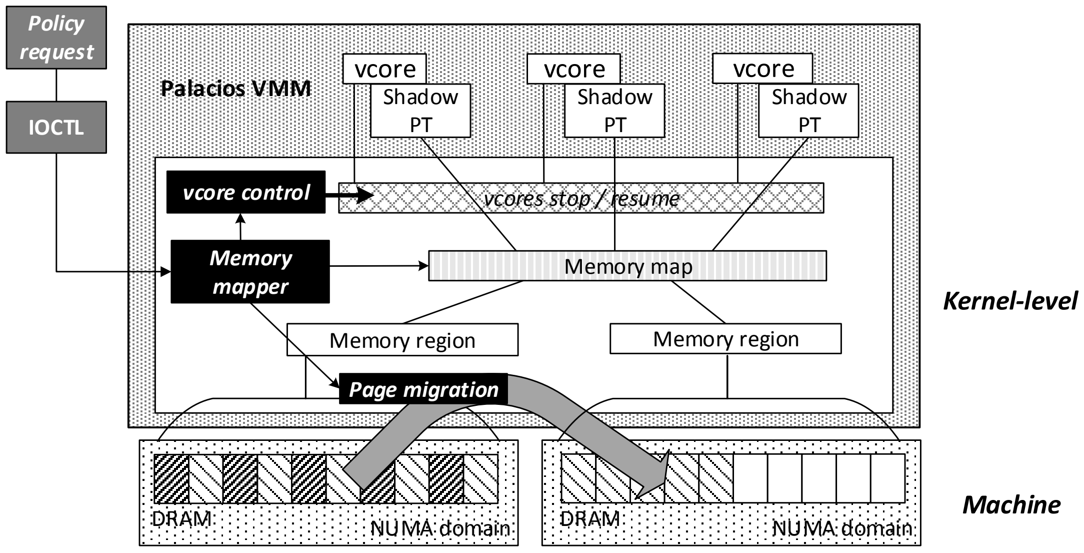


Figure 3.5: Illustration of the page migration mechanism. This is a case migrating page mapping from *consolidated* to *interleaved*. Migration cost is linear to amount of pages to be copied which also invokes more power. Physical address referenced by shadow page table (or nested page table) is routed via memory map and region information to a host physical address.

from the host, Palacios is now able to allocate each guest's memory from the offlined memory. What Palacios does for each VM is pointing out a start address and an offset that reaches the end of the physical memory. Therefore, one region per VM is allocated.

NUMA-aware multiple memory regions The memory model with one *region* should be fine with an SMP or "UMA" architecture as long as there is no pressure on host physical memory. Also, note that on top of the allocated memory, guest OS has its own memory manager that does memory fragmentation for its own usage. Now, when a NUMA architecture is the target, some issues are raised. In a mechanism perspective, a question is raised about how to implement multiple regions either transparently provided or explicitly.

Considering the scope of the defined problems and a direction on inference base approach, multiple guest physical memory regions need to be provided transparently. Now in terms of fragmented regions of memory, coarse-granularity is one option, that is to let each region to be contiguous over host physical addresses, and each region then ties to a NUMA domain. A typical NUMA architecture today includes a mapping of host physical address to NUMA domain. Therefore, implementation-wise, multiple regions equal in the number of the NUMA nodes are needed. That constitutes information that includes base address (start of a region) and offset (end of a region). Offset information for each NUMA domain is given by a policy component at the beginning or in the middle of guest runtime when the policy decides memory remapping is needed. In our prototype, a base address is determined per request by the memory mapper via the memory map, which is illustrated in Figure 3.5. A whole guest memory is allocated from offline memory where Palacios has full controllability; so, the memory manager is capable of (re-)allocating and deallocating an arbitrary amount of memory, down to the page granularity, as long as that fits in the reserved space. Once multiple regions are provided, Palacios can map each guest page to one of regions via a shadow or nested page table. In particular, here is the flow of memory remapping in runtime:

- Suspend all vcore execution per memory re-mapping request.
- Request demanded amount of memory from the memory manager along with NUMA domain information.
- Retrieve source NUMA domain per page, then migrate page content to the destination if destination NUMA domain is different. Memory map keeps the location in NUMA domain (the number of memory *region* as depicted in Figure 3.5) for each guest page and is updated per memory migration[5]. The memory map is global and

[5]Page migration and memory migration are used interchangeably to refer to one mechanism.

only allows atomic accesses. Note that some policies, such as first-touch, requires page location to be determined at runtime; for such a policy, the mechanism conservatively replicates a page across all NUMA domains.

- Request the memory manager to deallocate source pages that are no longer used after the remapping.
- Flush all shadow (or nested) page tables, and let all vcores proceed with execution.
- Palacios remaps pages per shaodw or nested page fault in each vcore. For policies that allow predetermined page location, location information is read from the global meta data per pages, but when a page location needs to be determined later, the meta data is atomically updated by the very first vcore that addresses page fault.

Performance and power overheads Similar to vcore migrations, all vcores are suspended during migrations. The amount of suspension time is linear to the time taken by the migration, which is a function of the amount of memory and the distance to migrates. Because of this, memory migrations tend to have more overhead than vcore migrations. Typically, memory footprint ranges in the gigabytes. Memory migration to a different NUMA domain routes to off-chip paths and causes increased power. On contrary, the amount of data for vcore migration is a few mega bytes that cover all register states and some memory footprints. A detailed comparison of overhead is depicted in Table 3.1. The implemented memory migration mechanism is intended to meet key functional requirements that smoothly require and release physical memory as well as to correctly remap guest page locations in the NUMA context. Many optimization strategies are possible; nonetheless, even with this basic version, if memory migration is sparsely requested, the opportunities for memory migration are available. The primary overhead is space, not time.

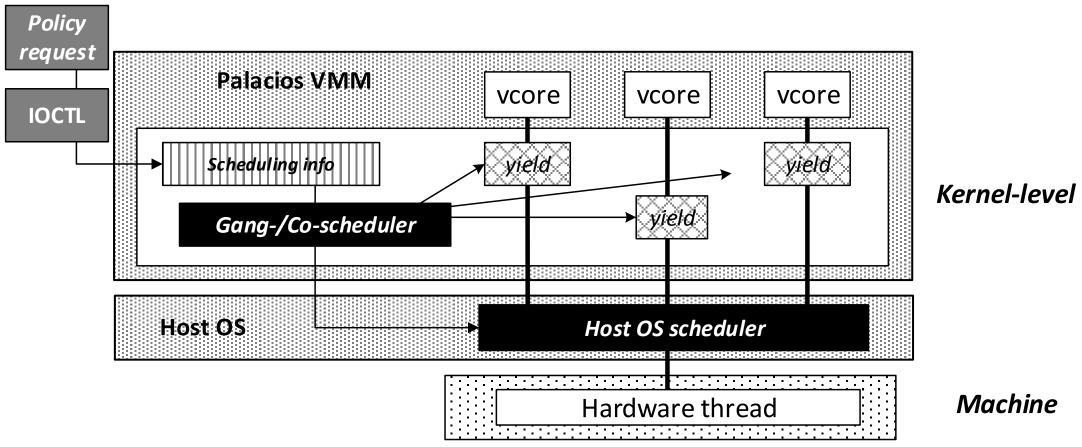


Figure 3.6: Illustration of the scheduling mechanism.

3.3.3 Gang-/Co-scheduling Mechanism

VMM scheduler vs. host scheduler In a virtualized system, gang-scheduling or co-scheduling technique can be beneficial. On top of the host scheduler, the VMM scheduling mechanism[6] can leverage the VMM perspectives to improve performance. Mechanism-wise, our key focus here is on the co-scheduling implementation, which is currently missing in both base Palacios code (VMM) and Linux kernel (host OS). In general, the scheduling mechanism that host OS provides is used to schedule vcores on a hardware thread. For gang-scheduling, the decisions are made about whether a vcore should yield or proceed when a host scheduler allows it to run. Even if a host scheduler allows a vcore to proceed, a vcore can be suspended by the Palacios VMM by yielding to another vcore in the hardware thread. When a vcore is in a group being gang-scheduled and the current time slice is allocated to the group, the vcore should be running. To do so, other vcores mapped to the hardware thread need to be suspended for the time slice. On the other hand, in other

[6]Gang-scheduling and co-scheduling is collectively called VMM scheduling mechanism here.

time slices, the vcore in the group should yield to other vcores; and, the other vcores may then proceed when allowed by the host scheduler. In short, a host scheduler decides what vcore may run, but VMM can control whether the granted vcore should be running or not. With that controllability, a gang-scheduler can be realized, as can other kinds ofscheduler. The following paragraphs describe the design choice and some considerations related to overhead. An initial version of implementation is then described in pseudocode.

Two types of implementations: leader-driven vs. flat models From the perspective of the scheduling mechanism, gang-scheduling and co-scheduling share numerous implementations. One is a variant of the other; the difference is in the scope of vcore grouping so that vcores in a group are simultaneously scheduled in selected time slots. Both scheduling mechanisms schedule a group of vcores in selected time slices, which is simple and the core of the mechanism on top of the default scheduler that the host OS provides.

There are two approaches for this: one is a leader-driven model and the other is a flat model. In the leader-driven approach, one of the vcores in a selected group is picked as the leader and all the other vcores follow the execution schedule of the leader via a signal or a global tag. When the leader goes wrong, this model is fragile and may face fairness issues like starvation.

The other approach picks a coordinator vcore – the first vcore that sees expiration of time ticks since the last scheduled time. Any vcore can be the coordinator. The coordinator is responsible for signaling all the remaining vcores to be scheduled. With external time ticks, a group of vcores are guaranteed to be scheduled in a timely manner, but this model could possibly cause more CPU fragmentation. All other scheduling queues other than the coordinator do not voluntarily yield. The leader-driven approach has at least some consistency, taking one specific running queue as its reference.

For these reasons, the leader-driven approach was chosen for its directness. The core of the implementation is on managing global information, which leads to support gang-scheduling. Read and write operations to the global tag are separate depending on who the leaders or the followers are. However, while deploying some real workloads, some performance issues were observed. Before addressing optimizations themselves – these are discussed in Chaper 7 – why the mechanism has to be optimized needs to be explained. Note that the other two mechanisms also need improvements in terms of performance; nonetheless, they met functional requirements, which are sufficient to draw experimental results, a proof-of-concept. There are a few points that distinguish this mechanism from other two.

One time event vs a continuous/underlying factor The comparison between mechanisms might begin by comparing *explicit* versus *implicit overheads*. Here, *explicit overhead* is defined as a cost that is explicitly distinguishable and, therefore, measurable in a timestamp, whereas *implicit overhead*, as defined here, is a cost that is hard to distinguish in a timeline. Cache refill, for example, sometimes happens speculatively and transparently, so it is rather hard to explicitly measure – there are sometimes so-called noise effects.

Given that, the two migration mechanisms and the scheduling mechanism lie in different categories in terms of the two overheads. First, the two migration mechanisms take more *explicit overhead* than *implicit overhead* and oppose the scheduling mechanism. The *explicit overheads* in the first two cases are measurable as an amount of expired time, during which vcores are suspended to migrate contents in cache-level or in memory-level. The *implicit overheads* in those cases are cache refills and reconstructions of flushed page tables, which are variable and hard to measure.

In contrast, the scheduling mechanism takes a very short time to update some global tags for the taken implementation approach. While the *explicit overhead* is small, side effects from the scheduling mechanisms are sustained longer and deeper, which leads to huge implementation costs.

For that reason, the scheduling mechanism has been iteratively revised to reduce such implicit side effects. The two mechanisms for vcore migration and memory migration are not optimized as much as the scheduling mechanism. The *explicit overhead* is compensated for by adaptive policy; one of the roles of which is to control *explicit overhead* by leveraging the sleep interval for the policy component. Here an initial and strict version of gang-scheduler is described along with some Palacios scheduling designs. Later, some perspective to change it is detailed in Chapter 7.

Initial version of gang-scheduler The gang-scheduler is implemented in a strict way under the leader-driven model. As mentioned, the following pseudo code is an initial version. Some revisions made can be found in Chapter 7. This version might be called a strict version to be separated from others – a relaxed version. The terms relaxed and strict are based on the differences between whether vcores in a gang are waiting for all vcores ot become ready for execution in the same time slice or not. The relaxed version is basically to let a vcore go as soon as it is ready. More detailed, empirical studies for those variants will be discussed in Chapter 7. The following pseudocode is implemented in Palacios' yield function (CondYield()), which is called during every VM exits.

It is important to understand that there is one global variable, $SchedState$, per hardware thread. The variable is used to convey state information for gang-scheduling. $GangLeader$ is put on the hardware thread where the gang leader vcore is running. As mentioned, all followers sync with the leader's schedule. Whenever the gang leader vcore is scheduled, it is supposed to broadcast the states to all the remaining hardware threads. At first, given its turn, $GangPending$ is marked on all the followers' hardware threads. In this global state, followers are busy waiting until its state changed to $GangRequested$ by the leader. Once changed to *GangRequested*, the vcore switches *LocalState* to *Ready*. Here *LocalState* is local variable and is maintained in the vcore context. This local variable is written only by a follower; so, it shows response to leader-driven *SchedState* changes. Therefore, leader keeps tracking all follower's local states until all followers are ready (*Ready*); then, finally the leader updates followers' state to *GangRequested*. With that, followers are now allowed to proceed with changing *LocalState* to *Running* . Under *GangRequested*, vcores outside of the gang are supposed to yield – calling the host scheduler. On the other hand, once vcores in the gang are done with a time slice, they change *LocalState* to *Waiting* and also update *SchedState* to *WaitForGang*. This is only a point when *SchedState* is updated by followers. For an expiration of time slice, each vcore is expected to check by itself for it is cheaper and more accurate than a case entirely controlled by a leader. Now, along with *SchedState* as global state, the local state is kept as *Wait* until leader updates followers' global state to *GangPending*. Lastly, regarding notations in the following pseudocodes, *VMID* represents a unique identification number for one VM and **schedule()** calls host scheduler per yielding due to time slice expiration. With this high-level idea in mind, precise steps are following:

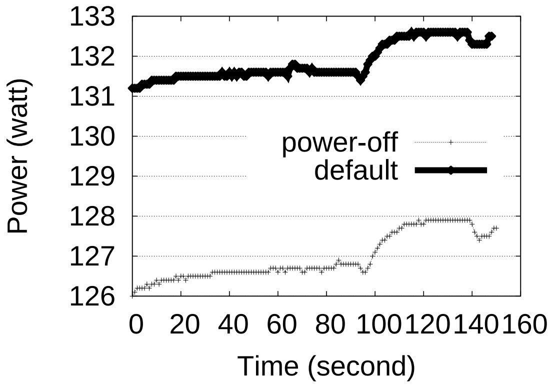
```
CondYield(...)
  ...
  switch (SchedState)
  case GangLeader:
    if VMID is the VM gang-schedule selected then
      if all vcores are in Running of LocalState then
        if time slice expires then
          call schedule()
          update SchedState of all the rest HW threads to be GangPending
```

```
            keep waiting until all vcores in $Ready$ of $LocalState$ or $WaitLimit$ expires
            update $SchedState$ of all the rest HW threads to be $GangRequested$
          end if
        else
          update $SchedState$ of all the rest HW threads to be $GangPending$
          keep waiting until all vcores in $Ready$ of $LocalState$ or $WaitLimit$ expires
          update $SchedState$ of all the rest HW threads to be $GangRequested$
        end if
      else
        if time slice expires then
          call schedule()
        end if
      end if
    case $GangPending$:
      if $VMID$ is the VM gang-schedule selected then
        if $LocalState$ is not $Ready$ then
          update $LocalState$ to $Ready$
          wait until $SchedState$ to be $GangRequested$
          change $LocalState$ to $Running$
        end if
      else
        call schedule()
      end if
    case $GangRequested$:
      if $VMID$ is the VM gang-schedule selected then
        if time slice expires then
```

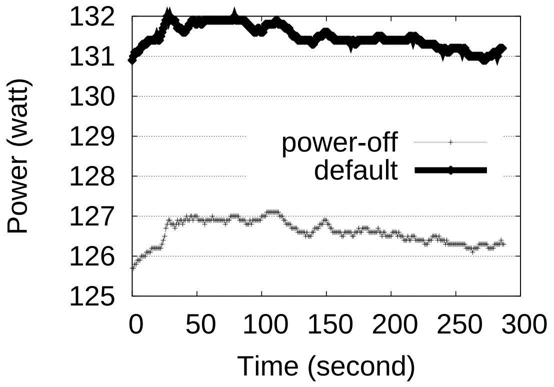
Benchmark	Power-off	Default
CANNEAL	126W	131W
FREQMINE	126W	131W
CG	135W	140W
X264	127W	133W

Table 3.2: Preliminary power measurement results. Regardless of workloads, the power reduction by the DIMM power-off is consistently observed, 5W on average. The vcore and memory mapping is the *consolidated* for both.

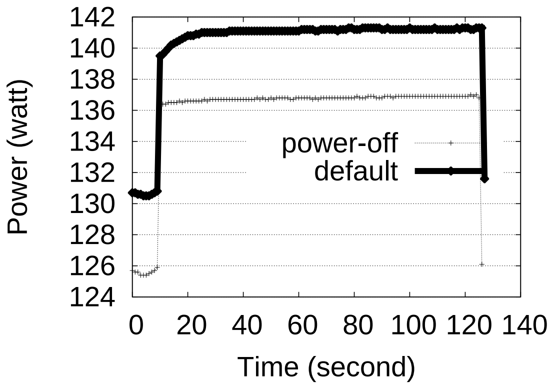
```
            set $SchedState$ to be $WaitForGang$
            set $LocalState$ to $Wait$
            call **schedule()**
        **end if**
    **else**
        call **schedule()**
    **end if**
**case** $WaitForGang$**:**
    **if** $VMID$ is the VM gang-schedule selected **then**
        call **schedule()**
    **else**
        **if** time slice expires **then**
            call **schedule()**
        **end if**
    **end if**
**end switch**
```



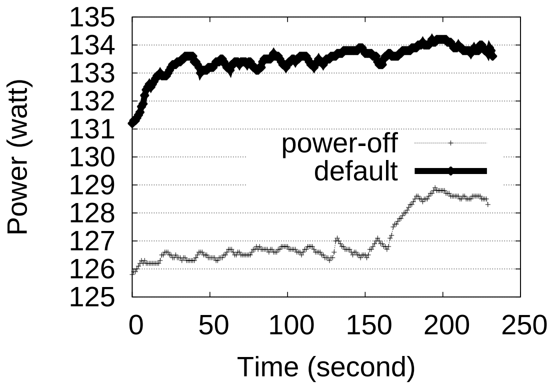
(a) CANNEAL



(b) FREQMINE



(c) CG



(d) X264

Figure 3.7: Illustration of instantaneous power shape between the default and the power-off cases. The vcore and memory mapping is the *consolidated* for both. For the power-off setup, all DIMMs on one NUMA domain, where no guest memory is allocated via the *consolidated* memory mapping, are physically detached.

3.4 Power-off Memory Sub-system

Some experiments assume the hardware capability of powering-off the entire memory in each NUMA domain. We now describe this proposal the hardware feature and how we estimate the experimental results.

3.4.1 Abstraction of Proposed Hardware Mechanism

Power concern in modern computing systems is not a new issue. Processors have already been designed to save energy by exposing control knobs to software such as the DVFS mechanism or transparent control like deep sleep modes. For the memory system, some recent work [15, 41, 42] proposes similar power control mechanisms. According to DDR specifications, there are various power modes; nonetheless, there is still room to save. I measured that the removal of memory in one NUMA domain in the test machine reduces power by 5 – 6 watts on average as shown in Figure 3.7 and Table 3.2. The portion of power in the entire system varies by the amount of load and CPU frequency, but it is typically in the range of 4 – 7 %. Therefore, power-off capability is important. The following experiments shown in Chapter 5, 6, 7, and 8, especially the cases that use memory mapping, assume these functions in the memory sub-system.

3.4.2 Estimation of the Impact of Memory Power-off

These experimental results provide estimations on power-off memory savings on top of measured system power and energy. Now, to estimate the portion it needs to clarify power reduction from offline memory varies to workloads or not. Figure 3.7 and Table 3.2 illustrate the potential power saving by power-off memory. Note that power reduction by the proposed hardware mechanism is realized when all guests' memory is consolidated to one NUMA domain. With the memory consolidation, the figure shows the shape of instantaneous power for both, the default and the case in which we have removed all DIMMs in one NUMA domain. The gap between two lines is consistently maintained. It shows the portion of reduced power coming from static power, which is agnostic to workload. This delta is taken for each test machine and used to subtract measured average power only when memory consolidation is configured. Similarly, measured energy is accordingly reduced using the estimated average power savings.

3.5 Conclusion

This chapter illustrate overall design of NAVAR adaptive system. The system comprises of three major components: the policy component, the adaptation mechanism, and the detection framework. Due to the discussion briefly about the policy in the last chapter, the detection framework and the adaptation mechanism are presented in this chapter. Along with this, the common experimental setups and the proposed hardware mechanism are also mentioned. The following chapters address the problems raised in one of the three resource demanding classes: under-subscription, full-subscription, and over-subscription. First, the next chapter preliminary discusses a special case of the under-subscription class.

Chapter 4

Under-subscription: vcore-mapping

The last chapter discusses the overall system design as well as the specific design for each component: policy, detection framework, and adaptation mechanisms. For the mechanisms, vcore mapping, memory mapping, and gang-scheduling were described. Possible policy design options were briefly introduced because policy design sis arguably specific to the condition of resource demand. Three classes of resource demands were also defined, namely under-subscription, full-subscription and over-subscription. This chapter and the next are focused on the under-subscription case. As under-subscription does not involve vcore-scheduling, vcore and memory mapping are the main concerns; however, this chapter focuses on vcore mapping with an affinity to memory mapping. We assume that the memory of all guests is allocated in one NUMA domain. The chapter at first begins with discussions of tradeoffs and the opportunities of the specific under-subscription case, the detection framework, policy and experimental results follow.

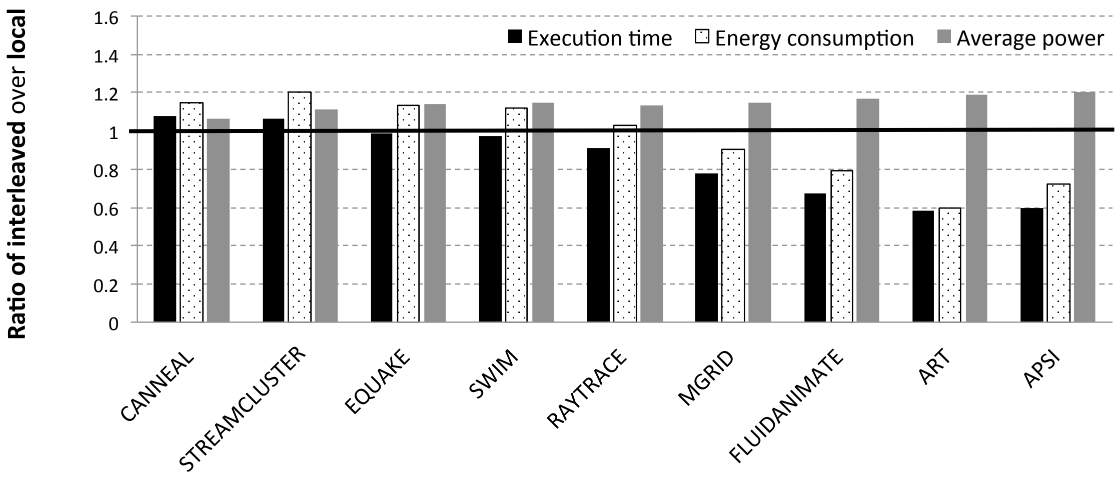


Figure 4.1: Comparing the *interleaved* and *local* vcore mappings for a range of benchmarks and objectives. Notice that the *local* mapping in this case refers to *consolidated* vcore mapping. There are considerable opportunities to trade off between performance, power, and energy. The tradeoffs are also clearly dependent on the workload.

4.1 Tradeoffs and Opportunities

In the real world, the three resource management problems may happen in a combined way. Moreover, it is quite challenging to comprehend the situation and tradeoffs between various mapping options. This chapter deals with a simple problem – **`vcore-mapping`**. An assumption here is that memory is mapped into one NUMA domain. The assumption may oversimplify the real problem, but let us narrows down to specific tradeoffs. Figure 4.1 shows ratios of *interleaved* over *local* (*consolidated*) vcore mapping for performance, power, and energy. Each benchmark has eight threads on top of a VM which has eight vcores. The metric for performance is execution time, makespan. Execution time varies up to 66%, where power and energy by 17% and 31% repeatedly. This shows great opportunities for an adaptive system.

The system is expected to maximize performance, minimize energy or minimize power per user demands. What the figure shows is, depending on the characteristic of workloads, it makes sense to take more resources at the cost of power. If computational threads in a guest have significant shared-memory communication, all vcores are better to be consolidated on the same socket. In a under-subscription case, the *local* vcore mapping also allows us to idle the other socket. However, if threads have negligible shared memory communication, the *local* vcore mapping will hurt performance. Off-chip delays and memory controller bandwidth are the factors. This latter type of workload benefits from memory bandwidth, but they are less sensitive to the delays in shared memory synchronization. The SPEC OMP and PARSEC benchmarks are mainly used in this chapter. Notice that a complete under-subscription case will be dealt in Chapter 5, along with different memory mapping strategies.

4.1.1 Memory Reference Behavior Affecting Performance

To better understand the tradeoffs appearing in Figure 4.1, the hardware monitor has been used to capture cache and memory traffic. Also, benchmarks shown in Figure 4.1 are running eight threads, the same as the number of vcores in their VM. The performance counter in the R410 testbed allows us to check cache coherence traffic such as the number of cache hits in modified cache blocks and whether invalidations come from the local or remote NUMA domain. Along with this, the detection framework, the hardware-assisted software monitor described in the previous chapter, is also used to check VMM-drived metrics such as accessed and written page rates. The two vcore mapping strategies, the *interleaved* and *local* vcore mappings, are compared, and workloads are measured with various degree of parallelism and compilation options. The measurements are intended to help us to understand variations driven by each application and it's implementation. Also, in the scope of this chapter, cache performance, not only the hit ratio, but also the amounts

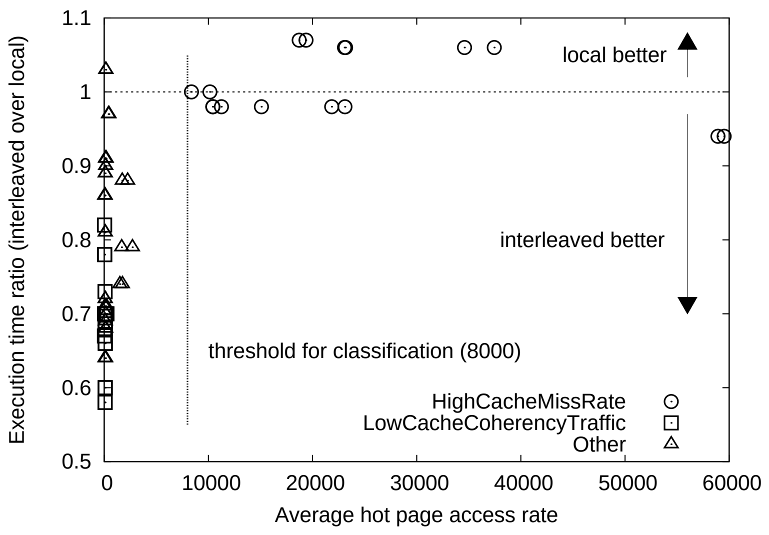
Benchmark	Class		
	0	1	2
ammp (SPEC)		✓	
apsi (SPEC)		✓	
art (SPEC)		✓	
blackscholes (PARSEC)			✓
bodytrack (gcc-pthread/icc-pthread) (PARSEC)			✓
bodytrack (gcc-omp/Intel-TBB) (PARSEC)		✓	
canneal (PARSEC)	✓		
equake (SPEC)	✓		
facesim (PARSEC)			✓
ferret (PARSEC)		✓	
fluidanimate (gcc-pthread/Intel-TBB) (PARSEC)		✓	
fluidanimate (icc-pthread) (PARSEC)			✓
fma-3d (SPEC)			✓
freqmine (PARSEC)			✓
galgel (SPEC)			✓
raytrace (PARSEC)			✓
streamcluster (PARSEC)	✓		
swaptions (PARSEC)			✓
swim (gcc) (SPEC)			✓
swim (icc) (SPEC)	✓		
mgrid (SPEC)			✓
wupwise (SPEC)		✓	

Table 4.1: Classifications of all of our benchmarks

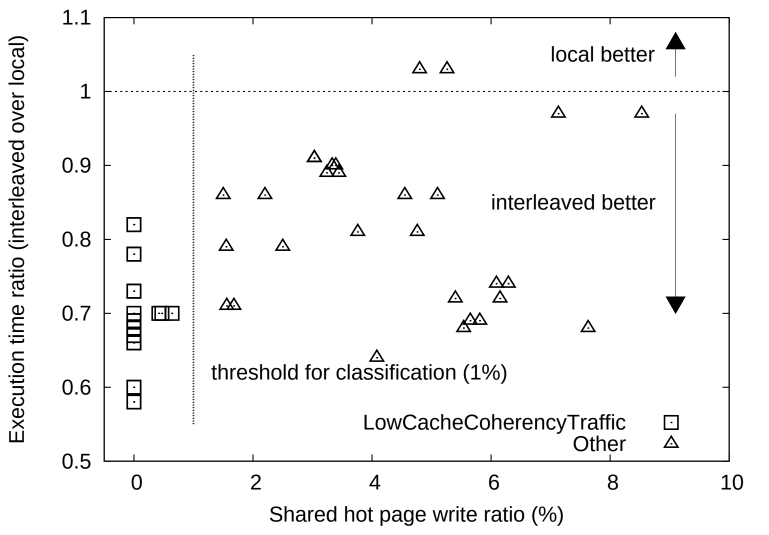
of coherence traffic across caches, are very sensitive factors. For this reason, variations in parallelism and compilers are as visible as some core characteristics such as working set size (WSS).

4.1.2 Classification

With all the measured metrics and various application setups, certain trends are captured as depicted in Figure 4.2. One point of the figure is uncovering underlying factors that result in different performance benefits between the two vcore mappings. The figure shows two



(a) Separating HighCacheMissRate from LowCacheCoherencyTraffic and Other



(b) Separating LowCacheCoherencyTraffic from Other

Figure 4.2: Classifying workloads by their memory traffic characteristics. Vertical axes indicate the performance ratio between interleaved and local execution, while the horizontal axis is the metric used for classification. Each point in a graph represents the measurement of the combination of a benchmark, a set of compilation options, and whether the measurement is per-memory-operation or per-store. HighCacheMissRate is distinguished by a high distinct page access rate over all the vcores. LowCacheCoherencyTraffic is distinguished by having only a small fraction of each vcore's writes going to pages written by other vcores. Other is the remaining class.

metrics, the overall page access rate and the fraction of a vcore's writes to cache blocks that overlap with the writes of other vcores. They are used to partition the three classes. The implication and classifying process of those classes is as follows.

First, when the page access rate, the rate at which distinct pages are either read or written, is high, the case is referred to as the HighCacheMissRate class. In this case, the contention is thought to be in the main memory system, and this is due to a large working set size. The *Local* mapping is preferred here as main memory-access time is the critical path. The *interleaved* mapping raises the cost of remote accesses. CANNEAL is an example of the HCM class.

Some workloads perform better in the *interleaved* mapping. As measured and compared using the shared page write ratio, some workloads with very small write ratio also perform better with the *interleaved* mapping. What the metric captures is the fraction of writes to pages, by a vcore, that are also written by another vcore. If the fraction is very small, it is in LowCacheCoherencyTraffic class. If any workload that is not either of the two classes, it is classified as Other class. An example for LowCacheCoherencyTraffic is APSI and MGRID is an example of the Other class. The whole list of classifications for our benchmarks is in Figure 4.1. The classification is intended to understand factors that bring the performance tradeoffs. If a class affects the performance difference sharply, it can be integrated into a runtime policy with a decision tree. However, some benchmarks, particularly in Ohter class, are not predictable with the classification. More details on policy design will be discussed in Section 4.3.

4.1.3 Tradeoffs in Power and Energy

Optimizing power is relatively straightforward. Although the best way to minimize power is by consolidating all vcores, here eight vcores, they are not overlapped in one hardware thread according to the assumption of under-subscription. Therefore, consolidating all vcores in one socket is the best mapping policy here. Now, for the energy objective, two aspects that should be contemplated are power and performance. Power is rather predictable, as our workloads are fully utilizing CPU resources. Many articles [45, 97, 120, 68] describe their experiments that show that CPU power is linear with the CPU utilization ratio. The utilization ratio is a ratio between active and sleep times in a short duration. As the selected workloads are highly computational, they mostly demand computational power. Sometimes, some workloads produce high memory traffic which may cause some delays, but prefetching and out-of-order executions keeps cores almost always running. With a power model, power difference in the two vcore mappings can be drawn. The energy tradeoff then is a matter of whether the performance benefit from the *interleaved* vcore mapping is more than the cost of increased power. Workloads in some classes exhibit energy tradeoffs, as their execution time differences between the two mappings are very predictable. However, a fine-grained model is still needed to predict a precise execution time, particularly for the workloads where execution time is hard to predict only with the classifications.

4.2 Extended Detection Framework

The detection framework (the hardware-assisted software monitor) needs to be extended for the adaptive system. The reason is tightly coupled with the needed metrics. Figure 4.3 illustrates the extension, namely the two phases of probing. Recall the variables described in the hardware-assisted software monitor design. During a probe, hardware raises an interrupt after a certain amount of events, either a number of memory operations or write operations. Given each interrupt, the shadow or nested page tables are scanned in each vcore. Two intervals are used to capture both the per-op and per-write metrics. Note that all

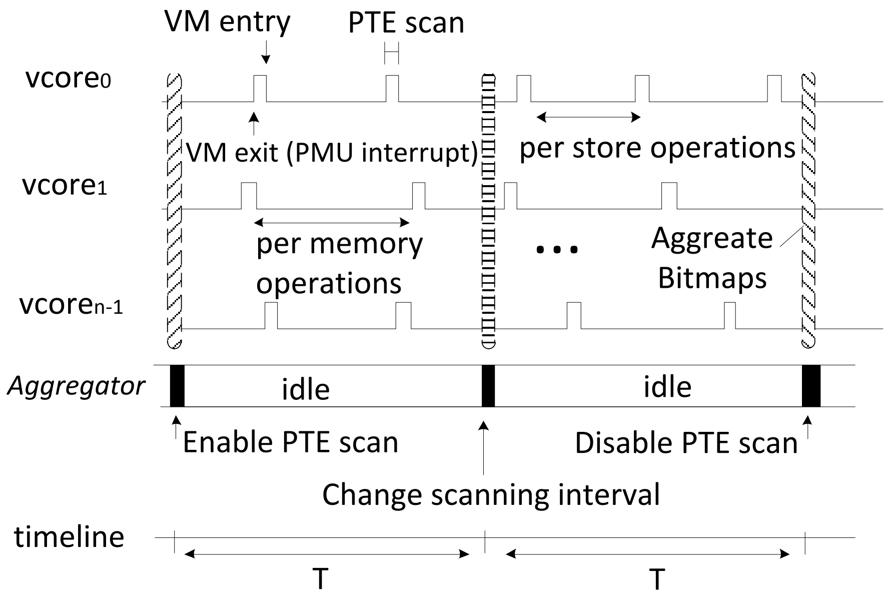


Figure 4.3: Illustration of probing on a timeline. In a probe, each virtual core scans its page table independently. The scanning interval is in units of memory operations (both stores and loads) or or store operation. At the end of the probing interval, information on accessed or written pages is collected from all virtual cores and the metrics are computed from it.

the bitmaps created with page-table-scans are collected using AND operations to capture hot pages. Statistically, these metrics have more correlation with the objective metric, makespan. The sensitivity to the sampling interval will be discussed the next chapter. The following is a list of all metrics collected.

- The average accessed hot page rate per memory operation, $r_{\cap am}$: Intuitively, this captures the offered memory system load from all of the virtual cores.
- The average written hot page rate per memory operation, $r_{\cap wm}$: Intuitively, this captures how much of that load is due to writes.
- The shared accessed hot page ratio per memory operation, $s_{\cap am}$: Intuitively, this

captures the fraction of page accesses from any virtual core that are also accessed from another virtual core—the degree of read or write sharing.

- The shared written hot page ratio per memory operation, $s_{\cap wm}$: Intuitively, this captures the fraction of page writes from any virtual core that are also matched with writes to the same page from another virtual core—the degree of write sharing.

- The average access hot page rate per store operation, $r_{\cap as}$.

- The average written hot page rate per store operation, $r_{\cap ws}$.

- The shared access hot page ratio per store operation, $s_{\cap as}$.

- The shared written hot page ratio per store operation, $s_{\cap ws}$.

4.2.1 Algorithm

We now describe the extension to the hardware-assisted software monitor. To get a clear idea of the extension and the detection framework as well, the following are the formulations for the extended detection framework. Let $accessed\_per\_mem_i$, $written\_per\_mem_i$ (for $r_{\cap am}, r_{\cap wm}, s_{\cap am}$ and $s_{\cap wm}$), $accessed\_per\_store_i$, $written\_per\_store_i$ (for $r_{\cap as}, r_{\cap ws}, s_{\cap as}$ and $s_{\cap ws}$), $accessed_i$ and $written_i$ be the bit vectors representing the sets of pages accessed and written on virtual core i. The bit vectors contain as many bits as there are pages in the physical address space of the guest that is backed with physical memory. Let n be the number of vcores, m be the number of pages, and T be the real time interval between aggregations. The following algorithm implements the core of the Probing routine of Section 4.3.3. The elements of a single probe operation are condensed into five events. The Probing routine initiates the process by invoking the Init function:

Init(aggregator): [Invoked at startup on aggregator]

```
  SetTimer($T$, SetAggregate)
  $Phase = 0$
  **for all** vcores $i$ **do**
    $EnableScan_i = 1$
    Force vcore $i$ to run InitVcore($i$)
  **end for**
InitVcore($i$): [Invoked at on vcore $i$]
  $accessed_i = \{k : 0 \ldots m - 1\}$
  $written_i = \{k : 0 \ldots m - 1\}$
  Set PMU exception for number of memory operations to trigger Scan($i$).
ReinitVcore($i$): [Invoked at on vcore $i$]
  $accessed\_per\_mem_i = accessed_i$
  $written\_per\_mem_i = written_i$
  $accessed_i = \{k : 0 \ldots m - 1\}$
  $written_i = \{k : 0 \ldots m - 1\}$
  Set PMU exception for number of store operations to trigger Scan($i$).
Scan($i$): [invoked when PMU exception occurs]
  **if** $EnableScan_i = 1$ **then**
    **for all** present shadow (or nested) PTEs on vcore $i$ **do**
      $k$ = DeriveGuestPhysicalPageNumberFrom(PTE)
      $curacc_i = \emptyset$
      $curwrit_i = \emptyset$
      **if** PTE.accessed **then**
        $curacc_i = curacc_i \cup \{k\}$
      **end if**
```

```
        if PTE.dirty then
            $curwrit_i = curwrit_i \cup \{k\}$
        end if
        PTE.accessed=0
        PTE.dirty=0
    end for
    $accessed_i = accessed_i \cap curacc_i$
    $written_i = written_i \cap curwrit_i$
end if
```

The PMU is set to raise an exception for a number of memory operations (or write operations) to trigger Scan(i). Scan(i) runs multiple times (at least twice) during a probe, depending on the memory-access rate. The purpose of the somewhat confusing intersection operations over these runs is to filter out pages that are infrequently written or read; in other words, the operation is selecting hot pages. At the end of a probe, it collects the set of pages that are consistently written or accessed during the whole probe interval.

```
SetAggregate(aggregator): [invoked when $T$ expires on aggregator]
  if $Phase = 0$ then
    for all vcores $i$ do
      $EnableScan_i = 0$
      Force vcore $i$ to run ReInitVcore($i$)
      $EnableScan_i = 1$
    end for
    $Phase = 1$
    SetTimer($T$, SetAggregate);
  else
```

```
    Aggregate(aggregator)
  end if

Aggregate(aggregator)
  for all vcores $i$ do
    $EnableScan_i = 0$
    $accessed\_per\_store_i = accessed_i$
    $written\_per\_store_i = written_i$
  end for
  $r_{\cap am} = \frac{1}{n}\sum_{i=0}^{n-1}|accessed\_per\_mem_i|$
  $r_{\cap as} = \frac{1}{n}\sum_{i=0}^{n-1}|accessed\_per\_store_i|$
  $r_{\cap wm} = \frac{1}{n}\sum_{i=0}^{n-1}|written\_per\_mem_i|$
  $r_{\cap ws} = \frac{1}{n}\sum_{i=0}^{n-1}|written\_per\_store_i|$
  $s_{\cap am} = \frac{1}{r_{am}}\frac{2}{(n-1)n}\sum_{j=0}^{n-2}\sum_{k=j+1}^{n-1}|accessed\_per\_mem_j \cap accessed\_per\_mem_k|$
  $s_{\cap as} = \frac{1}{r_{\cap as}}\frac{2}{(n-1)n}\sum_{j=0}^{n-2}\sum_{k=j+1}^{n-1}|accessed\_per\_store_j \cap accessed\_per\_store_k|$
  $s_{\cap wm} = \frac{1}{r_{\cap wm}}\frac{2}{(n-1)n}\sum_{j=0}^{n-2}\sum_{k=j+1}^{n-1}|written\_per\_mem_j \cap written\_per\_mem_k|$
  $s_{\cap ws} = \frac{1}{r_{\cap ws}}\frac{2}{(n-1)n}\sum_{j=0}^{n-2}\sum_{k=j+1}^{n-1}|written\_per\_store_j \cap written\_per\_store_k|$
```

4.3 Policy and System Design

Workloads are classified according from measured metrics by using the extended hardware-assisted software monitor. The classification is not sufficient to predict performance and energy. This section presents how the policy design and performance, power, and energy models fit in the policy component. Pseudo codes are also shown to describe the algorithm used for adaptive resource management.

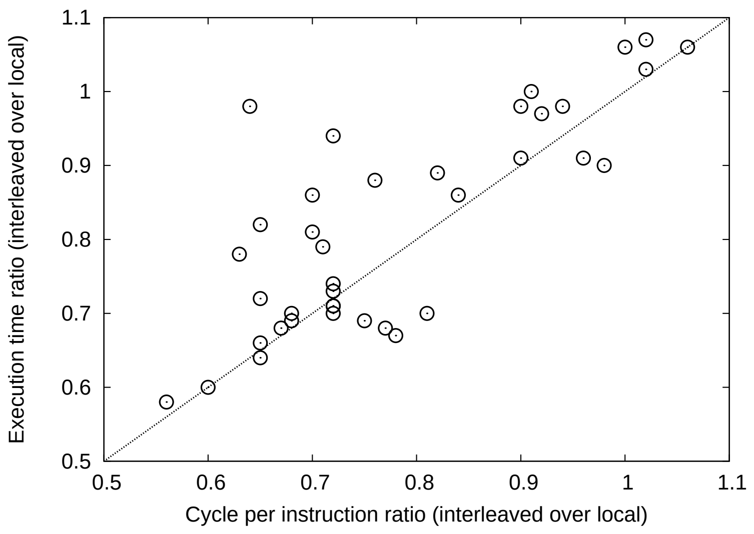


Figure 4.4: CPI is weakly predictive of the performance gain likely from moving to an interleaved model. Classification followed by linear regression provides much more predictability.

4.3.1 Approach

For an adaptive system, a brute-force approach can be at first considered. However, it is simply dropped in the scope of this chapter because a metric for a feedback loop is not clear. As mentioned earlier, the brute-force approach includes a feedback loop that contains a measured value. The measured value should be strongly coupled with an objective metric, execution time. Power, again, is measurable or very predictable. CPI (Cycle Per Instruction) is a very popular metric as a performance indicator. However, CPI has recently been reported to be inaccurate for predicting execution time particularly for concurrent and parallel workloads [48, 8]. As shown in Figure 4.4, CPI has a weak correlation with makespan. Nevertheless, the brute-force approach is taken with CPI as a feedback metric in Chapter 5, the next chapter.

In the current chapter, the adaptive system is designed to use model-based predictions. In terms of modeling, decision-tree base modeling is based on the classification of workloads is described. The classification is good for pointing out some extreme cases and their characteristics, but the classification based approach is insufficient for telling which benchmarks performs better with the *interleaved* vcore mapping, particularly for none-extreme cases. A different model is needed to predict the objective metric with better accuracy. Linear-regression based modeling is one viable option. The approach demands little computational power and is also suitable for incorporating multiple metrics captured by the detection framework. Even if this linear-regression approach is effective, the classification approach is still useful. Therefore, the two models need to be combined. My overall approach is to first classify workloads and, in each class, predict execution time through a linear regression model.

4.3.2 Models

Performance model The performance model has two steps: a decision-tree (classifier) and a first-order formula to predict execution time. In fact, the first-order (linear) formula does not necessarily estimate maskespan well. Instead, we predict the execution time ratio of the two vcore mappings.

Decision tree A decision is composed of two nodes as three classes are classified. The first node separates HighCacheMissRate from the rest of classes. The partition is made based on the average of $r_{\cap am}$ and $r_{\cap as}$. The idea is to filter out workloads that have a large working set size. This type of workload stresses the cache, particularly last level cache. The threshold value is derived from this equation:

$$threshold_{class0} = \frac{LastLevelCacheSize}{((PG\cdot PGUtil)\cdot (NTh - SR\cdot (NTh-1)))}$$

where PG is the page size, $PGUtil$ is the average number of the memory operations per page, NTh is the number of threads, and SR is the sharing ratio of accessed pages (that is $s_{\cap as}$ or $s_{\cap am}$). For the R410 test system, the threshold for the first node is calculated to be 8000. Second node separates LowCacheCoherencyTraffic from Other based on the average of $s_{\cap wm}$ and $s_{\cap ws}$. The threshold value is set to 1 %, to conservatively pick very low cache coherency-demanding workloads.

First-order formula Given the decision-tree, the execution time ratio is estimated through a first-order formula, the coefficient of which is determined by a training set. One limit of the work in this chapter is a lack of an evaluation set for the trained models. Also, two formulas are driven by two training sets. One set comes from the Other class and another set is from two other classes, HighCacheMissRate and LowCacheCoherencyTraffic. Even with the formula, sometimes when the ratio is near 1.0, the fitting error becomes large. For those cases, CPI is taken into account to compensate for the error, but CPI is not included in performance models. Due to the inaccuracy of CPI shown in Figure 4.4, this metric is sparsely used. Instead, the policy will treat them, the predicted execution ratio and the measured CPI, as individual metrics, and a decision is then made through the defined algorithm. Here is the formulation of performance model.

```
**if** $(r_{\cap am} + r_{\cap as}) > threshold_{HighCacheMissRate}$ **or** $(s_{\cap wm} + s_{\cap ws}) < threshold_{LastLevelCacheSize}$
**then**
    **if** current mapping is local **then**
        $ratio \leftarrow C01l_0 + [C01l_1, ..., C01l_8] \cdot [r_{\cap am}, r_{\cap wm}, ... s_{\cap as}, s_{\cap ws}]$
    **else**
        $ratio \leftarrow C01i_0 + [C01i_1, ..., C01i_8] \cdot [r_{\cap am}, r_{\cap wm}, ... s_{\cap as}, s_{\cap ws}]$
    **end if**
**else**
```

Num. of core w. 1+ threads (p)	Num. of core w. 2 threads (l)	Power (watts)
1	0	112.04
2	0	123.23
3	0	131.32
4	0	138.37
2	0	120.87
4	0	142.52
6	0	156.49
8	0	173.42
1	1	114.07
2	2	126.24
3	3	135.68
4	4	145.35
4	1	141.35
4	2	142.49
4	3	143.83

Table 4.2: System power consumption with varying numbers of threads and affinity. Each thread exhibits full core utilization, so the core remains in the P-state. A linear regression models this data as $104.63 + 8.69\cdot p + 1.62\cdot l$.

```
        **if** current mapping is local **then**
            $ratio \leftarrow C2l_0 + [C2l_1, ..., C2l_8] \cdot [r_{\cap am}, r_{\cap wm}, ... \; s_{\cap as}, s_{\cap ws}]$
        **else**
            $ratio \leftarrow C2i_0 + [C2i_1, ..., C2i_8] \cdot [r_{\cap am}, r_{\cap wm}, ... \; s_{\cap as}, s_{w\cap s}]$
        **end if**
    **end if**
    **return** $ratio$
```

The constant vectors [$C01l_0$, ..., $C01l_8$], [$C01i_0$, ..., $C01i_8$] , [$C2l_0$, ..., $C2l_8$] , and [$C2i_0$, ..., $C2i_8$] comprise the linear models (the coefficient vectors) that are from offline training. The predictions are formed by their dot product with the currently probed metrics. Notice that a different linear model is used depending on the class of the workload.

Power model Linear regression is also used to create a power model. The measured power during the execution of benchmarks on the test hardware, the R410, has been used to build the model. As shown in Table 4.2, the power for different core utilization scenarios behaves fairly linearly. This observation is well established and used in previous works [45, 97, 120, 68] where power predictions are accurate for CPU-dominated workloads. Based on the measurements in Table 4.2, a linear regression forms a power model whose inputs include the number of active cores and the number of threads per core. For the testbed machine, the coefficients of the model are also shown in Table 4.2. To illustrate how the these coefficients are used, two basic examples are given:

- one active core with one thread consumes 8.69 Watts (P_1)
- one active core with two threads consumes 10.31 Watts (P_2)

Note that the instantaneous power can vary widely over time. This is due to the varying behavior of a CPU core's power state which is driven by scheduling behaviors. In order to determine the average power usage, it is necessary to measure the amount of time that a core spends in both active and idle states. Fortunately, there is one easily accessible metric provided by the host OS: the CPU utilization. In the test system, the R410, the utilization of each hardware thread in a core matches with the vcore utilitzation, $vcoreUtil$. Using the utilization measurements, it becomes possible to determine the average time that a core spends in idle or active power states. Consider two active vcores, i, and j, with utilization $vcoreUtil_i$ and $vcoreUtil_j$. Notice the test machine has two hardware threads (two logical cores) per one physical core. When two vcores run at the same time, the power is expected to be P_2; otherwise, when two vcores run exclusively, the expected power is P_1. The utilization is then used to estimate the amount of time that the two vcores execute at the same time or exclusively in the following way:

$$\begin{aligned} power_{local} = & P_1 \cdot (max(vcoreUtil_i, vcoreUtil_j) - min(vcoreUtil_i, vcoreUtil_j)) \\ & + P_2 \cdot min(vcoreUtil_i, vcoreUtil_j) \end{aligned} \tag{4.1}$$

$$power_{inter} = P_1 \cdot vcoreUtil_i \tag{4.2}$$

The ratio $power_{interleaved}/power_{local}$ is the final result of the model. Here is a complete version as estimating multiple cores' power.

```
$P_{local} \leftarrow \sum_{i=0}^{max(core)} \sum_{j=0}^{max(vcore)-1} \sum_{k=j+1}^{max(vcore)} L_{j\to i} \cdot L_{k\to i} \cdot (P_1 \cdot (max((vcoreUtil)_j, (vcoreUtil)_k) - min((vcoreUtil)_j, (vcoreUtil)_k)) + P_2 \cdot min((vcoreUtil)_j, (vcoreUtil)_k))$
where, $L_{i\to j}$ = 1 if $vcore_i$ is mapped to $core_j$, otherwise 0 as configured in local mapping
$P_{interleaved} \leftarrow \sum_{i=0}^{max(core)} \sum_{j=0}^{max(vcore)} I_{j\to i} \cdot P_1 \cdot (vcoreUtil)_j$
where, $I_{i\to j}$ = 1 if $vcore_i$ is mapped to $core_j$, otherwise 0 in interleaved mapping
**return** $P_{interleaved}/P_{local}$
```

Energy model The energy model is derived from both the power and performance models. The model estimates the ratio of energy consumption between two mappings, *interleaved* and *local*, by multiplying the predicted execution time ratio, $ratio_{perf}$, with the predicted average power usage, $ratio_{power}$.

$$ratio_{energy} = ratio_{perf} \cdot ratio_{power}$$

.

4.3.3 Algorithm

The implementation of the online prediction and adaptation algorithm contains five major functions in the main loop. Updates on the measurements (Probing) and decisions (Voting) are made periodically. If the current vcore mapping ($Mapping_{cur}$) needs to be changed, ReMapping is called. If a remapping happens, the system pauses until the system overhead falls below the threshold, $threshold_{ovrhd}$. The duration for which the system pauses is called the Interim state. Note that $metrics$ is a vector containing the 8 metrics defined in Section 4.2. Additionally, the per-vcore cpi value is tracked, as well as its utilization, $vcoreUtil_i$. The overhead for a vcore is computed from a summation of page table scanning time ($ovrhd_{probe}$) and vcore remapping time ($ovrhd_{remap}$).

Pseudocode Main Loop periodically finds the correct mapping. Intuitively, it periodically probes the metrics described in Section 4.2, and then executes a voting procedure based on them. The voting procedure indicates the preferable mapping and the performance that is likely to result, based on the current objective. Additionally, it reports the confidence in its prediction. If the confidence is high, the new mapping is immediately committed; otherwise, is is temporarily switched to probe its behavior and the voting procedure is allowed to make a new prediction. If the two predictions agree, the mapping is then committed; while if they disagree, a tie-breaker is invoked. The code also tracks the overheads of its various components, and these overhead measurements are used to control the rate of execution of the loop. The user determines the maximum overhead that is tolerated. The Main Loop has the following pseudocode:

```
$ovrhd_{probe} \leftarrow 0$
$ovrhd_{remap} \leftarrow 0$
while $1$ do
```

```
        $ovrhd_{probe} \leftarrow$ Probing($metrics$)
        ($confidence$, $vote_1$, $cpi_1$)$\leftarrow$ Voting($metrics$)
        **if** $confidence$ is high **then**
            **if** $vote_1 \neq Mapping_{cur}$ **then**
                $ovrhd_{remap} \leftarrow$ ReMapping()
            **end if**
        **else**
            $ovrhd_{remap} \leftarrow$ ReMapping()
            sleep as long as a half of probing time
            $ovrhd_{probe} \leftarrow$ Probing($metrics$) + $ovrhd_{probe}$
            ($confidence$, $vote_2$, $cpi_2$)$\leftarrow$ Voting($metrics$)
            **if** $vote_1$ = $vote_2$ **then**
                **if** $vote_1 \neq Mapping_{cur}$ **then**
                    $ovrhd_{remap} \leftarrow$ ReMapping() + $ovrhd_{remap}$
                **end if**
            **else**
                $vote_3 \leftarrow$ finalVoting($cpi_1$, $cpi_2$)
                **if** $vote_3 \neq Mapping_{cur}$ **then**
                    $ovrhd_{remap} \leftarrow$ ReMapping() + $ovrhd_{remap}$
                **end if**
            **end if**
        **end if**
        Interim($ovrhd_{probe}$, $ovrhd_{remap}$)
    **end while**
```

Voting($metrics$) is called to make an initial prediction of the best mapping. If the initial vote had low confidence, the predicted mapping is temporarily switched to evaluate it. This voting procedure heavily depends on the predictions made by the models. Since the mapping strategy has two options, each model reports the ratio of the two estimated values in two mappings. The power model, for example, estimates the ratio of the power of the *interleaved* mapping over that of the *local* mapping. Thus, the more the ratio diverges from 1, the more confident it is.

```
if $objective$ is performance then
    $ratio \leftarrow$ PerformanceModel($metrics$)
else if $objective$ is energy then
    $ratio \leftarrow$ EnergyModel($metrics$)
else if $objective$ is power then
    $ratio \leftarrow$ PowerModel($metrics$)
end if
if $ratio > 1$ then
    $vote \leftarrow$ local mapping
else
    $vote \leftarrow$ interleaved mapping
end if
if $ratio$ is within unconfident intervals then
    $confidence \leftarrow$ low
else
    $confidence \leftarrow$ high
end if
get $cpi$ from $metrics$
return ($confidence$, $vote$, $cpi$)
```

PerformanceModel($metrics$) classifies the workload and then selects the correct performance model to compute the performance ratio of the *interleaved* to the *local* mapping.

```
if $(r_{\cap am}$ + $r_{\cap as}) > threshold_{class0}$ or $(s_{\cap wm} + s_{\cap ws}) < threshold_{class1}$ then
    if current mapping is local then
        $ratio \leftarrow C01l_0$ + $[C01l_1, ..., C01l_8] \cdot [r_{\cap am}, r_{\cap wm}, ...\ s_{\cap as}, s_{\cap ws}]$
    else
        $ratio \leftarrow C01i_0$ + $[C01i_1, ..., C01i_8] \cdot [r_{\cap am}, r_{\cap wm}, ...\ s_{\cap as}, s_{\cap ws}]$
    end if
else
    if current mapping is local then
        $ratio \leftarrow C2l_0$ + $[C2l_1, ..., C2l_8] \cdot [r_{\cap am}, r_{\cap wm}, ...\ s_{\cap as}, s_{\cap ws}]$
    else
        $ratio \leftarrow C2i_0$ + $[C2i_1, ..., C2i_8] \cdot [r_{\cap am}, r_{\cap wm}, ...\ s_{\cap as}, s_{\cap ws}]$
    end if
end if
return $ratio$
```

In the above, the constant vectors $[C01l_0, ..., C01l_8]$, $[C01i_0, ..., C01i_8$], $[C2l_0, ..., C2l_8$] , and $[C2i_0, ..., C2i_8]$ comprise the linear models (the coefficient vectors) described in Section 4.3.2. The predictions are formed by their dot product with the currently probed metrics. Notice that a different linear model is used depending on the class of the workload.

PowerModel($metrics$) estimates CPU power in the two mappings, and returns their ratio.

$P_{local} \leftarrow \sum_{i=0}^{max(core)} \sum_{j=0}^{max(vcore)-1} \sum_{k=j+1}^{max(vcore)} L_{j\rightarrow i} \cdot L_{k\rightarrow i} \cdot (P_1 \cdot (max((vcoreUtil)_j, (vcoreUtil)_k)$ - $min((vcoreUtil)_j, (vcoreUtil)_k)) + P_2 \cdot min((vcoreUtil)_j, (vcoreUtil)_k))$

```
    where, $L_{i \to j}$ = 1 if $vcore_i$ is mapped to $core_j$, otherwise 0 as configured in local mapping
    $P_{interleaved} \leftarrow \sum_{i=0}^{max(core)} \sum_{j=0}^{max(vcore)} I_{j \to i} \cdot P_1 \cdot (vcoreUtil)_j$
    where, $I_{i \to j}$ = 1 if $vcore_i$ is mapped to $core_j$, otherwise 0 in interleaved mapping
    **return** $\frac{P_{interleaved}}{P_{local}}$
```

This pseudocode incorporates Equations 4.1 and 4.2. The description of these equations is in Section 4.3.2.

EnergyModel($metrics$) is straightforward:

```
    $ratio_{power} \leftarrow$ PowerModel($metrics$)
    $ratio_{perf} \leftarrow$ PerformanceModel($metrics$)
    **return** $ratio_{power} \cdot ratio_{perf}$
```

finalVoting(cpi_1, cpi_2) comprises the tie-breaker in case the two initial votes contradict each other.

```
    **if** $cpi_1 < cpi_2$ **then**
        $vote \leftarrow$ previous mapping which brings $cpi_1$
    **else**
        $vote \leftarrow$ current mapping which brings $cpi_2$
    **end if**
    **return** $vote$
```

Probing($metrics$) collects the performance metrics described in Section 4.2:

```
    $tsc_{start} \leftarrow$ *readtsc*
    Call Init(aggregator) from Section 4.2.1 to initiate a two-step round of probing to collect
    the eight metrics described in 4.2
    then, update $metrics$ with values in 8 metrics, $cpi$, $vcoreUtil$
    $tsc_{end} \leftarrow$ *readtsc*
```

```
return $(tsc_{end} - tsc_{start})$
```

Although it is not shown here, it is important to note that a moving average or exponential average of the metrics could be taken to reduce burstiness of the measurements.

ReMapping() implements a mapping change and tracks the overhead of doing so:

```
$tsc_{start} \leftarrow$ readtsc
change vcore/core mapping
update $Mapping_{cur}$
$tsc_{end} \leftarrow$ readtsc
return $weight \cdot (tsc_{end} - tsc_{start})$
```

Interim($ovrhd_{probe}$,$ovrhd_{remap}$) controls the overhead of the system by comparing its measured overhead with a threshold. If the threshold is exceeded, the system sleeps for a time:

```
if $ovrhd_{probe} = 0$ and $ovrhd_{remap} = 0$ then
    $tsc_{prev} \leftarrow$ readtsc
    return
else
    $tsc_{cur} \leftarrow$ readtsc
    $tsc \leftarrow tsc_{cur} - tsc_{prev}$
    $ovrhd \leftarrow ovrhd_{probe} + ovrhd_{remap}$
    while $ovrhd / tsc > threshold_{ovrhd}$ do
        sleep for $window_{interim}$
        $tsc_{cur} \leftarrow$ readtsc
        $tsc \leftarrow tsc_{cur} - tsc_{prev}$
    end while
    $tsc_{prev} \leftarrow tsc_{cur}$
    $ovrhd_{probe} \leftarrow 0$
    $ovrhd_{remap} \leftarrow 0$
    **return**
  **end if**
```

In the above, tsc refers to the cycle counter.

4.4 Experimental Results

This section illustrates overall experimental results on nine selected benchmarks. Along with the performance of the adaptive system, the accuracy of the models and overheads are thoroughly discussed.

4.4.1 Model Predictions

As described in Section 4.3.2, the performance of the models that predict performance gains and power gains is of critical importance both directly for the performance and power objectives, and indirectly for the energy objective, because energy is power$\times$time. The predictive power of the models is based on their performance with test sets. For the performance models, the R^2 ranges from 0.76 to 0.93 when measurements are made with the *local* configuration, and 0.70 to 0.91 when the measurements are made in the *interleaved* configuration. The power model achieves an R^2 of almost 1 in both cases.

4.4.2 System Performance

Figure 4.5, 4.6, and 4.7 show the performance of the adaptive system, and can be compared directly with the opportunities shown in Figure 4.1. The system is able to choose the best of the *interleaved* and *local* mappings for each workload and each optimization goal.

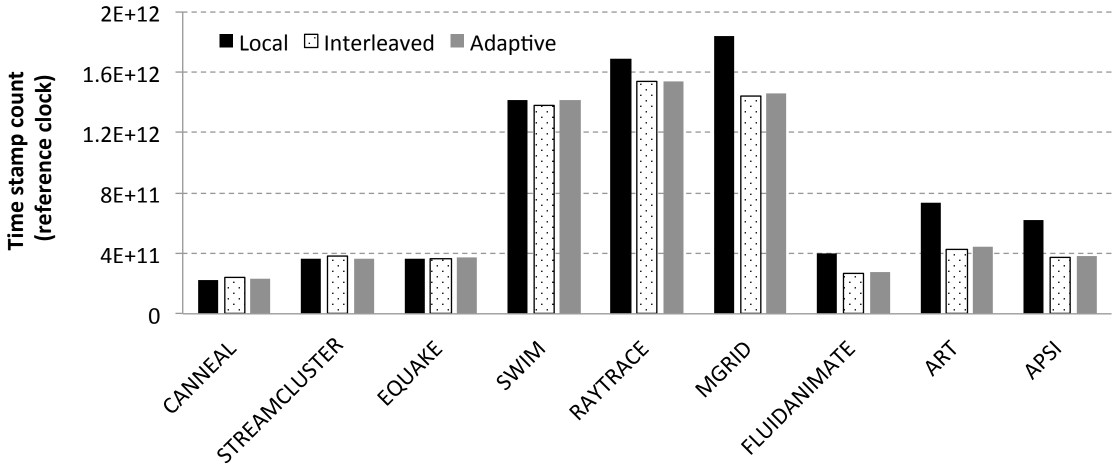


Figure 4.5: The adaptive system results for performance. The adaptive system can dynamically and automatically select a mapping that optimizes for the goal.

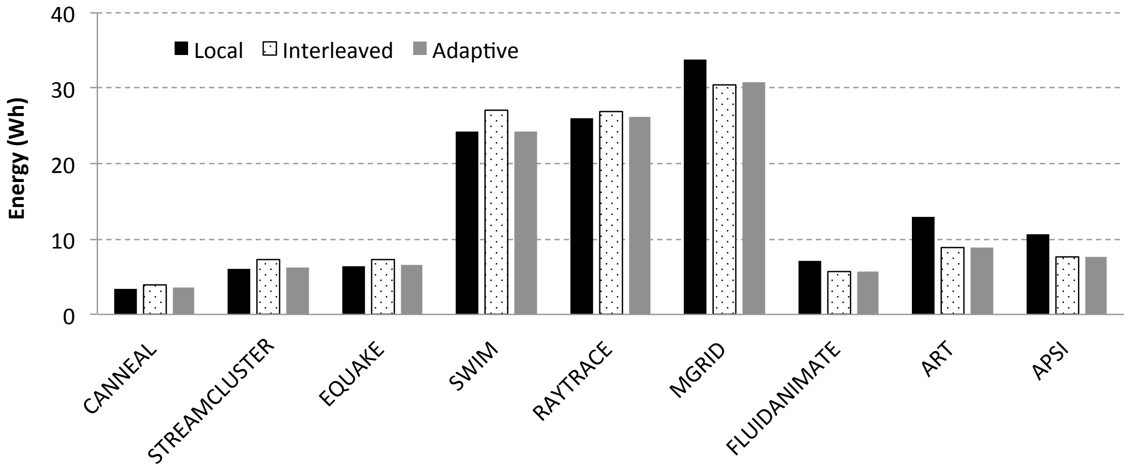


Figure 4.6: Performance of the adaptive system for energy objective.

It is also well maintained for the number of times that the vcores are remapped during the execution of each of the benchmarks and each of the optimization goals. In some cases, no remappings are done because the original mapping was the correct one. The default mapping is selected to be the *local* mapping. Remapping occurs in most cases infrequently and rarely. The RAYTRACE, SWIM, and MGRID benchmarks run for >10

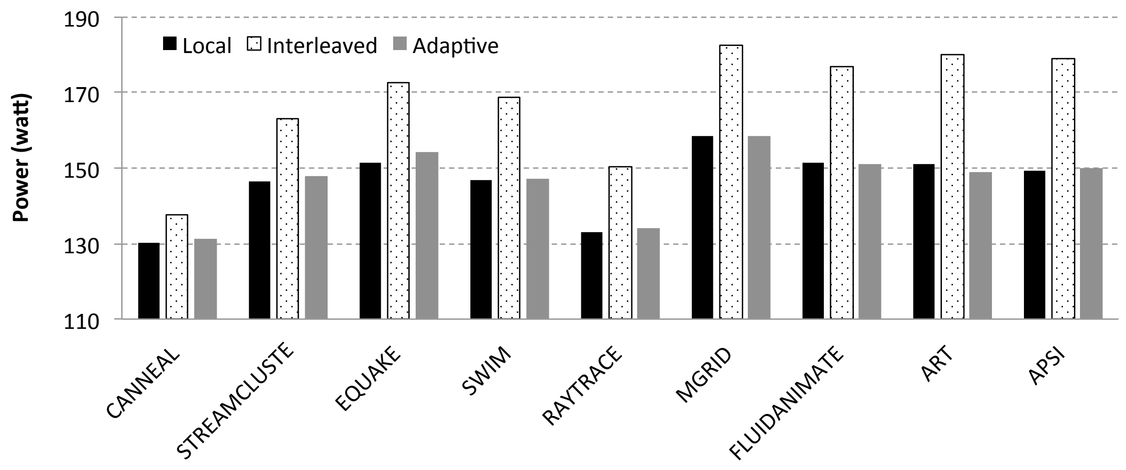


Figure 4.7: Performance of the adaptive system for power objective.

minutes, while the others run for 3-5 minutes. In these intervals, the common case is 0–2 remappings. RAYTRACE with an energy goal exhibits the largest number of remappings, eight remappings.

4.5 Conclusion

In this chapter, one solution is demonstrated that focus on the tradeoffs in vcore mappings. The tradeoffs are among the performance, power, and energy objectives, and our adaptive system is able to choose one of two vcore mappings: *interleaved* and *local* mappings. Note that the *local* mapping is the same as the *consolidated* mapping which will be used in the following chapters. The system incorporates sophisticated performance and power models built on top of offline-used analysis and training. Classification and linear regression are the methodologies to create them. The result of focusing on vcore-level tradeoffs, performance is more influenced by finer-grained concerns like cache performance and workload characteristics. The detection scheme focuses on the page-granularity. For the next chapter, we expand the scope of problem to include page mapping (memory mapping).

Chapter 5

Under-subscription

In the last chapter, preliminary studies were discussed that focus on vcore mapping tradeoff, relative to performance, power, and energy. An adaptive system was designed and evaluated to address the tradeoffs. However, it is not complete even for the under-subscription case. The assumption of memory affinity was made, which is unrealistic. Therefore, this chapter brings the configurability of memory mapping (page mapping) into the scope of the under-subscription problem. The chapter is comprised of four major parts, preliminary studies on tradeoffs, the detection framework, modeling and adaptive processes, and experimental results. This structure is repeated for the following chapters that address the full-subscription and over-subscription cases.

5.1 Tradeoffs and Opportunities

In the previous chapter, the situation was narrowed down to only vcore mappings. Now, this chapter includes memory mapping, so, the mapping strategy, in a coarse-grained level, includes *interleaved* and *consolidated* mapping for both vcore and memory. Additional attempts to conduct fine-grained mappings are discussed in Section 5.5. Having a memory affinity, in the fixed *consolidated* memory mapping shown in Chapter 4 tradeoffs depend on the degree of memory traffic and the levels of the memory hierarchy, whether it is last-level cache (LLC) intensive (HighCacheMissRate) or contains very low cache coherency traffic (LowCacheCoherencyTraffic). Although some workloads seem to have very mixed types of memory traffic (Other), specific cases are still distinguishable. When memory-access traffic is heavy beyond LLC due to a Large Working Set Size (LWSS), the *consolidated* (*local*) mapping is preferred; on the contrary, the *interleaved* vcore mapping is chosen, when the degree of cache coherence, inter-cache traffic, is very low with no LWSS. The following sections include an introduction to the two dimensional configuration space, preliminary performance results, and some discussion of the factors that bring tradeoffs.

5.1.1 Problem in Two-Dimensional Configuration Space

Four coarse-grained mapping cases are possible since we are considering both vcore and memory mappings. In short, the four mapping cases are:

- CCMC: virtual cores *consolidated* and memory *consolidated*
- CIMC: virtual cores *interleaved* and memory *consolidated*
- CCMI: virtual cores *consolidated* and memory *interleaved*
- CIMI: virtual cores *interleaved* and memory *interleaved*

Differences from the previous problem When HighCacheMissRate is the situation, it is not always desirable to consolidate vcores under the *interleaved* memory mapping. Two factors, bandwidth and memory-access distance, affect this. Memory bandwidth is limited when all of the guest pages are consolidated in one NUMA domain, regardless of the vcore mapping. Changing the vcore mapping, either to *interleaved* or *consolidated*, affects the load of the memory controllers in the NUMA domains. Therefore, the memory affinity makes the factor of memory-access distance more visible. Now, memory mapping is assumed to be reconfigurable, so the memory consolidation option limits resource provisioning, because it allows only a half or less of the available memory bandwidth to be used. Also, vcore consolidation, called *local* mapping earlier, no longer guarantees access to local memory. Therefore, we use the term *consolidated* vcore mapping here rather than *local* vcore mapping. Across the two dimensions in the configuration space, the factors that bring tradeoffs and the degree of the tradeoffs are quite different from those discussed in Chapter 4.

5.1.2 Tradeoff Factors

Given the results of execution times for the four configurations for each workload, shown in Figure 5.2, the execution times for the four configurations are now compared for each workload. Each workload ran in a VM which is created with eight vcores[1] in the R410 test machine with overall 16 hardware threads. This setup is the same for the experimental results in this chapter. Observations can be made with the two questions in mind, Q1 and Q2. Q1 is how to extract and analyze factors, and Q2 is whether changes in execution time by each mapping is orthogonal between the two dimensions. Q2 is more about interactions between factors. One approach to extract factors is to compare execution times by estimating the possible factors resulting from changing the configuration. Here are the four cases in which comparison are made after only one change is made:

- **CCMC vs. CCMI**: The memory mapping is changed between *consolidated* and *interleaved*, while the vcore mapping is fixed to the *consolidated* mapping.

[1]Each workload has eight threads.

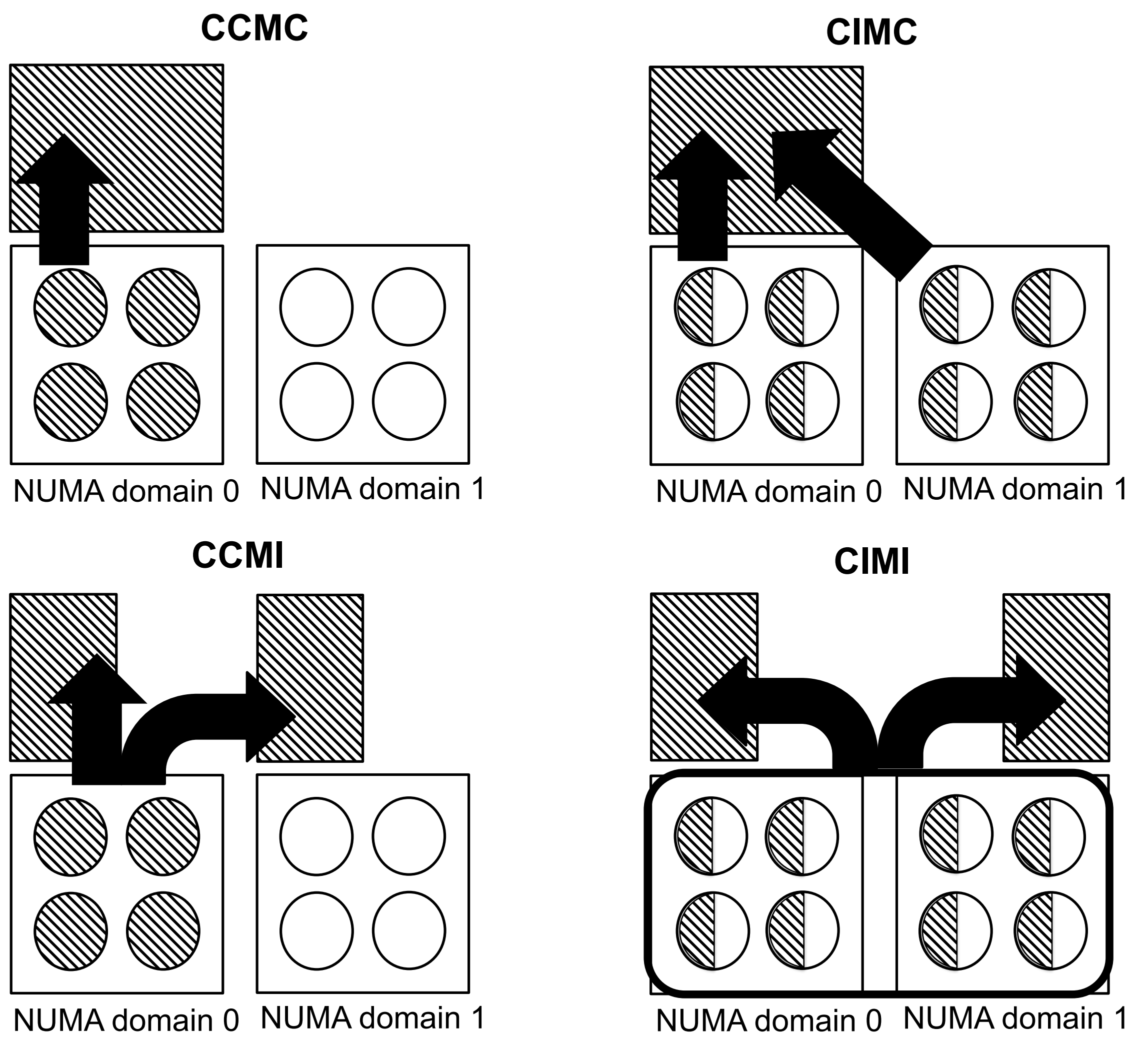


Figure 5.1: Illustration of four possible coarse-grained configurations for vcore and memory mappings: CCMC, CIMC, and CCMI, and CIMI. Each figure represents a machine with two NUMA domains. Two boxes, each of which is one CPU, below in each figure host four physical cores. Four vcores are running and each vcore takes one hardware thread, each of the two hardware threads provided by one physical core. The upper half of each figure represents memory. CCMC, for example, has four vcores that are all mapped to NUMA domain 0, and all of the VM's memory is also allocated in NUMA domain 0.

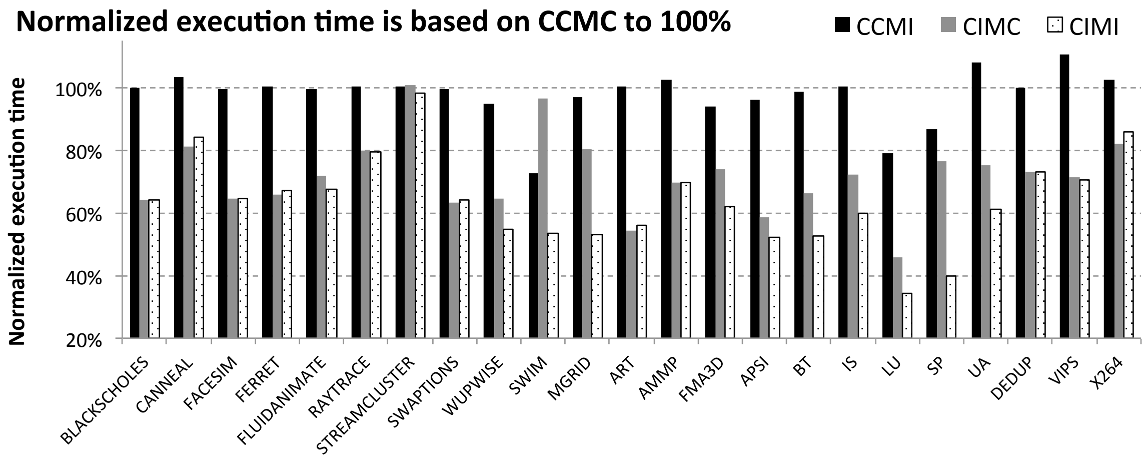


Figure 5.2: Illustration of tradeoffs for under-subscription, with respect to four possible coarse-grained mapping cases for vcore and pages: CCMC, CCMI, CIMC and CIMI. CCMC refers to virtual Core *Consolidated* and Memory *Consolidated*; similarly, CIMI represents virtual Core *Interleaved* and Memory *Interleaved*. Notice that CCMC is equivalent to the *local* vcore mapping appearing in Chapter 4 and CIMC is equivalent to the *interleaved* vcore mapping in the chapter.

Mapping Changes	Memory Bandwidth (BW)	Memory Distance	Cache Contention
CCMC $\rightarrow$ CCMI	more BW (+)	remote access (-)	no effect
CIMC $\rightarrow$ CIMI	more BW (+)	mixed	no effect
CCMC $\rightarrow$ CIMC	no change	remote access (-)	less contention (+)
CCMI $\rightarrow$ CIMI	no change	no remote access (+)	less contention (+)

Table 5.1: Summary of how three factors may change when the mapping changes. The plus and minus notation indicates the direction of the impact. Each mapping change represents a pair of configurations.

- **CIMC vs. CIMI**: The same as **CCMC vs. CCMI** except that vcore mapping is fixed to the *interleaved* mapping.
- **CCMC vs. CIMC**: Virtual core mapping is changed between the *consolidated* and the *interleaved* mappings while memory mapping is kept to the *consolidated* mapping.

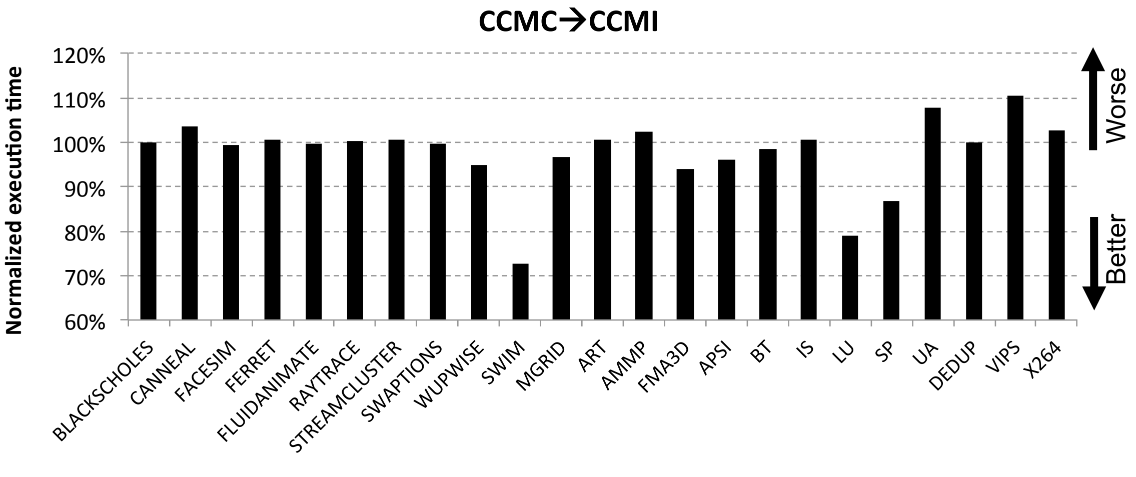


Figure 5.3: Comparison of the execution time ratio of the CCMC mapping over the CCMI mapping (CCMC→CCMI)

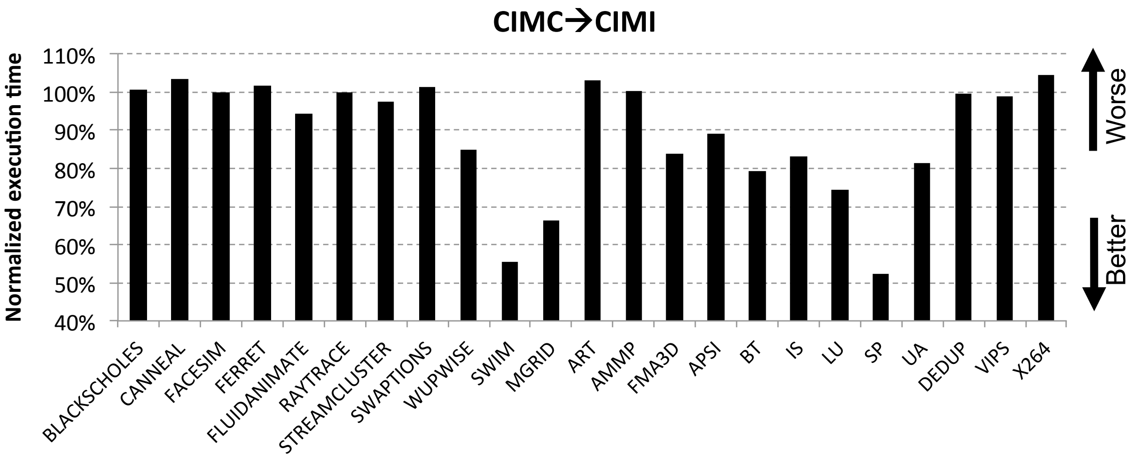


Figure 5.4: Comparison of the execution time ratio of the CIMC mapping over the CIMI mapping (CIMC→CIMI)

- **CCMI vs. CIMI**: The same as **CCMC vs. CIMC** except that the memory mapping is fixed to the *interleaved* mapping.

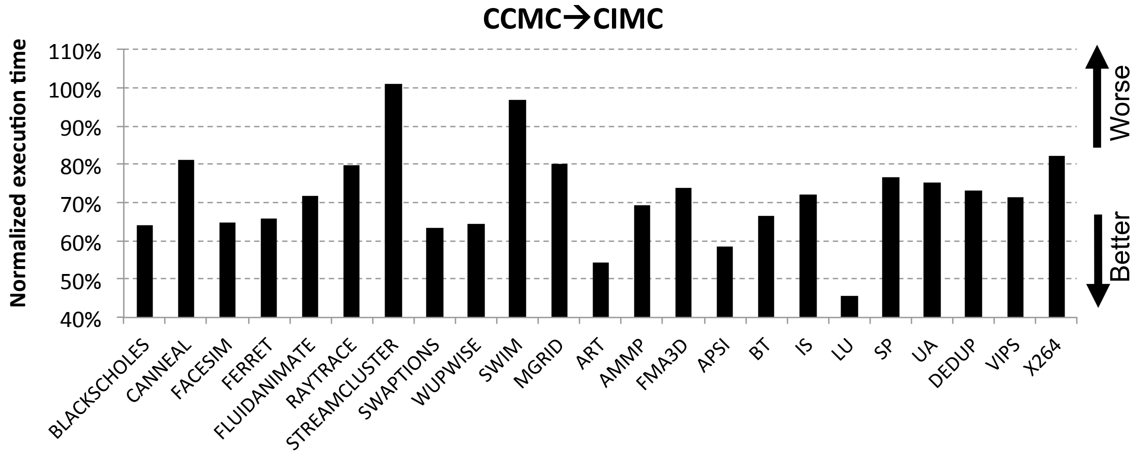


Figure 5.5: Comparison of the execution time ratio of the CCMC mapping over the CIMC mapping (CCMC→CIMC)

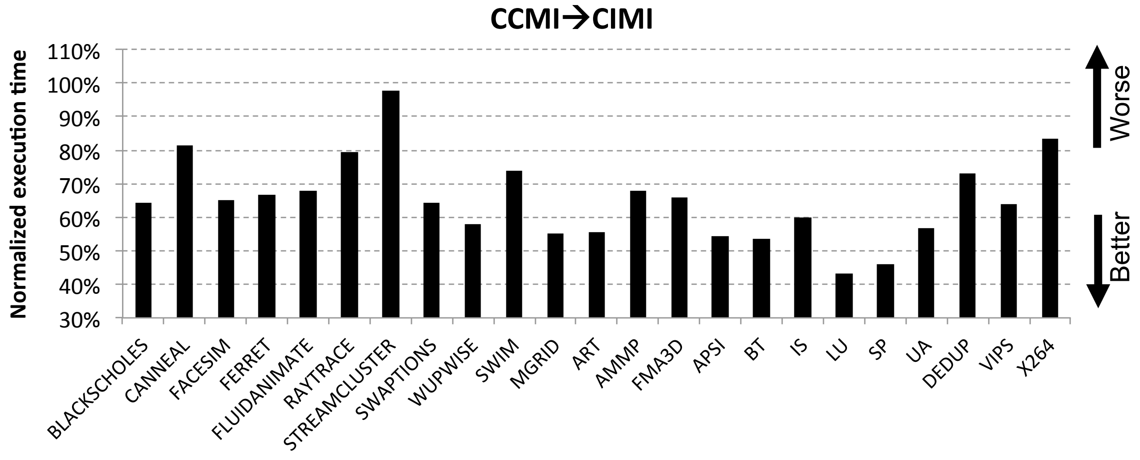


Figure 5.6: Comparison of the execution time ratio of the CCMI mapping over the CIMI mapping (CCMI→CIMI)

Along with the four cases, Table 5.1 lists potential factors and their expected impacts depending on mapping changes. These rules of thumb and based on the use of the given NUMA architectural models; even so, the degree of impact is workload dependent. More specifically, the memory-access characteristics that each workload has can cause more cache stress with a large working set and/or intensive shared-memory. Figure 5.2 already contains differences in the execution times between the four mapping configurations; the configurations are separated into the four comparative figures, as depicted in Figure 5.3, 5.4, 5.5, and 5.6. The following points are made to answer Q1:

- **Cache contention is the most important factor for performance.** Reducing cache contention by the *interleaved* vcore mapping results in consistent performance increases, irrespective of the memory mappings. As CCMC vs. CIMC and CCMI vs. CIMI show, most of benchmarks, except for a few cases, improve. However, it should be mentioned that the *interleaved* memory mapping tends to enhance performance when combine with the *interleaved* vcore mapping. This is expected because memory-access distance gets shorter as pages are interleaved. Note that CCMC vs. CIMC is the exact case that is dealt with Chapter 4. However, some variations are observed as a result of a few changes on a guest kernel as well as the different Palacios code base.

- **Memory bandwidth (BW) is observed to be the second most important factor for performance.** Relevant cases are CCMC vs. CCMI and CIMC vs. CIMI. The impact, whether positive or not, is not as consistent as the cache contention factor. Particularly when vcores are consolidated, the bandwidth advantage is reduced by the longer memory-access distances; nevertheless, the performance of some workloads are enhanced, even with the *consolidated* vcore mapping. For these cases, the adverse impact of remote access is thought to be mitigated due to their high BW demands.

- The impact of memory distance is shown in part in CCMC vs. CCMI, CCMC vs. CIMC and CCMI vs. CIMI along with the two afore mentioned factors. Basically this impact is hard to distinguish, because, for the most part, it lowers performance. However, a comparison between CCMC vs. CIMC and CCMI vs. CIMI may provide some insight. Both of the two pairs have cache contention and memory distance, as shown in Table 5.1. The *interleaved* vcore mapping is expected to boost performance by reducing cache contention. However, the extent of the benefit would be different between the two cases. Under memory consolidation, the *interleaved* vcore mapping also impairs performance due to the increased memory-access distances. Therefore, when the performance improvement shown in CCMI vs. CIMI is not realized in CCMC vs. CIMC, the impact of the memory distance can be inferred. EQUAKE and SWIM lose performance due to remote memory-accesses. STREAMCLUSTER on the other hand is the case in which there is no effect of memory-access distances. Interestingly, at the same time, vcore interleaving does not boost performance either. It is true that a reduction in cache contention boosts performance, but it is not clear whether the *interleaved* vcore mapping indeed reduces cache contention. STREAMCLUSTER is thought to be unchanged cache-contention-wise with two vcore mappings. Based on these observations, I classify memory distance as a minor factor, which is consistent with the findings reported [40], which was published recently.

Now, regarding the interactions between the two mappings, an orthogonality test helps to answer Q2. Comparing CIMI and CCMC, some workloads get performance improvements. If it is true that the improvements come from the two *interleaved* mappings, the sum of the reductions of execution time from CCMC$\rightarrow$CIMC and CCMC$\rightarrow$CCMI should be at least close to the reduction from CCMC$\rightarrow$CCMC. This orthogonality check is formulated as follows:

$$|M_{CIMI} - M_{CCMC}| == |M_{CIMC} - M_{CCMC}| + |M_{CCMI} - M_{CCMC}|$$

where, $M_{configuration}$ represents the makespan of the $configuration$. There are some workloads that meet this equality. However, many of them do not. There are two points to consider:

- **Memory bandwidth benefit is affected by cache misses (cache contention).** The *interleaved* memory mapping benefits come from the increased bandwidth. However, the memory bandwidth demand depends on off-chip memory traffic. Due to the memory hierarchy, benefits from reduced cache contention are not affected by memory mappings. However, the reduced cache contention affects on off-chip memory traffic, which determines a significant amount of the memory bandwidth demands.

- **Memory distance side-effect is apparent only when the *interleaved* mapping is exclusively applied in the two dimensions.** The increased memory-access distances for CCMC$\rightarrow$CIMC and CCMC$\rightarrow$CCMI are amortized when we switch CIMI. For example, when CCMC is changed to CIMC, remote access traffic is introduced. Now, configuring further to CIMI from CIMC, memory accesses become mixed. The vcores that must access the remote memory under CIMC now have mixed traffic, needing access both remote and local memory. Similarly, the vcores, that always access local memory under the CIMC configuration, now most access both local and remote memory.

Given these observations, it is clear that vcore mapping and memory mapping should not be considered separately. Also, the listed factors give insight into when to directly configure one of four configurations.

5.2 Detection Framework

In this scope of chapter, the baseline detection framework, the hardware-assisted software monitor, is not extended. Instead, the detection framework is checked for sensitivity.

5.2.1 Need for Sensitivity Checks

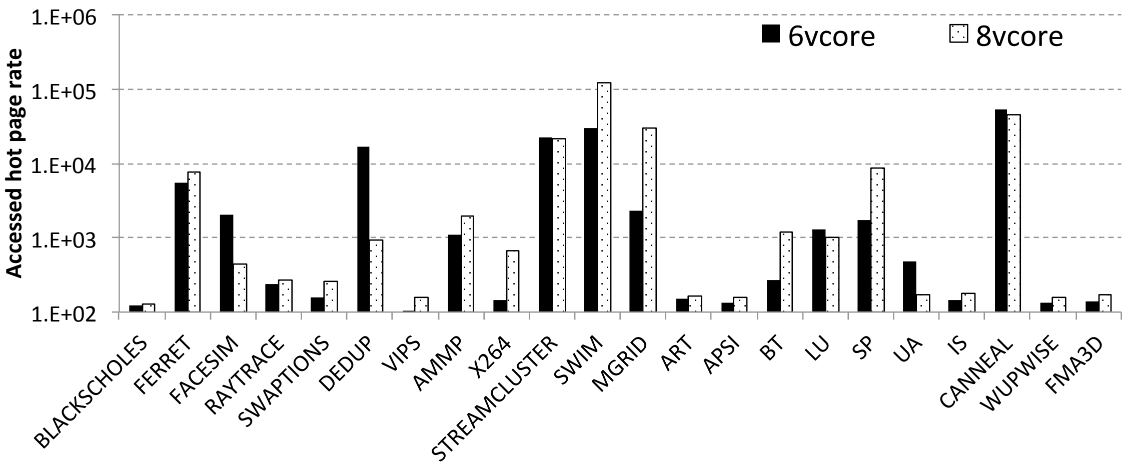
We consider sensitivity to the following: the number of threads, the mappings, the probing duration, and the sampling interval. The first two are more critical. If there is a high sensitivity to the number of threads and mappings, separate models are needed for different numbers of threads as well for different mappings. On the other hand, the last two factors, the probing duration and the sampling interval, are knobs to tune the detection framework. As long as the first two variables are not a problem, these latter two are not an issue. They are mostly sampling intervals in different levels; therefore, once threshold values are set with the various sampling intervals, the sampling intervals themselves are not changed later at all.

5.2.2 First-Order Sensitivity Checks

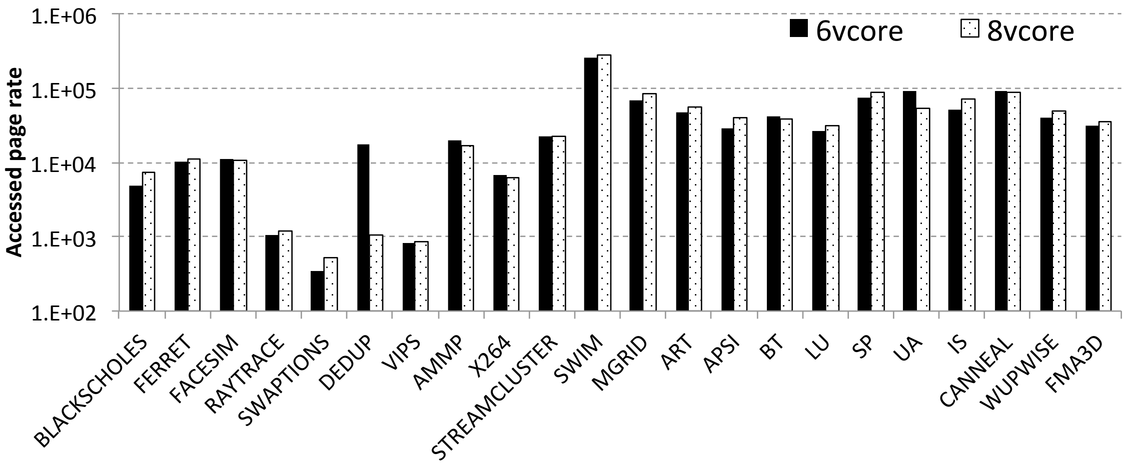
In this sensitivity check, two metrics, on access page rate per memory operation, are mostly compared. They are different in how bitmaps are aggregated, whether counting pages that are always touched ($r_{\cap as}$) or bits that are touched at least one ($r_{\cup as}$). Note that $r_{\cup as}$ is used in Chapter 4.

Sensitivity in the number of threads Table 5.2 shows the detailed setup. Figure 5.7[2] shows the $r_{\cap as}$ and $r_{\cup as}$ values from two different numbers of threads [3]. Overall, a trend

[2]Not all benchmarks are covered in the sensitivity checks.
[3]The number of threads and the number of vcores are the same.



(a) Comparisons of $r_{\cap as}$ for the changed number of vcores



(b) Comparison of $r_{\cup as}$ for the changed number of vcores

Figure 5.7: $r_{\cup as}$ has very low sensitivity to the changes in the number of vcores; and, $r_{\cup as}$ has also low sensitivity.

in both metrics is observable, regardless of the number of vcores. $r_{\cup as}$ looks to be more resilient to the changed number of vcores. DEDUP is the only benchmark that has significant changes between six and eight vcores. On the other hand, $r_{\cap as}$ has more cases. With eight vcores, the metric value for DEDUP is decreased and for the other three benchmarks,

Configuration variable	Setup
Machine	Dell PowerEdged R410 machine
Resource mapping	CCMC
Probing duration	2,394,000,000 cycles (about one sec.)
Scanning interval	10,000,000 write instructions
Number of vcores	six or eight vcores

Table 5.2: Summary of setup. Note that the number of vcores (threads) are only varied.

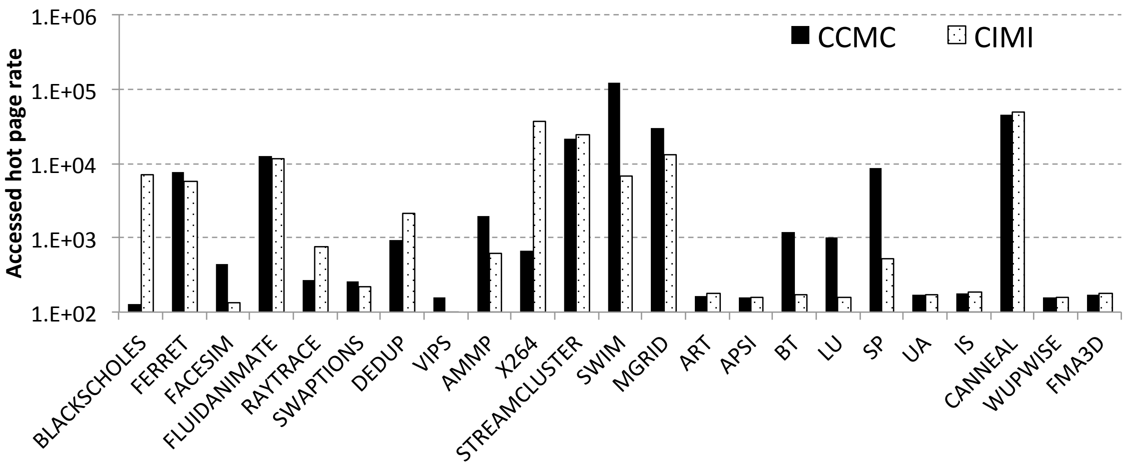
Configuration variable	Setup
Machine	Dell PowerEdged R410 machine
Resource mapping	CCMC and CIMI
Probing duration	2,394,000,000 cycles (about one sec.)
Scanning interval	10,000,000 write instructions
Number of vcores	eight threads

Table 5.3: Summary of setup. Note that the vcore- and memory mappings are varied.

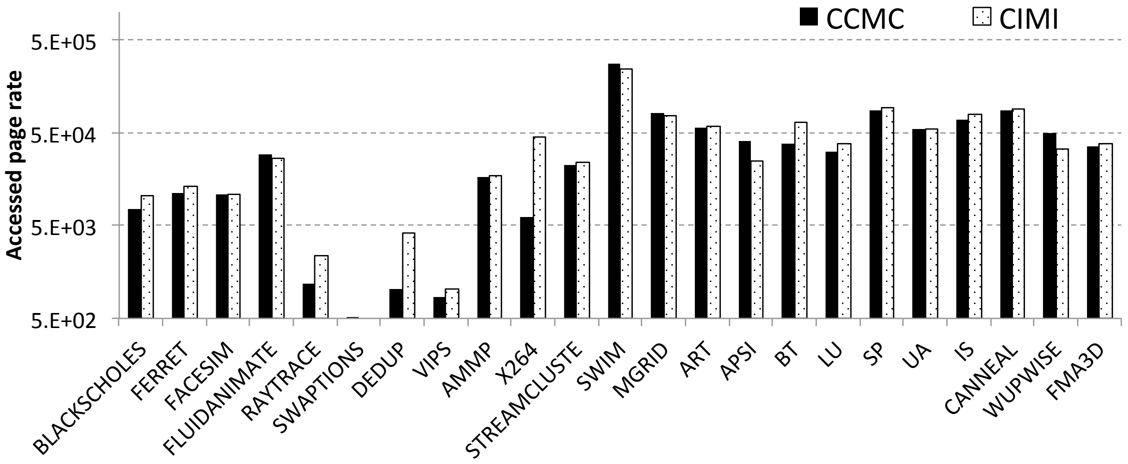
MGRID, BT, and SP, the value is increased. Considering 28 cases overall, one case particularly is quite small. The sensitivity for $r_{\cap as}$ at least is quite low as the number of vcores change.

Sensitivity in mappings Table 5.3 shows setup specifics. In this case, the resource mapping is changed. The two extreme cases of CCMC and CIMI are compared. Figure 5.8 compares the metrics between two configurations. Benchmarks are counted when they have huge changes in the metrics. Six cases for $r_{\cap as}$, and two cases for $r_{\cup as}$ are shown. The two cases for $r_{\cup as}$ are apparently noise.

Up to this point, we have found that $r_{\cup as}$ and $r_{\cap as}$ tend to less sensitive. Offline analysis with a threshold value, therefore, is applicable to the changes in the mapping and the number of vcores/threads.



(a) Comparisons on $r_{\cap as}$ for the mappings



(b) Comparisons on $r_{\cup as}$ for the mappings

Figure 5.8: Sensitivity for $r_{\cup as}$ is very low, considering the two cases that are different are due to noise. Some benchmarks show some changes in $r_{\cap as}$, nonetheless it is also very consistent.

5.2.3 Sampling Intervals

Probing duration vs scanning period Before the discussion begins, a couple of terms need to be clarified again. There are two intervals in the detection framework, the hardware-

Configuration variable	Setup
Machine	Dell PowerEdged R410 machine
Resource mapping	CCMC
Probing duration	2,394,000,000 clocks (about one sec.)
Scanning interval options evaluated	10,000,000 counts as *base count* CLK base: 100x *base count* for clocks MEM base: *base count* for memory operation **WR base: *base count* for write operation (selected)**
Number of vcores	eight threads

Table 5.4: Summary of setup. Note that the scanning interval is varied.

assisted software monitor. In the detection framework, periodic page table scans happen only in a longer time period that is called the **probing duration** – T in Figure 3.3. That is for performance reasons. Typically it is set about one second. After passing an interim period, that disables page table scans, about ten seconds or so, probing duration reoccurs. The **scanning interval** is a shorter hardware driven time interval *within* the probing duration in Section 3.2. Sensitivity to probing duration is dealt with in Chapter 6.

Cases for various scanning intervals Two kinds of memory related performance counter events are possible options for defining scanning intervals. The metrics, by the detection framework, represent page table access rate, so the scanning interval can be the number of page table accesses. Every memory operation necessarily goes through page tables or entries in a TLB. The number of page table accesses are coupled to memory operations. Another alternative to counting memory operations in to count write. Lastly, a fixed time as measured by a certain number of clock cycles is also possible. Therefore, there are all three different scanning intervals. We measure and compare six metrics:

- $r_{\cap am}$: the average accessed hot page rate per memory operation
- $r_{\cup am}$: the average accessed page rate per memory operation

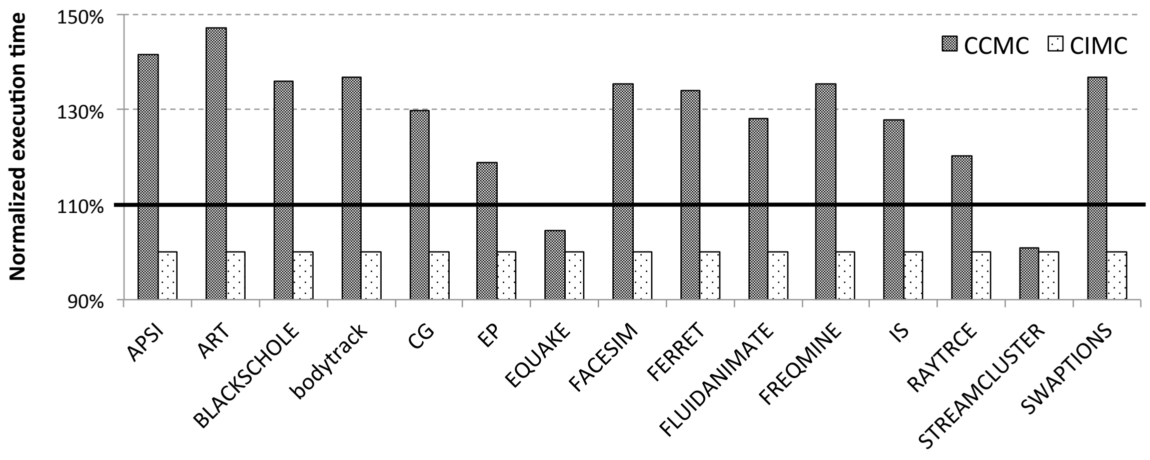


Figure 5.9: Performance comparison between the CCMC and CIMC mappings. Two cases, STREAMCLUSTER and EQUAKE, show less impact on interleaved vcores, which reflects that there are very few cache contention differences in between CCMC and CIMC.

- $r_{\cap as}$: the average accessed hot page rate per store operation
- $r_{\cup as}$: the average accessed page rate per store operation
- $r_{\cap ac}$: the average accessed hot page rate per clock cycle
- $r_{\cup ac}$: the average accessed page rate per clock cycle

Sensitivity check and scanning interval selection First, as shown in Figure 5.11, the metrics with the $\cup$ notation do not have much sensitivity to different scanning intervals. Therefore, metrics with the $\cap$ notation are the focus for the following sensitivity checks. The variations on these metrics depending a lot on which type of sampling intervals is chosen, as shown in Figure 5.10. One approach to select a scanning interval is by comparing measured metrics with the objective metric. We show correlations of the metrics with the execution time results in Figure 5.9. EQUAKE and STREAMCLUSTER in Figure 5.9

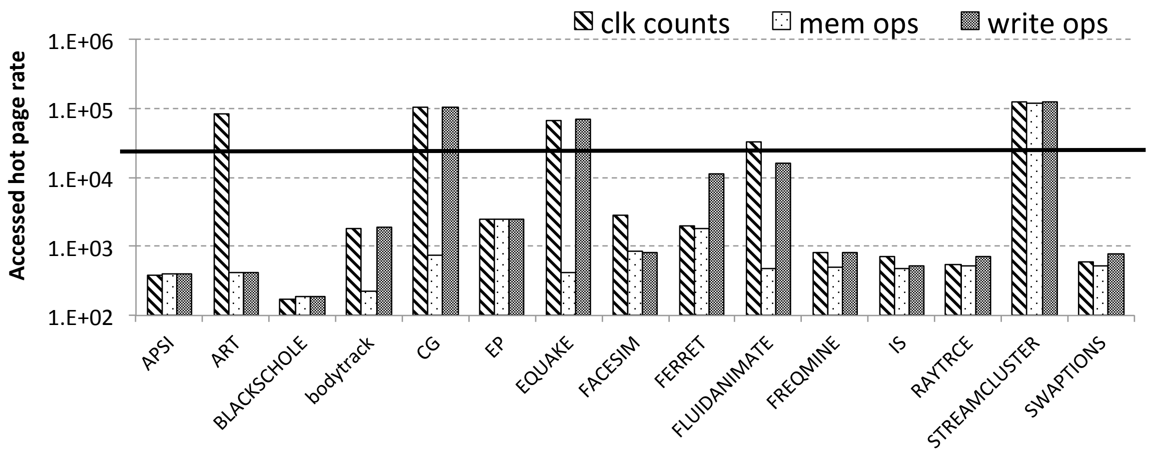


Figure 5.10: Sensitivity in $r_{\cap as}$ for different scanning interval. Three outstanding cases are observed with write operation; at the same time, they likely match with the extraordinary two cases in Figure 5.9.

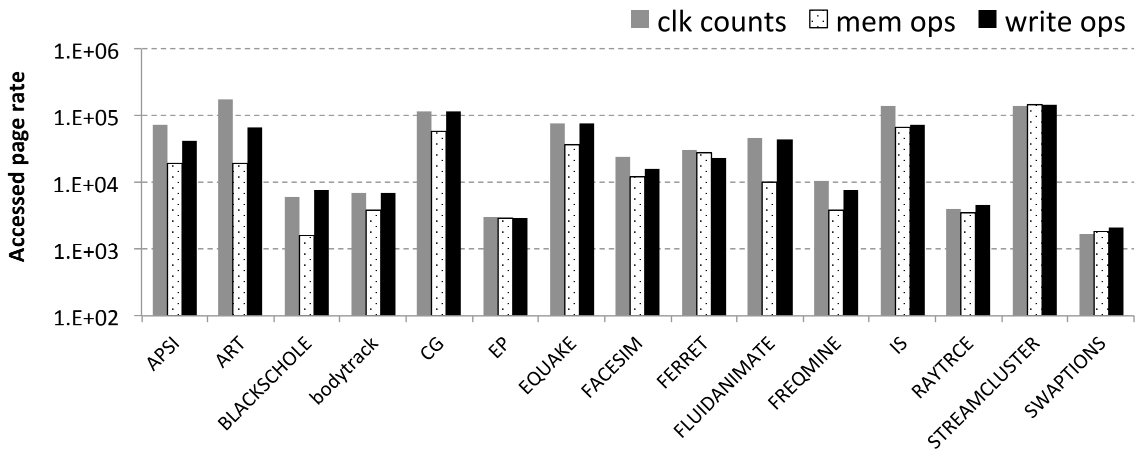


Figure 5.11: Sensitivity in $r_{\cup as}$ for various scanning intervals is low.

are two workloads that less than 10% differences. On the other side, Figure 5.10 presents measured three metrics, $r_{\cap am}$, $r_{\cap as}$, and $r_{\cap ac}$. $r_{\cap as}$ seems to have some correlation with the execution time results because the three cases, CG, EQUAKE, and STREAMCLUSTER, are above a threshold, 30000. EQUAKE and STREAMCLUSTER are common with the results from Figure 5.9. However, cache miss rate for CG is below 10 %, so CG should not classified as the CacheStress class. The cases with small differences in CCMC-vs-CIMC reflect a high-cache-stress characteristic, so a small cache miss rate is not aligned with the performance results. Also, recall that an average of $r_{\cap as}$ and $r_{\cap am}$ is used to classify HighCacheMissRate scenario as depicted in Figure 4.2.

Given these results, we have taken a count of write operation for our scanning interval. Detail on the test setup is found in Table 5.4. Note that other two variables – probing duration and different machine – are discussed in Chapter 6 and 8 respectively.

5.2.4 Detection Mechanism

The detection framework now is quite simple, with only one aggregation along with one probing phase. Metrics collected by the aggregator are reduced to two: $r_{\cap as}$ and $r_{\cup as}$. Note this is designed for the NAVAR policy. Other policies such as those for fine-grained mapping, may be different. The detection framework operates as follows:

```
Init(aggregator): [Invoked at startup on aggregator]
  SetTimer($T$, Aggregate)
  **for all** vcores $i$ **do**
    $EnableScan_i = 1$
    Force vcore $i$ to run InitVcore($i$)
  **end for**

InitVcore($i$): [Invoked at on vcore $i$]
  $hot\_accessed_i = \{k : 0 \ldots m-1\}$
  $accessed_i = \{k : 0 \ldots m-1\}$
  Set PMU exception for number of store operations to trigger Scan($i$).
```

```
Scan($i$): [invoked when PMU exception occurs]
  if $EnableScan_i = 1$ then
    for all present shadow (or nested) PTEs on vcore $i$ do
      $k$ = DeriveGuestPhysicalPageNumberFrom(PTE)
      $curacc_i = \emptyset$
      if PTE.accessed then
        $curacc_i = curacc_i \cup \{k\}$
      end if
      PTE.accessed=0
    end for
    $hot\_accessed_i = hot\_accessed_i \cap curacc_i$
    $accessed_i = accessed_i \cup curacc_i$
  end if
Aggregate(aggregator)
  for all vcores $i$ do
    $EnableScan_i = 0$
    $hot\_accessed\_per\_store_i = hot\_accessed_i$
    $accessed\_per\_store_i = accessed_i$
  end for
  $r_{\cap as} = \frac{1}{n} \sum_{i=0}^{n-1} |hot\_accessed\_per\_store_i|$
  $r_{\cup as} = \frac{1}{n} \sum_{i=0}^{n-1} |accessed\_per\_store_i|$
```

5.3 Offline Modeling for NAVAR

Before describing the details of the NAVAR policy for under-subscription, it is necessary to describe the offline modeling that is used to produce a decision tree that is required in NAVAR. The following modeling effort is specific to NAVAR. The objective of offline modeling is to build a decision tree that can be used to identify the optimal resource mapping in the configuration space from the observed metrics.

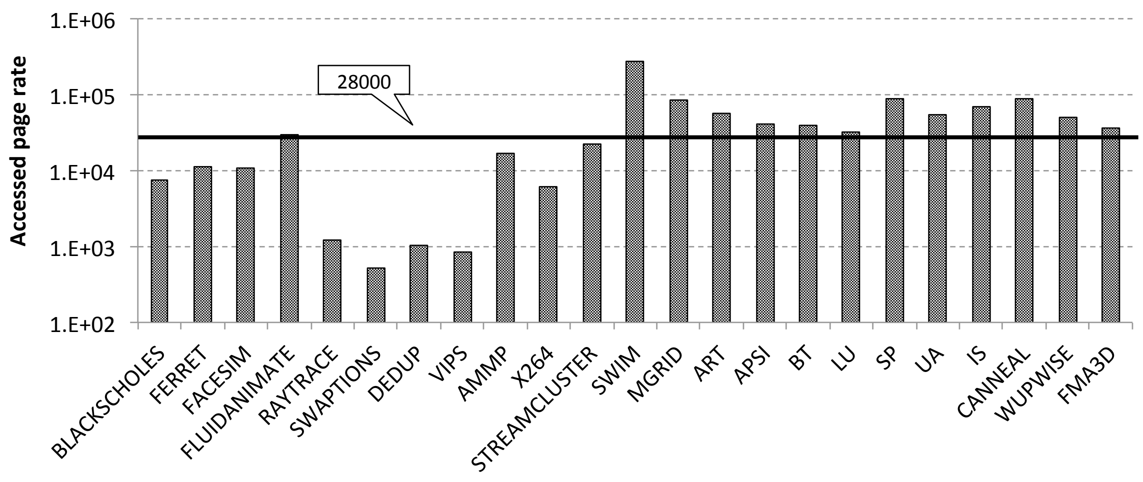
5.3.1 Classification

$r_{\cap as}$ and $r_{\cup as}$ are considered as conveying properties of the referenced pages. The difference is whether they are accessed frequently (hot) or accessed at least once. Also, memory-access traffic occurs in two domains, on-chip (cache) and off-chip (main memory). According to the degree of traffic in each domain, a workload can be coarsely classified into two classes, high and low, for each domain.

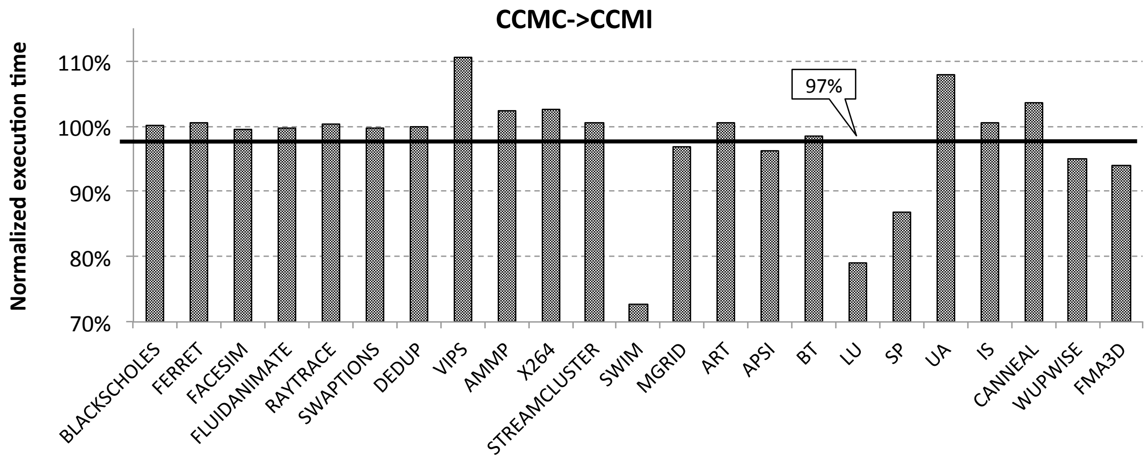
LargeWorkingSetSize (LWSS) Classification

Off-chip memory traffic for LWSS classification $r_{\cup as}$ measures degrees of overall accessed pages. At least, cold misses in cache must touch the main memory. The overall amount of cold misses constitutes a large portion of the main memory traffic, i.e. off-chip memory traffic. The threshold is set on 28,000, as depicted in Figure 5.12. If $r_{\cup as}$ is above the threshold, the workload is classified as LargeWorkingSetSize (LWSS), otherwise it is classified as !LWSS. This threshold value is set to align with the execution time ratio, as shown in Figure 5.3. More details about the execution time ratio follow.

LWSS from preliminary performance results Comparing execution times between CCMI and CCMC provides insight for the classifications. The fact that each workload



(a) Measured values for $r_{\cup as}$



(b) Execution time ratio of CCMI over CCMC

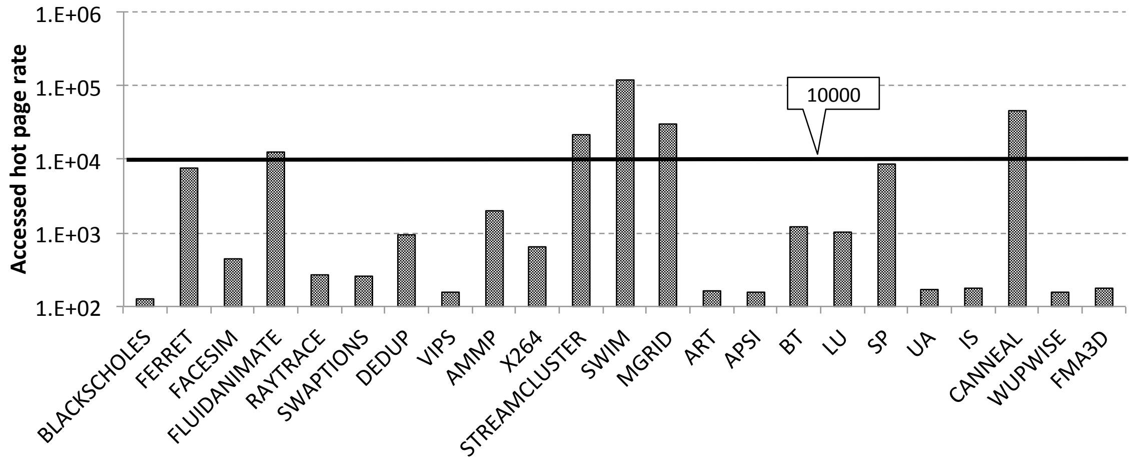
Figure 5.12: The LWSS class is defined through the measured $r_{\cup as}$ values with the threshold inspired by the performance differences between CCMC and CCMI. $r_{\cup as}$ is measured with the setup described in Table 5.2 except that 8 vcore are used.

has different performance variations for the changed resource mappings implies the workload's characteristics. For example, performance improvement with *interleaved* memory over CCMC is different on different workloads. Using the threshold shown in Figure 5.12, workloads are classified in one of the two classes, LWSS and !LWSS. This classification is now called the $reference$ classification. CIMC vs. CIMI also conveys *interleaved* memory impact, but CCMC vs. CCMI is referenced because it is thought that it highlights the degree of off-chip traffic. *Consolidated* vcore mapping enforces memory distance impact (negative impact); when performance gain with *interleaved* memory is visible, the BW demand from the workload is large enough to overcome the side effect of increased memory distance. Therefore, this classification, the $reference$ classification, is expected to provide an insight for the selection of the threshold by comparing execution time differences between various resource mappings.

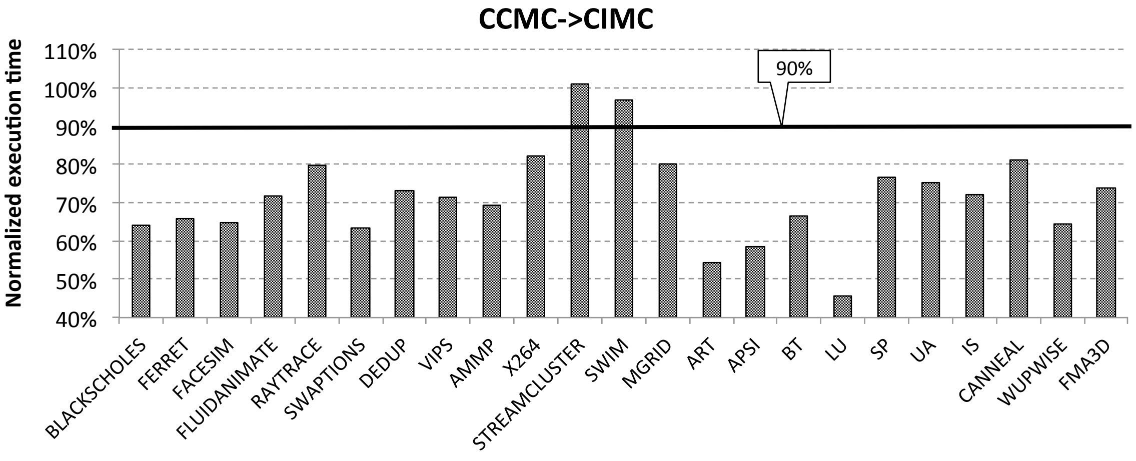
CacheStress (CS) Classification

On-chip memory traffic for CS classification The other metric, $r_{\cap as}$, is useful for classifying workloads in on-chip memory traffic. It filters the pages that are more continuously referenced (hot pages), and the set of pages captured by $r_{\cap as}$ is a sub-set of the pages from $r_{\cup as}$. In terms of access locality, $r_{\cap as}$ likely represents the degree of data that is best to keep in the cache. In fact, the sensitivity check for the scanning interval selects the scanning interval that matches for our benchmarks' performance results of CCMC and CIMC. The threshold value on $r_{\cap as}$ is set to 10000. When a workload has $r_{\cap as}$ above the threshold, it is classified as CacheStress (CS); otherwise, the workload is classified as !CS. Even if the $reference$ classification results are different, they are only used only to establish the classification that actually is included in the decision tree.

CS from preliminary performance results Here is a brief explanation of why CCMC vs. CIMC is taken instead of CCMI vs. CIMI to complete the $reference$ classification. Moving from CCMC to CIMC includes both positive and negative impacts, including reduced cache contention and memory distance. The reduced execution time shown in Fig-



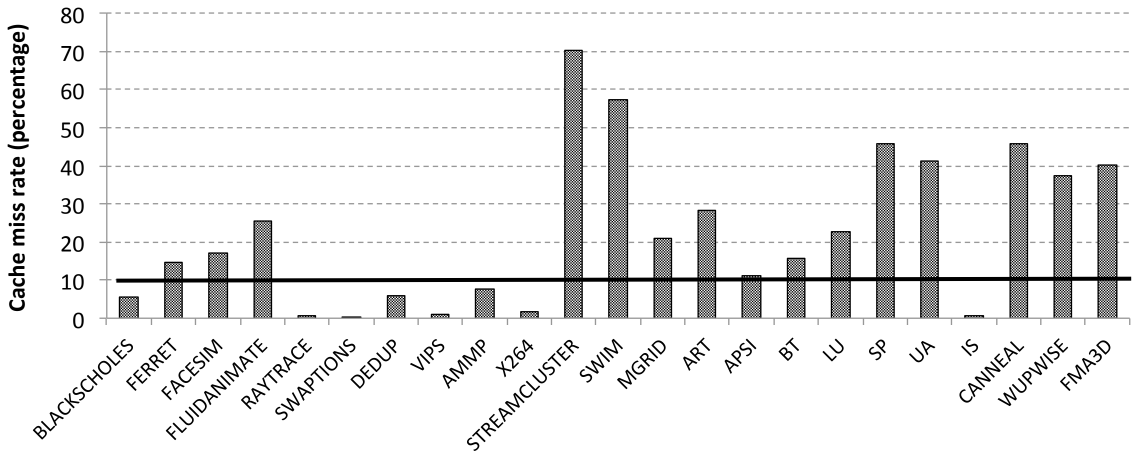
(a) Measured values for $r_{\cap as}$



(b) Execution time ratio of CIMC over CCMC

Figure 5.13: $r_{\cap as}$ is used to determine the CS class, and the threshold is set similar to in the LWSS class. The performance results between CCMC and CIMC are used to define threshold value. The specific setup is the same as described in Figure 5.12.

ure 5.13 highlights the cases in which the benefit of reduced cache contention outweighs the increased memory distance penalty. In terms of classification, this case, CCMC vs. CIMC, is clearer, because the negative factor distinguishes the degree of cache perfor-



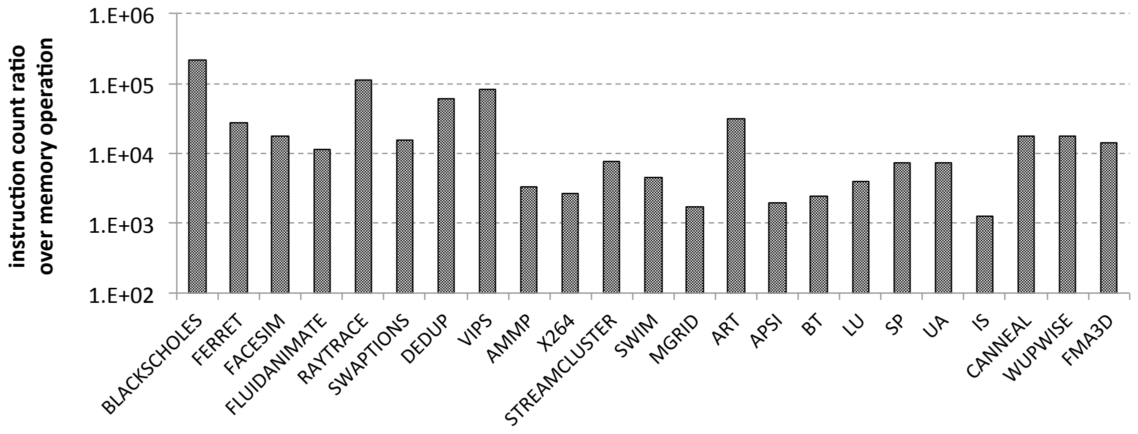


Figure 5.14: Variations in instructions across workloads

mance gains.

Correlation Check for Other Metrics

Some metrics produced by performance counters (the hardware monitor) are also measured, including the cache miss rate and the ratio of memory instructions to all instructions; however, they do not show any significant correlations as to the metrics selected

	CacheStress (CS)	**LargeWorkingSetSize (LWSS)**
Definition	Contiguously accessed page count is high	Access page count is high
Domains	On-chip memory traffic	Off-chip memory traffic
Resource mapping effects	Little benefit for *interleaved* vcore mapping	Large benfit for *interleaved* memory mapping

Table 5.5: Comparisons of the two dimensions for the classification (the two classifiers).

above. Nonetheless, cache miss rates above 10% are the cases that are classified in CS. Thus, it may help to check validity of the classification and cache miss rate is included in the decision tree for NAVAR. This comparison supports the hardware-assisted software monitor (the detection framework) over the hardware monitor (the performance counters).

5.3.2 Discussion of Classification

Further discussion is needed to clarify the implications of the classes and to explain how the classification helps with resource mapping. The following discussions include the conveyed meaning of each class, interactions and the relationship between the two classes, and then the expected vcore and memory mappings for the four classes:

- Cache Stress (**CS**) and Large Working Set Size (**LWSS**): **CS&LWSS**
- Not Cache Stress (**!CS**) and Large Working Set Size (**LWSS**):**!CS&LWSS**
- Cache Stress (**CS**) and Not Large Working Set Size (**!LWSS**): **CS&!LWSS**
- Not Cache Stress (**!CS**) and Not Large Working Set Size (**!LWSS**): **!CS&!LWSS**

Implications of class The two separate classifiers are used to develop the four overall classes: CS&LWSS, !CS&LWSS, CS&!LWSS and !CS&!LWSS. Each classifier addresses a separate dimension, and the four metrics come from the combination of the dimensions. Here, the two classifiers are compared with each other to provide a better understanding. Two metrics, $r_{\cap as}$ and $r_{\cup as}$, are used to classify workloads into four classes. Each metric is used to divide two parts; $r_{\cap as}$ is used to differentiate CS from !CS, and $r_{\cup as}$ differentiate LWSS from !LWSS. CS includes on such workloads that access a number pages (hot pages) consecutively that exceeds the threshold. Therefore, the memory traffic for two classifications, CS and LWSS, is different. CS is focused on the intensity of the on-chip memory traffic as capturing in hot pages, whereas LWSS focused on the degree of off-chip memory traffic as reflected in the total number of pages that are accessed. When workloads are classified as CS, the benefit from the *interleaved* vcore mapping is rather small. Conversely, the expected performance gain with the *interleaved* memory mapping for the LWSS class is huge. These are the implications of and comparisons of the two classifications. It is important to understand the differences between consequences and characteristics. Cache miss rate is a result of interaction between provisioning cache size and workload characteristics. These classifications are on workload characteristics, thus CS does not refer to cache miss rate as it depends on a given cache size. Rather, CS captures the characteristic of memory traffic that it consists of heavy traffic in the on-chip domain.

Orthogonality of the two dimensions Among the four classes, CS&!LWSS and !CS&LWSS are somewhat difficult to accept, especially given the statement "memory bandwidth benefit is affected by cache misses" in Section 5.1.2. Two cases are depicted in Figure 5.15. STREAMCLUSTER is one case for CS&!LWSS and APSI is for !CS&LWSS. These two cases are quite opposite. APSI has a very small number of hot pages, but it involves a large number of pages, whereas STREAMCLUSTER has a huge number of hot pages, with the hot pages making up almost all of the pages it accesses. APSI is the case that does not

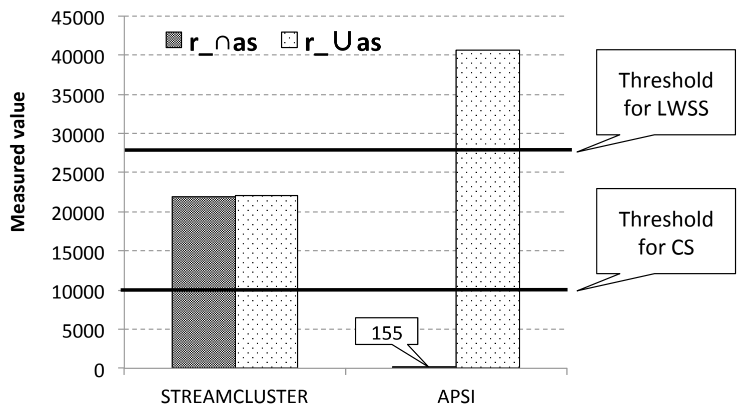


Figure 5.15: Two cases from CS&!LWSS and !CS&LWSS classes. The comparison demonstrates no dependency between the CS and LWSS characteristics.

Class	Resource mapping
CS&LWSS	CIMI
CS&!LWSS	CCMC
!CS&LWSS	CIMI
!CS&!LWSS	CIMC

Table 5.6: Resource mapping table for the four classes.

invoke huge stress on the cache, but still demands memory bandwidth. STREAMCLUSTER looks to have relatively smaller working set size, but it accesses almost all of them continuously, which effectively causes stresses on caches. The statement in Section 5.1.2 is very general, and, in these cases, the CS characteristic does not necessarily match the LWSS characteristic and vice versa. Therefore, orthogonality exists in the two dimensions.

Mapping class to resource mapping Given the class of the workload, the question remains concerning how to map it to an appropriate vcore- and memory mapping. Our mapping strategy is as described below:

- The CS class is better for the *consolidated* vcore mapping for lower energy. CS workloads have $<$10% performance gain when using a *interleaved* vcore mapping instead of a *consolidated* vcore mapping, depending on the workload, and about 10% more power is consumed, which is usually fixed regardless of the workload.

- The !CS class is better for the *interleaved* vcore mapping for both energy and performance. This class of workloads has at least 10% performance gain, when so mapped, which is more than the cost, the increased power. Therefore, the *interleaved* vcore mapping is good for energy as well.

- LWSS is better for the *interleaved* memory mapping. Regardless of the vcore mapping, the performance gain with the *interleaved* memory mapping is at least 3%, which is at least comparable to the cost in power, about a 5% increase for the *interleaved* memory mapping over the *consolidated* mapping.

- !LWSS is better for the *consolidated* memory mapping for energy efficiency. The !LWSS class of workloads mostly does not get performance benefits with the *interleaved* memory mapping, except for a few cases, thus, it is better to conservatively consolidate memory to lower energy.

However, there is one case in which the listed principles needs to be adjusted. When the LWSS type of workloads is mapped using the *interleaved* memory mapping, the *consolidated* vcore mapping is not a good choice for energy. Comparisons between CCMI and CIMI show that CIMI is always better, because it reduces cache contention and increases memory bandwidth. Therefore, the CS and LWSS classes of workloads benefit more from CIMI mapping than from CCMI mapping. A summary of the mapping strategy is presented in Table 5.6.

Class	Workloads
CS&LWSS	SWIM, MGRID, APPLU, CANNEAL, FLUIDANIMATE
CS&!LWSS	STREAMCLUSTER
!CS&LWSS	BT, LU, SP, APPLU, GAFORT, UA, IS, WUPWISE, FMA3D, APSI, ART
!CS&!LWSS	BLACKSCHOLES, FERRET, FACESIM, RAYTRACE, SWAPTION, DEDUP, VIPS, GALGEL, AMMP, X264, CANNEAL

Table 5.7: Classification results. With the detection framework, each workload is classifiable during runtime; then, according to the resource mapping table, the workload is configured with the best mapping. The decision tree equipped in the policy component processes the described task.

5.4 Coarse-grained VCore/Memory Mapping Policies

We now describe the coarse-grained resource mapping policy for the under-subscription situation. The coarse-grained policy includes the NAVAR policy as well as the brute-force approach. The brute-force approach can be attempted at first, because it does not require models and any relevant offline analysis; however, the model-based prediction without feedback loop is chosen for the skeleton of the NAVAR policy. The followings describe the NAVAR policy and the brute-force approach.

5.4.1 NAVAR for Under-subscription

The structure of the NAVAR policy for under-subscription is model-based prediction with open-loop control. DecisionTree and Interim are the two core functions. DecisionTree receives the measured metrics, $r_{\cap as}$, $r_{\cup as}$, and the cache miss rate, and determines the mapping. The interim function controls the sleep time dynamically. By dong that, the frequency of searching for an optimum mapping is dynamically adjusted. This is important since there is no feedback. When the application is in a phase transition, multiple predictions can compensate open-loop control. On other hand, when the phase is not changing, infrequent predictions are sufficient as the prediction results tent to be consistent. The following pseudocode illustrates the NAVAR policy implementation. $INTER_{ConfigurationSpace}$ and $CONSOLID_{ConfigurationSpace}$ represent the *interleaved* mapping and *consolidated* mapping separately in $ConfigurationSpace$, either for the vcore or the memory configuration.

```
$exe\_state \leftarrow$ INIT_STATE
$interim\_scale \leftarrow 1$
$tsc_{state\_init} \leftarrow$ *readtsc*; $tsc_{prev} \leftarrow$ *readtsc*
$ovrhd_{probe} \leftarrow 0$; $ovrhd_{vcore} \leftarrow 0$; $ovrhd_{mem} \leftarrow 0$
$Mapping_{cur} \leftarrow (INTER_{vcore}, INTER_{mem})$

**while** $1$ **do**
  $ovrhd_{probe} \leftarrow$ Probing($metrics$)
  $(VCoreMapPolicy, MemMapPolicy) \leftarrow$ DecisionTreeUS($metrics$)
  $(VCoreMap_{need}, MemMap_{need}) \leftarrow$ ExtractMap($VCoreMapPolicy, MemMapPolicy$)

  **if** $(VCoreMap_{cur}, MemMap_{cur}) = (VCoreMap_{need}, MemMap_{need})$ **then**
    $interim\_scale\_mode \leftarrow$ NEED_SCALE_UP
  **else**
    **if** $VCoreMap_{cur} \neq VCoreMap_{next}$ **then**
      $ovrhd \leftarrow$ VCoreReMapping($VCoreMap_{next}, VM0$)
      $ovrhd_{vcore} \leftarrow ovrhd + ovrhd_{vcore}$
    **end if**
    **if** $MemMap_{cur} \neq MemMap_{next}$ **then**
      $ovrh \leftarrow$ MemReMapping($MemMap_{next}, VM0$)
      $ovrhd_{mem} \leftarrow ovrhd + ovrhd_{mem}$
    **end if**
```

```
        $(VCoreMap_{cur}, MemMap_{cur}) \leftarrow (VCoreMap_{need}, MemMap_{need})$
        $interim\_scale\_mode \leftarrow$ NEED_RESET
    **end if**
    Interim($ovrhd_{probe}, ovrhd_{vcore}, ovrhd_{mem}$)
**end while**
```

DecisionTreeUS($metrics$) returns the determined mapping information. For the power objective, it first immediately returns the *consolidated* mappings for both vcore and memory. For performance and energy objectives, it checks whether the measured metric values are above the thresholds or not. Accordingly, it finds the best mappings through the embedded decision tree.

```
**if** $objective$ is power **then**
    $(VCoreMapPolicy, MemMapPolicy) \leftarrow (CONSOLID_{vcore}, CONSOLID_{mem})$
**else if** $r_{\cup as} > threshold\_LWSS$ **then**
    $(VCoreMapPolicy, MemMapPolicy) \leftarrow (INTER_{vcore}, INTER_{mem})$
**else if** $r_{\cap as} > threshold\_CS$ **then**
    **if** $objective$ is energy and $CacheMissRate > threshold\_CS\_CacheMissRate$ **then**
        $(VCoreMapPolicy, MemMapPolicy) \leftarrow (CONSOLID_{vcore}, CONSOLID_{mem})$
    **end if**
**else**
    $(VCoreMapPolicy, MemMapPolicy) \leftarrow (INTER_{vcore}, CONSOLID_{mem})$
**end if**
**return** $(VCoreMapPolicy, MemMapPolicy)$
```

Probing($metrics$) collects the performance metrics. It begins with a call to Init(aggregator) to initiate a probing internal to collect the two metrics, $r_{\cup as}$ and $r_{\cap as}$, and the $CacheMissRate$. Then, it updates the $metrics$ vector with updated collected values.

```
$tsc_{start} \leftarrow$ *readtsc*
Call Init(aggregator)
Measure $cache\_miss\_rate$
$metrics \leftarrow (r_{\cap as}, r_{\cup as}, CacheMissRate)$.
$tsc_{end} \leftarrow$ *readtsc*
**return** $(tsc_{end} - tsc_{start})$
```

ExtractMap($VCoreMapPolicy$, $MemMapPolicy$) extracts the given mapping info from a predefined table.

VCoreReMapping($VCoreMap_{next}$, $VMID$) reconfigure the vcore mapping and returns the overhead of the remapping:

```
$tsc_{start} \leftarrow$ *readtsc*

change *vcore mapping* to $VCoreMap_{next}$ for the $VMID$ VM
$tsc_{end} \leftarrow$ *readtsc*
**return** $(tsc_{end} - tsc_{start})$
```

MemReMapping($MemMap_{next}$, $VMID$) changes the memory mapping according to the request, and returns the overhead of the remapping:

```
$tsc_{start} \leftarrow$ *readtsc*
change *page mapping* to $MemMap_{next}$ for $VMID$ VM
$tsc_{end} \leftarrow$ *readtsc*
**return** $(tsc_{end} - tsc_{start})$
```

Interim($ovrhd_{probe}$,$ovrhd_{remap}$) controls the frequency of searching for the best resource mapping by changing the sleep time so as to call DecisionTree depending on status. When the selected configurations are consecutively the same, the status (exe_state) is set to STA-

BLE_STATE. In this state, Interim increases sleep time incrementally to avoid unnecessary searches. Also, it sleeps until the demanded overhead ratio is achieved. An overhead ratio is calculated as the measured migration time for the vcore and page migrations plus the overall page table scanning time divided by the spent wall clock time. However, when DecisionTree indicates a different configuration than previously found under STABLE_STATE, Interim changes the execution state to INIT_STATE and resets sleep time to the default. It also skips sleeps and keeps searching until STABLE_STATE is reached.

```
if $exe\_state$ is STABLE_STATE and $interim\_scale\_mode$ is NEED_RESET then
    $exe\_state \leftarrow$ INIT_STATE; $tsc_{state\_init} \leftarrow$ readtsc; $interim\_scale \leftarrow 1$
end if
sleep for $(window_{interim} \cdot interim\_scale)$
$tsc \leftarrow$ readtsc $- tsc_{prev}$
if $exe\_state$ is not INIT_STATE then
    if $objective$ is not performance then
        $ovrhd \leftarrow weight_{probe} \cdot ovrhd_{probe} + weight_{vcore} \cdot ovrhd_{vcore} + weight_{mem} \cdot ovrhd_{mem}$
    else
        $ovrhd \leftarrow ovrhd_{probe} + ovrhd_{vcore} + ovrhd_{mem}$
    end if
    while $ovrhd / tsc > threshold_{ovrhd}$ do
        sleep for $window_{interim}$
        $tsc \leftarrow$ readtsc $- tsc_{prev}$
    end while
    $ovrhd_{probe} \leftarrow 0$; $ovrhd_{vcore} \leftarrow 0$; $ovrhd_{mem} \leftarrow 0$
end if
if $intermin\_scale\_mode$ is NEED_SCALEUP then
    $interim\_scale \leftarrow 2 \cdot interim\_scale$
```

```
**end if**
$tsc \leftarrow$ ***readtsc*** $- tsc_{state\_init}$
**if** $exe\_state$ is INIT_STATE and $tsc > 20 \cdot window_{interim}$ **then**
    $exe\_state \leftarrow$ STABLE_STATE
    $tsc_{state\_init} \leftarrow$ ***readtsc***
**else if** $exe\_state$ is STABLE_STATE and $tsc > 30 \cdot window_{interim}$ **then**
    $interim\_scale \leftarrow 1$
    $tsc_{state\_init} \leftarrow$ ***readtsc***
**end if**
$tsc_{prev} \leftarrow$ ***readtsc***
**return**
```

5.4.2 Brute-force Approach

This approach is literally brute-force searching for the best mapping. Without any modeling and characterization, it simply goes through all possible configurations and picks one with the best result in the observable metric. Here, CPI[4] is used. The overall skeleton is the same as the NAVAR policy, but DecisionTree is replaced with BruteForce and Probing is removed.

```
$exe\_state \leftarrow$ INIT_STATE
$interim\_scale \leftarrow 1$
$tsc_{state\_init} \leftarrow$ ***readtsc***; $tsc_{prev} \leftarrow$ ***readtsc***
$ovrhd_{probe} \leftarrow 0$; $ovrhd_{vcore} \leftarrow 0$; $ovrhd_{mem} \leftarrow 0$
$Mapping_{cur} \leftarrow (INTER_{vcore}, INTER_{mem})$
```

[4]CPI is not an accurate metric for estimating the execution time. However, regardless of its accuracy, the brute-force approach is found to be a limited solution.

```
while $1$ do
  $(\mathit{VCoreMap}_{need}, \mathit{MemMap}_{need})$ ← BruteForce()
  if $(\mathit{VCoreMap}_{cur}, \mathit{MemMap}_{cur}) = (\mathit{VCoreMap}_{need}, \mathit{MemMap}_{need})$ then
    $\mathit{interim\_scale\_mode}$ ← NEED_SCALE_UP
  else
    if $\mathit{VCoreMap}_{cur} \neq \mathit{VCoreMap}_{next}$ then
      $\mathit{ovrhd}$ ← VCoreReMapping($\mathit{VCoreMap}_{next}$, $\mathit{VM0}$)
      $\mathit{ovrhd}_{vcore} \leftarrow \mathit{ovrhd} + \mathit{ovrhd}_{vcore}$
    end if
    if $\mathit{MemMap}_{cur} \neq \mathit{MemMap}_{next}$ then
      $\mathit{ovrhd}$ ← MemReMapping($\mathit{MemMap}_{next}$, $\mathit{VM0}$)
      $\mathit{ovrhd}_{mem} \leftarrow \mathit{ovrhd} + \mathit{ovrhd}_{mem}$
    end if
    $(\mathit{VCoreMap}_{cur}, \mathit{MemMap}_{cur}) \leftarrow (\mathit{VCoreMap}_{need}, \mathit{MemMap}_{need})$
    $\mathit{interim\_scale\_mode}$ ← NEED_RESET
  end if
  Interim($\mathit{ovrhd}_{probe}$, $\mathit{ovrhd}_{remap}$)
end while
```

BruteForce() basically works through all four coarse-grained resource mapping cases and measures CPIs. Then, it picks the best one (lowest CPI).

```
if $\mathit{objective}$ is power then
  $(\mathit{VCoreMap}_{need}, \mathit{MemMap}_{need}) \leftarrow (\mathit{CONSOLID}_{vcore}, \mathit{CONSOLID}_{mem})$
  return $(\mathit{VCoreMap}_{need}, \mathit{MemMap}_{need})$
end if
$\mathit{CPI}_{min}$ ← INFINITE
```

```
**for all** $(VCoreMap_{need}, MemMap_{need})$ **do**
    **if** $VCoreMap_{cur} \neq VCoreMap_{next}$ **then**
        $ovrhd \leftarrow$ VCoreReMapping($VCoreMap_{next}$, $VM0$)
        $ovrhd_{vcore} \leftarrow ovrhd + ovrhd_{vcore}$
    **end if**
    **if** $MemMap_{cur} \neq MemMap_{next}$ **then**
        $ovrhd \leftarrow$ MemReMapping($MemMap_{next}$, $VM0$)
        $ovrhd_{mem} \leftarrow ovrhd + ovrhd_{mem}$
    **end if**
    $(VCoreMap_{cur}, MemMap_{cur}) \leftarrow (VCoreMap_{need}, MemMap_{need})$
    $INST_{prev} \leftarrow readinst$
    $CLOCK_{prev} \leftarrow readclk$
    sleep for $window_{interim}$
    $INST_{cur} \leftarrow readinst - INST_{prev}$
    $CLOCK_{cur} \leftarrow readclk - CLOCK_{prev}$
    $CPI_{cur} = CLOCK_{cur}$ / $INST_{cur}$
    **if** $CPI_{min} > CPI_{cur}$ **then**
        $CPI_{min} \leftarrow CPI_{cur}$
        $(VCoreMap_{opt}, MemMap_{opt}) \leftarrow (VCoreMap_{cur}, MemMap_{cur})$
    **end if**
**end for**
**return** $(VCoreMap_{opt}, MemMap_{opt})$
```

5.5 Fined-grained VCore/Memory Mapping Policies

So far, the coarse-grained resource mapping approaches have been considered and discussed. Now, a couple of fine-grained mapping policies are introduced. Fine-grained mapping is intended to enhance the coarse-grained *interleaved* mappings for both vcore and memory. The fine-grained vcore mapping approach will relocate vcores without changing the *interleaved* vcore mapping style, and the fine-grained memory mapping is similarly a variation on memory interleaving.

5.5.1 Fine-grained VCore Mapping Policy

At first, the scope of the fine-grained configuration needs to be clarified. The fine-grained vcore mapping approach is considered only when CIMI is configured by the coarse-grained policy. Given that, moving vcores between different NUMA domains needs to aim for performance improvements. The performance enhancement can be realized by minimizing average memory-access distance, and/or load-balancing cache contention across the NUMA domains, balancing the Working Set Size (WSS) across the NUMA domains. More specifically, the targets are rephrased like this:

- **Reducing average memory distance:** If the portion of remote memory-accesses in a vcore is high, relocate the vcore.
- **Balancing WSS:** If the WSS is unevenly spread across NUMA domains, rebalance WSS through the vcore remapping.

Two kinds of metrics are needed for these goals:

- Distributions of NUMA domains of accessed pages per vcore:

$$d_{vcore} = (DistributionRatio_{node0}, ..., DistributionRatio_{nodeN-1})$$

where, there are N NUMA domains.

- WSS of each vcore, WSS per NUMA domain, and, WSS ratio:
  - Page access rate per memory operation of a vcore: $r'_{\cup as}(vcore)$
    where $vcore \in VCORE$, the set of all vcores. This metric is used because we expect it to have a high correlation with the WSS for each vcore.
  - WSS per NUMA domain: $r'_{\cup as}(node)= \bigcup (r'_{\cup as}(vcore))$
    where $node \in NODE$, and the set of all NUMA domains.
  - WSS ratio:
    $$(rawRatio_{node}) = \frac{r'_{\cup as}(node)}{\sum_{node' \in NODE} r'_{\cup as}(node')}$$

As d_{vcore} values are preliminary measured for the benchmarks[5], they are all evenly distributed with the *interleaved* memory mapping. As long as the accessed pages are near to each other in the guest physical address space, the *interleaved* memory mapping will make them well balanced across the NUMA domains. Therefore, the focus should be on the second target, **balancing the WSS**.

Algorithm The algorithm is designed to balance the WSS across NUMA domains. Whenever any imbalanced WSS distribution is observed, along with the *interleaved* vcore mapping being selected by NAVAR, the fine-grained policy attempts to redistribute vcores (fine-grained vcore remapping). In the main loop of NAVAR, a few changes are made:

Before:

$(VCoreMapPolicy, MemMapPolicy) \leftarrow$ DecisionTree$(metrics)$

$(VCoreMap_{need}, MemMap_{need}) \leftarrow$ ExtractMap$(VCoreMapPolicy, MemMapPolicy)$

[5]The *interleaved* memory preferred workloads are LU, CG, FLUIDANIMATE, WUPWISE, SWIM, MGRID, EQUAKE, FMA3D, APSI, BT, SP, UA, APPLU, and GAFORD. They are selected to be measured.

After:

$(\mathit{VCoreMap}_{need}, \mathit{MemMap}_{need}) \leftarrow$ FineVCoreDecisionTree($\mathit{metrics}$, $\mathit{VCoreMap}_{cur}$)

The core function, DecisionTree, is now replaced by the FineVCoreDecisionTree function. The new function contains the fine-grained vcore mapping algorithm, which is a greedy algorithm. The procedure is as follows. It first sorts vcores in the order of the WSS values, $r'_{\cup as}(vcore)$. Starting with the maximum, the vcore is located to the NUMA domain, $node$, that currently holds the minimum value of the sum of the located vcores' WSS, $minimum(Bucket_{node})$, and that also has room to include more vcores, $|\mathit{VCORES}'_{node}| < \mathit{MAX\_VCORES}'$. The maximum available number of vcores, $\mathit{MAX\_VCORES}'$, is the number of vcores mapped to each NUMA domain under the *interleaved* vcore mapping. When this iteration is finished, the vcore mapping is determined. Remember the approach is trying to enhance the *interleaved* mapping, the entire decision tree codes are used on, as shown below (the codes before the fine-grained vcore mapping algorithm begins). A few conversion functions are added to properly manage the mapping information. SplitVCoreMap and MergeVCoreMap split and merge the allocated vcore information per NUMA domain from/to the vcore map, $\mathit{VCoreMap}$.

```
FineVCoreDecisionTree($metrics^{6}$, $VCoreMap_{cur}$)
  **if** $objective$ is power **then**
    $(VCoreMapPolicy, MemMapPolicy) \leftarrow (CONSOLID_{vcore}, CONSOLID_{mem})$
  **else if** $r_{\cup as} > threshold\_LWSS$ **then**
    $(VCoreMapPolicy, MemMapPolicy) \leftarrow (INTER_{vcore}, INTER_{mem})$
  **else if** $r_{\cap as} > threshold\_CS$ **then**
    **if** $objective$ is energy and $CacheMissRate > threshold\_CS\_CacheMissRate$ **then**
      $(VCoreMapPolicy, MemMapPolicy) \leftarrow (CONSOLID_{vcore}, CONSOLID_{mem})$
```

[6]The detection framework is slightly modified to produce the new metrics: $\forall r'_{\cup as}(vcore)$, $\forall r'_{\cup as}(node)$, and $rawRatio_{node}$.

```
  end if
else
  ($VCoreMapPolicy$, $MemMapPolicy$) $\leftarrow$ ($INTER_{vcore}$, $CONSOLID_{mem}$)
end if

($VCoreMap_{need}$, $MemMap_{need}$) $\leftarrow$ ExtractMap($VCoreMapPolicy$, $MemMapPolicy$)
($VCORES_{node0}$, ..., $VCORES_{nodeN-1}$) $\leftarrow$ SplitVCoreMap($VCoreMap_{cur}$)

{the fine-grained vcore mapping algorithm begins}
if $\exists$ $node$, where $(rawRatio_{node}) > 55\%$ and $VCoreMapPolicy$ is $INTER_{vcore}$ then
  $Sorted$ $\leftarrow$ (sorted sequence of $r'_{\cup as}(vcore)$, except for the max)
  reset all $Bucket_{node}$ and $VCORES'_{node}$, where $node \in NODE$
  $Bucket_{node0}$ $\leftarrow$ max of $r'_{\cup as}(vcore)$
  add the $vcore$ info to the $VCORES'_{node0}$ vector
  for all $r'_{\cup as}(vcore)$: each of $Sorted$ in a descendant order do
    while search $minimum(Bucket_{node})$ do
      if $|VCORES'_{node}| < MAX\_VCORES'$ then
        add $r'_{\cup as}(vcore)$ to the $Bucket_{node}$
        add the $vcore$ info to the $VCORES'_{node}$ vector
        break the loop
      end if
    end while
  end for
  $VCoreMap_{need}$ $\leftarrow$ MergeVCoreMap($VCORES_{node0}$, ..., $VCORES_{nodeN-1}$)
end if
return ($VCoreMap_{need}$, $MemMap_{need}$)
```

5.5.2 Fine-grained Memory Mapping Policy

One alternative memory mapping policy is the first-touch policy, commonly used in Linux and other platforms. Preserving the *interleaved* memory mapping style, a hybrid memory mapping is considered to take both the first-touch approach and the baseline one (the modulo-base *interleaved* mapping). This approach is conditionally to take the first-touch policy for pages that are privately owned by a single vcore, while the modular-base policy is applied to the rest of the pages. To do so, the ownership status, the information on whether a page is shared or privately owned, is stored in the bitmap, OwnershipStatusBitmap. This needs to be checked regularly for every single pages, where we select and apply the memory mapping policy indicated in the PolicyBitmap. A bitmap is a sufficient data structure as long as two mapping policies are considered. When the *interleaved* memory mapping is consistently selected by NAVAR, OwnershipStatusBitmap needs to be periodically updated. If the OwnershipStatusBitmap is changed considerably, for example, $>$20% change, PolicyBitmap is reconfigured, and accordingly FirstTouchInfo for the first-touch policy needs to be flushed for the first-touch selected pages.

OwnershipStatusBitmap and the ratio OwnershipStatusBitmap is updated by the aggregator of the detection framework as long as the *interleaved* memory mapping is selected. For each vcore bitmap, $r'_{\cup as}(vcore)$, the related bit is set if the page is privately owned (touched by only one vcore). As this bitmap is kept updated, it needs to track the degree of change between the old bitmap and the new one, $oldBitmap$ and $newBitmap$. One metric to represent the degree of change is the $OwnershipStatsBitmapRatio$ which is defined as:

$$OwnershipStatsBitmapRatio = \frac{weight(newBitmap \oplus oldBitmap)}{weight(newBitmap \wedge oldBitmap)}$$

where $weight(bitmap)$ represents Hamming weight (bitmap count)[7] of the $bitmap$, and $\oplus$ and $\wedge$ refer to XOR and AND operations respectively.

Algorithm FineMemDecisionTree($metrics$, $VCoreMap_{cur}$) adds an additional decision node that determines whether it needs to update PolicyBitmap and reset FirstTouchInfo or not. PolicyBitmap determines whether the corresponding page is for the modulo-base policy or for the first-touch one. Even with $ADAPTIVE\_INTERLEAVED$, it needs to update OwnershipStatusBitmap. The configuration for PolicyBitmap is embedded in ExtractMap($VCoreMapPolicy$, $MemMapPolicy$). As it translates the mapping policy info to the mapping info, $MemMap_{need}$, PolicyBitmap is reconfigured under $ENFORCED\_ADAPTIVE\_INTERLEAVED$ as described, marking the privately owned page bit for the first-touch policy, and FirstTouchInfo is flushed. ExtractMap also needs to deliver the old FirstTouchInfo so as to find the source data correctly.

```
if $objective$ is power then
  $(VCoreMapPolicy, MemMapPolicy) \leftarrow (CONSOLID_{vcore}, CONSOLID_{mem})$
else if $r_{\cup as} > threshold\_LWSS$ then
  $VCoreMapPolicy \leftarrow INTER_{vcore}$

        $OwnershipStatsBitmapRatio \leftarrow \frac{weight(newBitmap \oplus oldBitmap)}{weight(newBitmap \wedge oldBitmap)}$

  if $OwnershipStatsBitmapRatio > threshold\_OwnershipStatsBitmapRatio$ then
    $MemMapPolicy \leftarrow ENFORCED\_ADAPTIVE\_INTERLEAVED$
  else
    $MemMapPolicy \leftarrow ADAPTIVE\_INTERLEAVED$
```

[7]The number of set bits

```
        end if
    else if $r_{\cap as} > threshold\_CS$ then
        if $objective$ is energy and $CacheMissRate > threshold\_CS\_CacheMissRate$ then
            $(VCoreMapPolicy, MemMapPolicy) \leftarrow (CONSOLID_{vcore}, CONSOLID_{mem})$
        end if
    else
        $(VCoreMapPolicy, MemMapPolicy) \leftarrow (INTER_{vcore}, CONSOLID_{mem})$
    end if
    $(VCoreMap_{need}, MemMap_{need}) \leftarrow$ ExtractMap($VCoreMapPolicy, MemMapPolicy$)
    return $(VCoreMap_{need}, MemMap_{need})$
```

5.6 Experimental Results

There are three to the experimental results. The first is the NAVAR results, evaluating how well the adaptive system configures the system for the user-driven objective. The next two parts are about the comparisons of the brute-force and of the fine-grained approaches with the NAVAR policy. All the experiments have 8 vcore/threads running on the R410 machine.

5.6.1 NAVAR Results

The NAVAR policy is designed through offline-based modeling and analysis. Since the decision tree (the model) is established by the learning phase, the model should be validated with a separate test set of benchmarks. A few test cases[8], therefore, are intentionally excluded from the learning phase and included in the validation set for use in these experi-

[8]BODYTRACK, CG, EP, FREQMINE and EQUAKE are the validation cases.

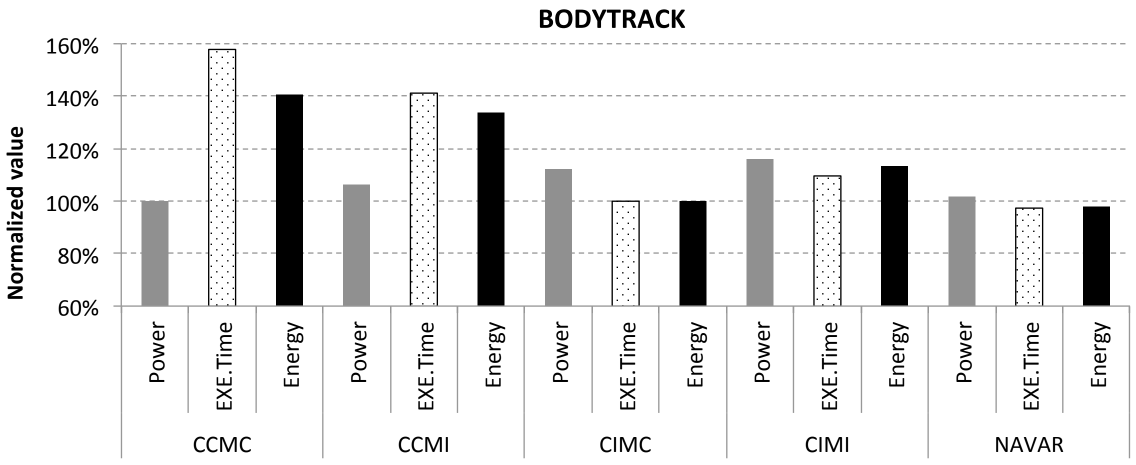


Figure 5.16: The results of BODYTRACK. NAVAR selects CCMC for the power objective, and CIMC for the energy and performance objectives.

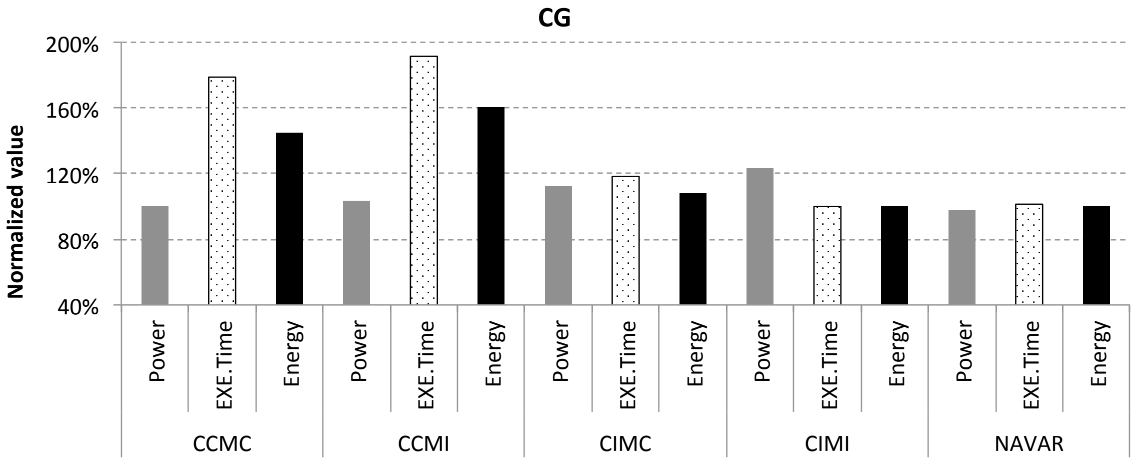


Figure 5.17: The results of CG. For the performance and energy objectives, NAVAR produces almost the same results as CIMI, and the same power by NAVAR as by CCMC.

mental results. Additionally, STREAMCLUSTER is also added even though it is included in the learning set because of its unique characteristics. Figures 5.16, 5.17, 5.18, 5.19, 5.20, and 5.21 show that NAVAR correctly finds the best configuration to produce the optimal

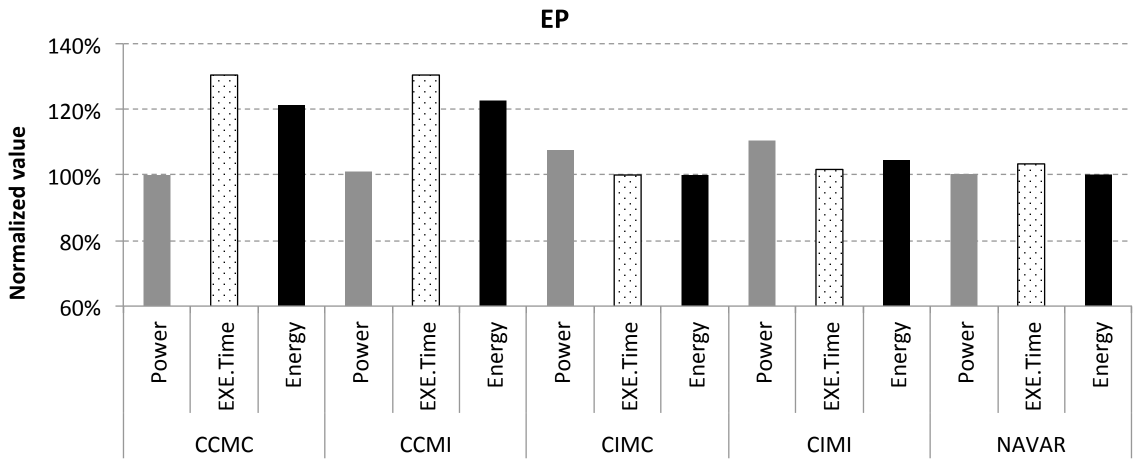


Figure 5.18: The results of EP. NAVAR selects the configurations that produce almost the same execution time and energy as the best static mapping does.

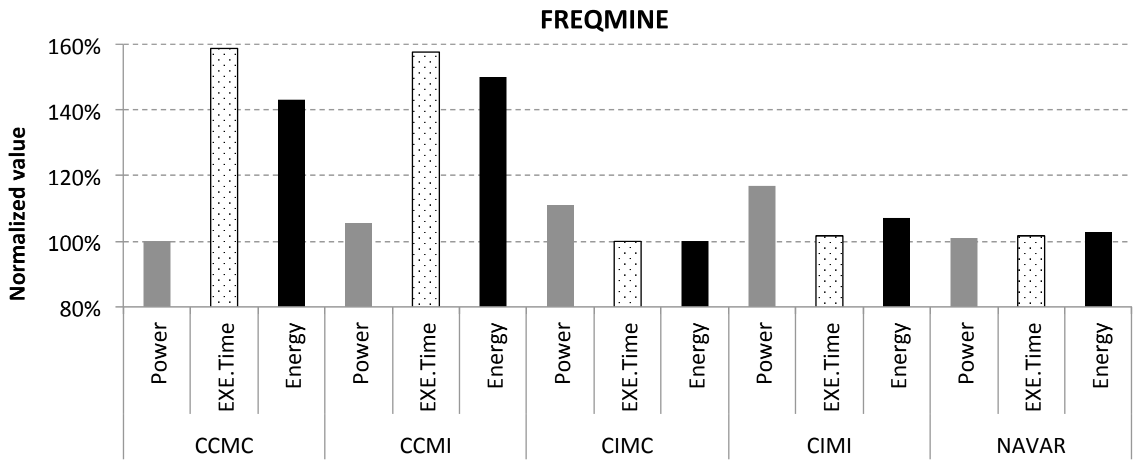


Figure 5.19: The results for FREQMINE benchmark. NAVAR configures the system to be optimized for the given objective.

results for the requested objectives. The impact of power saving from the proposed hardware mechanism, the memory power-off feature, is also included with estimates of the expected power reduction. On the test machine, the R410, the power consumed by the

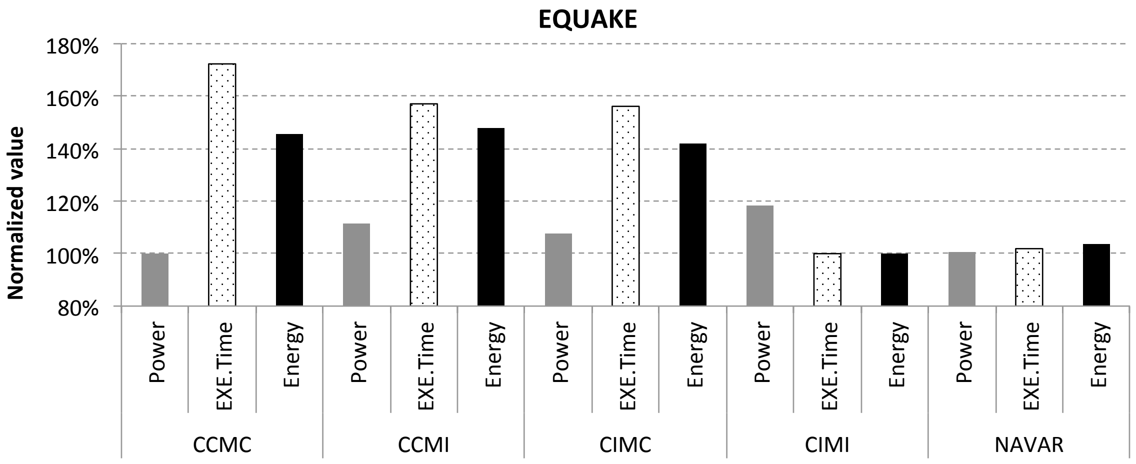


Figure 5.20: The results for EQUAKE. NAVAR produces results for the optimized performance, energy, and power respectively.

memory sub-system is relatively small compare to the entire system's power. Therefore, to distinguish memory power-off effects from noise, the CPU frequency is selectively[9] set to the lowest one; nonetheless, this configuration is fair as in all the compared cases, the static configuration results as well as the NAVAR results are equally determined with the same CPU frequency. EP and FREQMINE fairly demonstrate the energy efficiency with the *consolidated* memory over the *interleaved* mapping, saving about 4–7%, while no significant performance differences are observed between the two.

5.6.2 Brute-force vs NAVAR

Table 5.8 shows comparisons between NAVAR and the brute-force policy. The migration mechanisms, both vcore and memory migrations, suspend vcore execution. The mechanisms have some room to be optimized, nevertheless, they have limitations in that contents in memory have to be migrated on every comparison across all the vcore and memory

[9]All cases except for STREAMCLUSTER

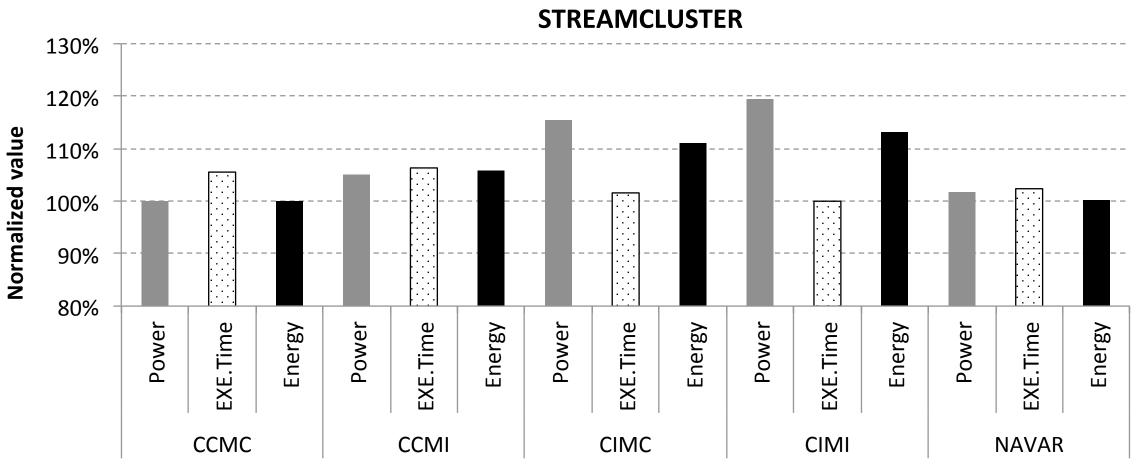


Figure 5.21: The results of STREAMCLUSTER. Different from the rest benchmarks, CCMC shows the best energy efficiency, and NAVAR matches it. The execution time that NAVAR gets is less than 3% different from the best static mapping result (CIMI).

Workload	NAVAR	Brute-force
BODYTRACK	100%	127.19%
CG	100%	147.89%
EP	100%	116.95%
FREQMINE	100%	124.18%
EQUAKE	100%	111.43%
STREAMCLUSTER	100%	120.63%

Table 5.8: Normalized execution time comparing NAVAR with the brute-force approach. Due to the significant overheads in the mechanism, the brute-force policy has a large penalty to its performance.

configurations. During the migrations, some or all of the vcores need to be suspended. At least for performance, the overall migration cost needs to be below 3% of the best execution time. According to the measured overhead, show in Table 3.1, it is quite a challenge to achieve this threshold. Considering the rate of the comparison, the whole comparison along with the relevant migrations are made every 10 seconds.

Workload	NAVAR	NAVAR with fine-grained vcore
FMA3D	100%	182.95%
WUPWISE	100%	100.02%
EQUAKE	100%	100.08%

Table 5.9: Normalized execution time of NAVAR (100%) and of the fine-grained vcore mapping. WUPWISE and FMA3D are selected for the potential opportunity of memory traffic shaping (balancing). The two cases, EQUAKE and WUPWISE, turn out to have very little vcore migrations; whereas, FMA3D has lots migrations to get performance penalties.

5.6.3 Fine-grained Approach vs Coarse-grained (NAVAR)

Since the configuration space is conservatively set to the coarse-grained level, the fine-grained approach may improve performance. The fine-grained mapping strategies taken for both vcore and memory mappings are primarily focused on the enhancement of the NAVAR policy. Unfortunately, the results do not seem to be promising. The following discussions describe the situation.

Fine-grained vcore mapping We speculate that there are two major reasons for the fine-grained vcore mapping is performed. The limited opportunity makes the overhead more visible. Most of all selected benchmarks, about 14 of the cases, have well-balanced Working Set (WS) distributions as observed in $r'_{\cup as}(node)$. The two cases[10], WUPWISE and FMA3D, appear to have occasional unbalanced WS distributions on the order of 60% vs 40% across two NUMA domains. Therefore, any improvement is the effect of about 10% reshaped memory traffic, which is very limited over the whole execution time. Also, there is a potential gap between $r'_{\cup as}(node)$ and the WSS. The false positive eventually results in unnecessary migrations that may constitute a performance penalty. Nonetheless, the limited opportunity is still true in this case, and the opportunity is possibly different

[10]FMA3D is observed to have more unbalanced working set distributions than WUPWISE.

larger scaled NUMA machines.

Fine-grained memory mapping CG and EQUAKE are compared with respect to the NAVAR results. For both performance and energy, the difference is less than 2%, not a huge penalty but also not a huge improvement. Separately, the two cases, statically applying the first-touch (with no flushes) policy and the *interleaved* vcore mapping separately, are compared with each other, and no significant difference is found. One possible reason is the limited impact of memory distance factors, whereas the memory bandwidth impact is more visible and affective.

Discussion of coarse-grained vs fine-grained A more fundamental question is whether the fine-grain configuration space is effective or not. One thought is that there is possibly some overlap between guest-side scheduling and memory management, particularly on the fine-grained level of configuration. The guest scheduler also tries to schedule and load balance threads under the SMP model, and the memory manager in the guest also maps pages in a fragmented way. These all may collectively remove the opportunities for VMM-driven fine-grained resource managements. The coarse-grained resource management, on the other hand, achieves decent and efficient benefits, and as far as the under-subscription case and the test machine are concerned, the performance gain looks to be saturated when trying to go beyond the coarse-grained configuration space.

5.7 Conclusion

This chapter has shown and compared three different approaches to addressing the resource management tradeoffs in the under-subscription case, along with various mapping strategies. Overall, the coarse-grained approach looks to be the most promising, as the brute-force approach has significant migration costs and the fine-grained approach does not achieve any visible improvement, but does increase the costs. Nonetheless, the two-dimensional configuration spaces need to be considered at the same time. Therefore, the policy needs some elaboration. NAVAR clearly demonstrates its ability to adjust the system to the best configuration, in the given configuration space, for the user-driven objective.

Chapter 6

Full-subscription

In the previous chapter, the resource management problem for the under-subscription case was discussed and NAVAR was introduced to address the problem of the vcore and memory mapping concerns. Workloads were classified, which helped to build a decision tree. The decision tree model was used for the NAVAR system and compared with the brute-force approach as well as with more fine-grained vcore and memory mapping policies. The experimental results showed that the NAVAR system with the coarse-grained mappings automatically configure a system as well as the best static configuration. This chapter extends the scope of a problem to cover the full-subscription case.

6.1 Tradeoffs and Opportunities

Context of full-subscription In the under-subscription case, one VM is assumed to run on the physical machine. In Chapter 3, a scenario for full-subscription is 2 VMs with six vcores with 16 available hardware threads. This type of workload involves a moderately utilized resource, where the system can be space-shared and does not need vcore-scheduling onto a hardware thread. Consolidating multiple VMs in one machine is a good strategy for reducing the maintenance cost. Particularly, when multiple NUMA domains are incorporated in one physical machine, each VM can be allocated to a separate NUMA domain. This resource partitioning is a conventional setup reported in recent articles [74, 62]. With this partitioning strategy under a multiple-socket multicore machine, a VM's vcores are separately consolidated into one NUMA domain and at most one vcore is served by each hardware thread. Therefore, in the full-subscription case, memory mapping is the configuration space that needs to be considered.

Tradeoffs in memory mappings The possible memory mapping cases, when there are two NUMA domains, are as follows, and they are also depicted in Figure 6.1:

- **MC^2@(NUMA_NODE)**: Map all guest pages (memory) to a single NUMA domain, one of the two domains (**N0** or **N1**[1]).
- **MCMC**: Map each Vm's pages to its matching host NUMA domain[2]. According to the resource partitioning strategy, this is the default configuration.
- **MIMC** or **MCMI**: One of the two VMs gets the *interleaved* memory mapping, while the other VM has the *consolidated* memory mapping in its host NUMA domain. Each of the first two and last two letters refers to one VM, e.g. VM0 and VM1. **MIMC** is the case that VM0 has the *interleaved* memory mapping, but VM1 has *consolidated* memory mapping
- **MIMI**: Both VMs have the *interleaved* memory mapping.

[1]N0 refers to NUMA domain 0 and N1 refers to NUMA domain 1.

[2]The NUMA domain where the VM's vcores are mapped. Also, a NUMA domain is interchangeably called node or socket.

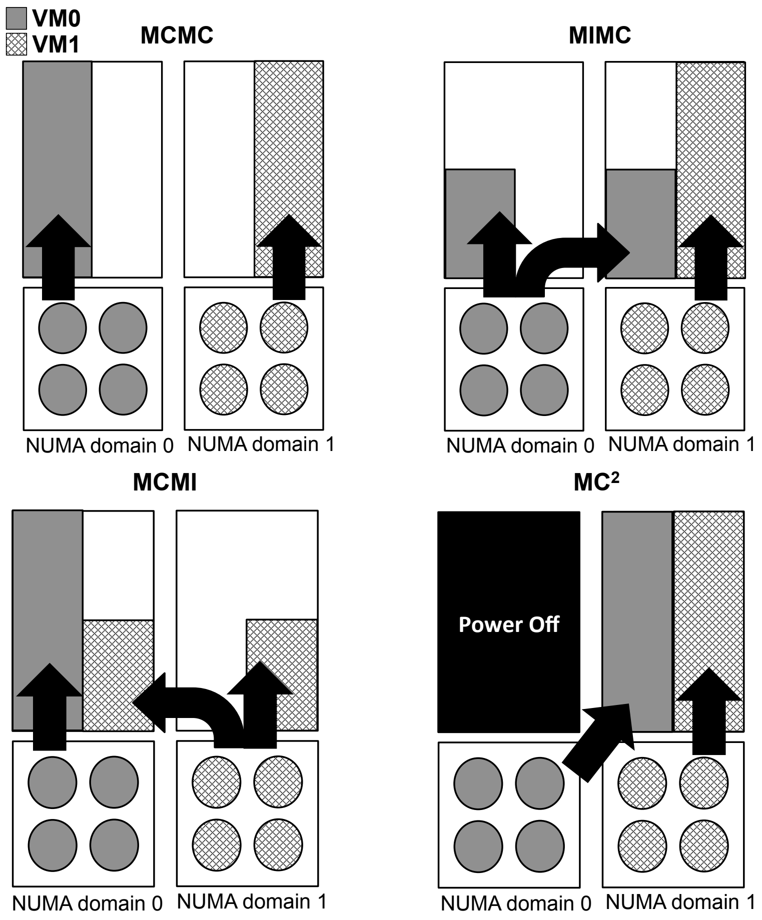


Figure 6.1: Illustration of 4 of the 6 possible memory mapping cases for the full-subscription case. MIMI and MC^2@(N0) are not shown here. It is important to understand that any MC^2 type of memory mapping makes it possible to power-off the memory of the NUMA domain that is not used.

According to the partitioning strategy, the default mapping is MCMC, which consolidates each VM's memory in the NUMA domain where the VM's vcores are mapped. This partitioning strategy isolates main memory traffic completely. However, some workloads demand more memory bandwidth than what one node can provide. MIMC or MCMI is useful particularly when two very different types of applications in VM0 and VM1 are

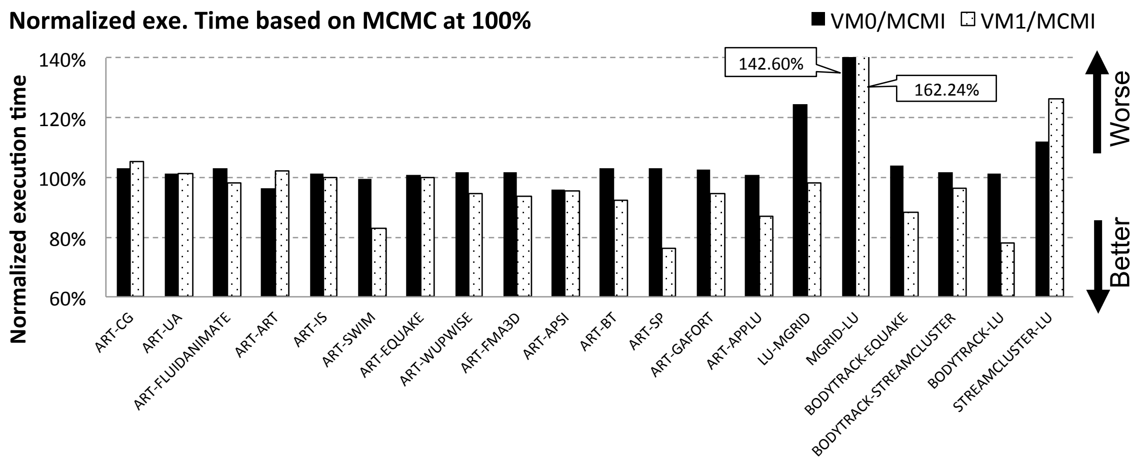


Figure 6.2: Illustration of tradeoffs in the full-subscription case for paired VMs running the noted benchmarks. The performance ratio is shown. The baseline performance is execution time of MCMC, *consolidated* memory mappings for the two VMs. The execution time of MCMI, *consolidated* memory mappings for VM0 and *interleaved* memory mappings for VM1, is divided by the execution time of the baseline for each VM. For example, BODYTRACK-LU shows that BODYTRACKS's execution time does not changed between MCMC and MCMI, whereas the execution time of LU is reduced more than 20% by using MCMI instead of MCMC. The LU benchmark clearly gains performance, while the co-runner, BODYTRACK, does not have any penalty on the performance. However, it is important to understand each pair of applications affects each other in different ways.

running. When the workload on VM0 demands more memory bandwidth but the available memory bandwidth in one node is sufficient for the workload in VM1, MIMC is a reasonable choice. MIMC provides more memory bandwidth to VM0 by allowing access to the other NUMA domain, which VM1 also accesses, which is fine when the memory traffic from VM1 is low. However, it should be checked whether MIMC raises traffic congestion in the commonly accessed NUMA domain. MIMI mapping also seems appropriate, but preliminary performance results show that MIMI does not clearly outperform MIMC or MCMI across our benchmark pairs.

When both VMs' memory bandwidth demands are small, MC^2 can be selected. The benefit of MC^2 allows us to leverage the proposed hardware mechanism for powering-off the memory in one NUMA domain. Because MC^2 is not expected to increase execution time for both VMs, this memory mapping will also save energy. Figure 6.2 shows MCMI performance compared to MCMC (the default memory mapping) for different pairs of benchmarks.

Test scalability and an approach to modeling Previously, when tradeoffs were addressed for resource mapping concerns, all possible configurations for our benchmarks were compared preliminary. The reason why all cases were measured was to try to capture characteristics of workload as much as possible[3]. The cases for under-subscription are limited, so it is possible to configure each static configuration and to measure execution times accordingly. However, when running multiple VMs for full-subscription, it is hard to measure all possible cases. Two applications interact with each other in accessing main memory; therefore, each *pair* should be treated as a separate case. Given about 30 different applications, the number of combinations comes to be more than 400 cases, $\binom{30}{2} = \frac{30 \times 29}{2 \times 1}$. It is laborious to do modeling based on execution time results of all possible configuration cases[4]. Nonetheless, the preliminary performance results as well as classifications for under-subscription are useful. Applications are individually characterized in regard to memory traffic. This model can fit into a broaden interaction model that predicts the benefits of the *interleaved* memory mapping and so on. The following sections describe the procedures to build an offline-based model, and then metrics to be measured are described. The revised detection mechanism is also discussed. After that, policy design and its experimental results follow.

[3]This assumes the model-based prediction approach, which is pursued for full-subscription case as well.
[4]The case refers to a learning set. A validation set should not be measured.

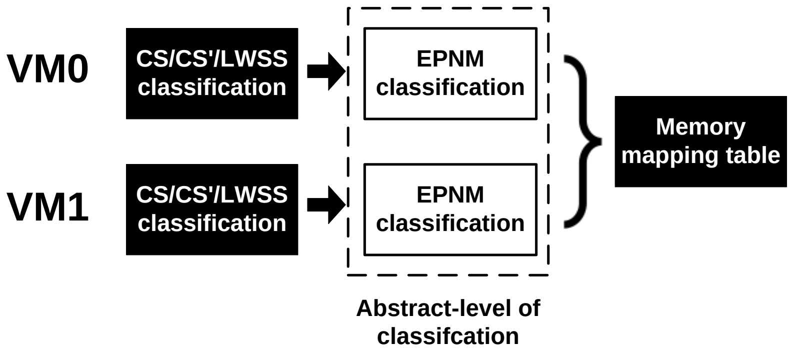

VM0
CS/CS'/LWSS classification
EPNM classification
VM1
CS/CS'/LWSS classification
EPNM classification
Memory mapping table
Abstract-level of classifcation

Figure 6.3: Illustration of a new classification. This classification is called EPNM classification. EPNM stands for the four classes: Expandable, Protected, Normal, and Mergeable. The CS/CS'/LWSS (the adjusted CS and LWSS classification) class, explained in Section 6.2.2, can populate the memory mapping table. EPNM classification is inserted for comprehensiveness.

6.2 Offline Modeling

Considering memory migration cost and the fact that the brute-force policy does not work for under-subscription, the model-based prediction approach is more applicable, here. Classification that is reproducible during runtime is presented along with measurable metrics. Accordingly, the detection framework is modified and the extension will be discussed in the next section. At first, reusing the under-subscription classification, CS and LWSS, is attempted. However, as the overall aspects of the full-subscription cases are studied, it turns out that the four classes are not sufficient for the six memory mapping choices. Therefore, a different classification of workloads is needed to address the memory mapping problem for the full-subscription problem.

6.2.1 Classification

As it is hard to cover all cases, the combinations of all workloads, selected test cases are chosen and then workloads are intuitively classified to select the best memory mapping. As shown in Figure 6.3, this classification is used to build the memory mapping table. The following includes the descriptions of the classes and classification itself. Classifiers will be further clarified by relating a EPNM class[5] to a CS/CS'[6]/LWSS class.

Expandable class As shown in Figure 6.2, the performance effects of the *interleaved* memory by (MIMC or MCMI) are different depending on the characteristica of workloads. If a $>5\%$ performance benefit is observed with the *interleaved* memory mapping, the workload is classified as Expandable. With the *interleaved* memory mapping, this benchmark also has energy-efficiency. On the flip side, when both two VMs are of the Expandable type, when one VM is given an *interleaved* memory mapping and the other VM is given the *consolidated* memory mapping, both workloads used significantly underperform. In fact, this case indicates that MCMC should be used when the two VMs are in the Expandable class. Therefore, even when a workload[7] is classified as Expandable, it is conditional on the other workload whether to use *interleaved*. Besides the case where a co-runner is in the Expandable class, there is another class that disallows the VM to have *interleaved* memory mapping, the Protected class.

Protected class This class does not take advantage of the *interleaved* memory mapping, but it does not allow its co-runner to use its memory bandwidth, particularly when the co-runner is the Expandable class. Protected benchmarks, such as STREAMCLUSTER, worsen then execution time when a co-runner has the *interleaved* memory mapping.

[5]EPNM class refers to one of the Expandable, Protected, Normal, and Mergeable classes.

[6]CS' class is a new class used for the full-subscription case. Detail can be found in Section 6.2.2.

[7]As one application is assumed to run on a VM, an application or a workload refers to a VM here.

VM0	VM1	Memory mapping (VM0, VM1)
Megeable	Mergeable	(MC^2@N0, MC^2@N0)
Protected or Expandable	Protected or Expandable	(MC,MC)
Normal or Protected	Mergeable	
Normal or Protected	Normal	
Expandable	Mergeable or Normal	(MI,MC)

Table 6.1: Memory mapping table for the EPNM classes. The sequence of VM0 and VM1 is interchangeable. When either one of VMs is Expandable and the other VM is either Mergeable or Normal, the Expandable VM takes MI while the other VM has MC. MC^2 is thought to be equivalent to either of one of two mappings as the remote memory-access effect for Mergeable workloads is small.

Mergeable BODYTRACK and BLACKSCHOLES are example where even with MC^2 there is no significant performance penalty. The execution time of MC^2 over MCMC is 0.22% lower for BODYTRACK and 2.48% higher for BLACKSCHOLES, which demonstrates that MC^2 is a viable choice for energy for these benchmarks. MCMC consumes about 5% more power than MC^2. We refer to workloads as being in the Mergeable class.

Normal The Normal class includes those workloads which neither get better performance by having the *interleaved* memory mapping nor get worse by the co-runner's having an *interleaved* memory mapping. Nonetheless, they produce enough traffic that the MC^2 option results in a performance penalty.

Mapping strategy Given the EPNM classes, the mapping strategy is illustrated in Table 6.1. Note that the order of classes in each pair is interchangeable for the indicated memory mapping. There are two distinct cases that are mapped to MC^2 or MIMC (or MCMI). Otherwise, workloads under full-subscription are MCMC-mapped. MIMI is not

EPNM class	Workloads
Expandable	SWIM, EQUAKE, MGRID, APPLU, LU, SP, GAFORT, WUPWISE, FMA3D
Protected	STREAMCLUSTER
Normal	BT, CG, UA, IS
Mergeable	BLACKSCHOLES, BODYTRACK, FERRET, FACESIM, FREQMINE, FLUIDANIMATE, RAYTRACE, SWAPTION, DEDUP, VIPS, ART GALGEL, EP, AMMP, X264, CANNEAL, APSI

Table 6.2: Preliminary EPNM classifications for the selected benchmarks, based on observations of static mapping results. The complete classification is shown in Table 6.4.

taken into account because both the Mergeable and Normal classes have no significant performance benefit from the *interleaved* memory mapping. Therefore, MIMI is, at best, equal to MIMC in performance.

6.2.2 Revising CS and LWSS

CS and LWSS classification is already available from measurable data, but it does not exactly match with the EPNM classification. The CS and LWSS classification, therefore, needs to be revised.

Among the four classes produced from CS and LWSS classification, some maps to one class of EPNM. The CS&!LWSS class is equivalent to the Protected class, the !CS&!LWSS class is equivalent to the Mergeable class, and the workloads in the CS&LWSS class are also classifiable as the Expandable class. These three equivalents are intuitively explicable in terms of memory traffic demands. The CS&LWSS class indicates that both numbers of pages either frequently accessed ($r_{\cap as}$) or just touched ($r_{\cup as}$) are high, so the type matches with the Expandable class. On the opposite side, the !CS&!LWSS class has low memory traffic, as reflected in the two metrics, $r_{\cap as}$ and $r_{\cup as}$, so it is Mergeable. Finally, the CS&!LWSS class has high memory traffic observable by $r_{\cap as}$, and so, it is also classified

Configuration variable	Setup
Machine	Dell PowerEdge R410 machine
Resource mapping	CCMC
Probing duration	varying 0.25x 0.5x 1x of base cycles count (2,394,000,000)
Scanning interval	10,000,000 counts of write instructions
Number of vcores	eight threads

Table 6.3: Summary of setup for the sensitivity check for the probing duration. Only the probing duration is changed, compared to the previous chapter.

as the Protected class.

!CS&LWSS is the challenge, because some workloads in the !CS&LWSS class are in the Expandable class and others are in the Normal class. For example, CG, UA and IS are in the Normal class, and GAFORT, EQUAKE, WUPWISE, and FMA3D are in the Expandable class. However, all six benchmarks are classified to the !CS&LWSS class. To complete the transition from the CS and LWSS classification to the EPNM class, the !CS&LWSS class needs to be split. With the separator, all workloads in the CS&LWSS class and some of those in the !CS&LWSS class, called CS'&LWSS, are in the Expandable class, and the rest of the cases in the !CS'&LWSS class are in the Normal class. This adjusted classification is called the CS/CS'/LWSS classification. The separator is based on metrics when the probing duration is adjusted as discussed in the next section.

6.3 Adjusted Detection Framework

We now consider how to tune the detection framework to appropriately detect the !CS&LWSS class. The CS classification already separates the LWSS class into the CS&LWSS and !CS&LWSS classes. Since the expected partition to separate !CS&LWSS in Expandable and Normal is in the middle of !CS&LWSS, the threshold for CS depicted in Figure 5.13 could be lowered to include more benchmarks. However, in the metric, $r_{\cup as}$, WUPWISE

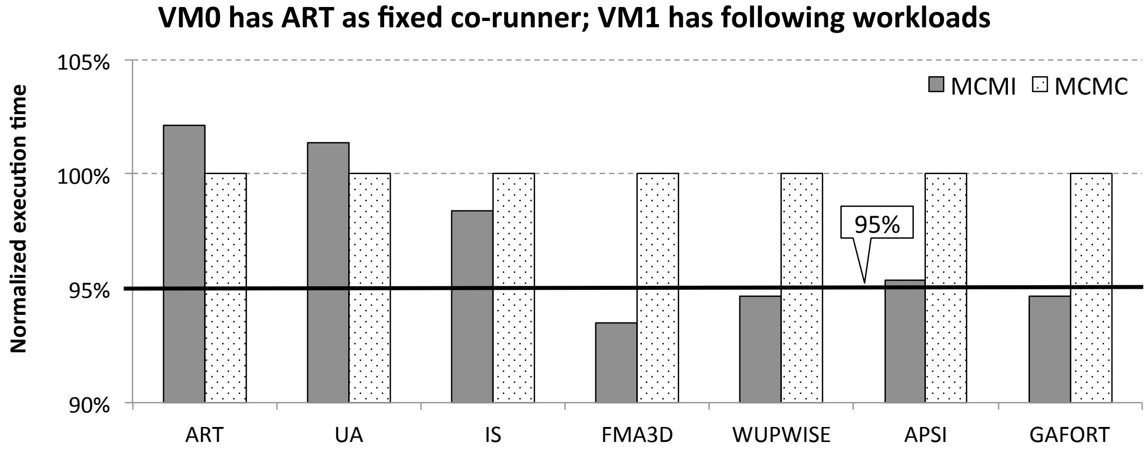


Figure 6.4: Performance comparison between the CCMI and CCMC mappings. WUPWISE, for example, has a 5.4% execution time reduction with the *interleaved* memory mapping over the *consolidated* memory mapping, while the co-runner best uses the *consolidated* memory mapping consistently. Note that execution time variation of the co-runner is not shown here.

and FMA3D, for example, which are classified to the Expandable class remain indistinguishable with the other benchmarks such as UA and IS that are classified to the Normal class. Another way to partition is to adjust the probing duration. The longer the probing duration, the bigger the gap between $r_{\cup as}$ and $r_{\cap as}$. For the following show the sensitivity of the probing duration to $r_{\cap as}$ especially for benchmarks that are in the !CS&LWSS class.

6.3.1 Sensitivity Check for Probing Duration

In Section 5.2.3, the probing duration is briefly mentioned as one of two parameters to control the detection framework. The probing duration can be used to control the relationship between $r_{\cap as}$ and $r_{\cup as}$. When the probing duration is very short, $r_{\cap as}$ tends to be close to $r_{\cup as}$, but when we increase the probing duration, $r_{\cap as}$ begins to diverge from $r_{\cup as}$ and its value decreases. Given this, another sensitivity check is needed using the setup shown in

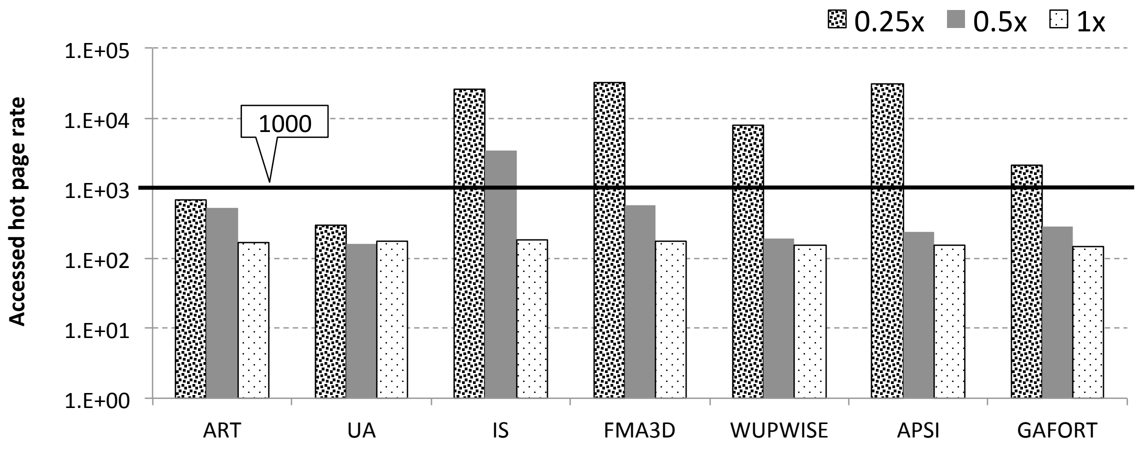


Figure 6.5: Sensitivity in the probing interval for $r_{\cap as}$. Variations in the measured value are illustrated, as changing the probe duration. "1x" represents the baseline duration, and the two cases are the measured value with the shorter probing durations.

Table 6.3. Figure 6.4 shows the selected benchmarks' execution times comparing between MIMC and MCMC. These benchmarks are in the !CS&LWSS class. For comparison, Figure 6.5 illustrates variations of $r_{\cap as}$ as the probing duration is changed. The baseline, shown as "1x", is used to classify the CS class. With the baseline probing duration, the workloads have no differences; however, after reducing the probing duration by four times, they become quite different. With this "0.25x" probing duration, the measured values have some correlation with the performance results. Therefore, a threshold value can be set to classify workloads separting Normal and Expandable. The threshold value can be 1000. In addition, the cache miss rate[8] is checked to see if it is beyond a 10% threshold. IS turns out to be the incorrectly classified case that is detectable with the cache miss rate check. When a workload has $r_{\cap as}$ and cache miss rate values that are beyond the two respective thresholds, they are classified as the CS' class; otherwise, workloads are classified to the !CS' class. CS'&LWSS ties to Expandable, and so does !CS'&LWSS to Normal. With this new

[8]Last-level cache miss rate

CS/CS'/LWSS Class (EPNM class)	Workloads
CS'&LWSS (Expandable)	FMA3D, WUPWISE, APSI, GAFORT, BT, LU, SP, APPLU, SWIM, EQUAKE, MGRID, CANNEAL, FLUIDANIMATE
CS&!LWSS (Protected)	STREAMCLUSTER
!CS'&LWSS (Normal)	UA, IS, CG, ART
!CS&!LWSS (Mergeable)	BLACKSCHOLES, FERRET, FACESIM, FREQMINE, SWAPTION, DEDUP, VIPS, GALGEL, AMMP, X264, EP, BODYTRACK, RAYTRACE

Table 6.4: The CS/CS'/LWSS classification results. This classification expands on the previous CS and LWSS classification used for under-subscription. Each of previous four classes is directly mapped to the four class of the EPNM classification with a change to how the CS class is defined (CS'). With that, the EPNM classification is now clarified with thresholds in the measurable metrics.

classifier, the CS/CS'/LWSS classification is completed. Table 6.4 lists the classification results along with the matched class of the EPNM classification.

6.3.2 Algorithm

In regards to the detection mechanism, aggregating the bitmap values in the middle of a probe needs to be added to the detection framework as shown in the last chapter. This adjusted detection framework is very similar to the version of Chapter 4. Here, however, we change the probing duration rather than the scanning interval. Lastly, it should be mentioned that the probing process occurs separately for each VM.

```
Init(aggregator): [Invoked at startup on aggregator]
  SetTimer($0.25T$, SetAggregate)
  $Phase = 0$
  **for all** vcores $i$ **do**
    $EnableScan_i = 1$
    Force vcore $i$ to run InitVcore($i$)
  **end for**
InitVcore($i$): [Invoked at on vcore $i$]
  $hot\_accessed_i = \{k : 0 \ldots m-1\}$
  $accessed_i = \{k : 0 \ldots m-1\}$
  Set PMU exception for number of store operations to trigger Scan($i$).
Scan($i$): [invoked when PMU exception occurs]
  **if** $EnableScan_i = 1$ **then**
    **for all** present shadow (or nested) PTEs on vcore $i$ **do**
      $k$ = DeriveGuestPhysicalPageNumberFrom(PTE)
      $curacc_i = \emptyset$
      **if** PTE.accessed **then**
        $curacc_i = curacc_i \cup \{k\}$
      **end if**
      PTE.accessed=0
    **end for**
    $hot\_accessed_i = hot\_accessed_i \cap curacc_i$
    $accessed_i = accessed_i \cup curacc_i$
  **end if**
SetAggregate(aggregator): [invoked when $T$ expires on aggregator]
  **if** $Phase = 0$ **then**
    $Phase = 1$
    quickAggregate(aggregator)
    SetTimer($0.75T$, SetAggregate);
  **else**
```

```
        Aggregate(aggregator)
    end if

quickAggregate(aggregator)
    for all vcores $i$ do
        $hot\_accessed\_per\_store_i = hot\_accessed_i$
        $accessed\_per\_store_i = accessed_i$
    end for
    $r_{\cap as\_s} = \frac{1}{n}\sum_{i=0}^{n-1} |hot\_accessed\_per\_store_i|$
    $r_{\cup as\_s} = \frac{1}{n}\sum_{i=0}^{n-1} |accessed\_per\_store_i|$

Aggregate(aggregator)
    for all vcores $i$ do
        $EnableScan_i = 0$
        $hot\_accessed\_per\_store_i = hot\_accessed_i$
        $accessed\_per\_store_i = accessed_i$
    end for
    $r_{\cap as} = \frac{1}{n}\sum_{i=0}^{n-1} |hot\_accessed\_per\_store_i|$
    $r_{\cup as} = \frac{1}{n}\sum_{i=0}^{n-1} |accessed\_per\_store_i|$
```

6.4 Policies

Since the brute-force policy was not successful for the under-subscription case, it is not attempted here. One root cause of the failure was that the memory migration cost is huge, much higher than the vcore migration overhead. As memory migration is the key mechanism for the full-subscription case, no improvement on the brute-force approach is expected. Now, the NAVAR policy for the full-subscription is described. It has the same overall structure as the NAVAR policy designed for the under-subscription case. Additionally, a fine-grained memory mapping policy for the full-subscription case is presented. The experimental results can be found in the next section.

6.4.1 NAVAR for Full-subscription

As mentioned, the overall structure of the NAVAR policy has not changed much from that described in the last chapter. Most changes for the full-subscription case are made in the decision tree function (DecisionTreeFS). Pseudocode follows.

```
$exe\_state \leftarrow$ INIT_STATE
$interim\_scale \leftarrow 1$
$tsc_{state\_init} \leftarrow$ readtsc; $tsc_{prev} \leftarrow$ readtsc
$ovrhd_{probe} \leftarrow 0$; $ovrhd_{vcore} \leftarrow 0$; $ovrhd_{mem} \leftarrow 0$
$MemMapPolicy \leftarrow MC^2@N0$

while 1 do
    $ovrhd_{probe} \leftarrow$ Probing($metrics$)
    $MemMapPolicy \leftarrow$ DecisionTreeFS($metrics$)
    $MemMaping_{need}(VM0, VM1) \leftarrow$ ExtractMap($MemMapPolicy$)
    if $MemMap_{cur}(VM0, VM1) = MemMap_{need}(VM0, VM1)$ then
        $interim\_scale\_mode \leftarrow$ NEED_SCALE_UP
    else
        for all $VMID \in \{VM0, VM1\}$ do
            if $MemMap_{cur}(VMID) \neq VM0MemMap_{need}(VMID)$ then
                $ovrhd \leftarrow$ MemReMapping($MemMap_{need}(VMID), VMID$)
                $ovrhd_{mem} \leftarrow ovrhd + ovrhd_{mem}$
            end if
        end for
        $MemMap_{cur}(VM0, VM1) \leftarrow MemMap_{need}(VM0, VM1)$
        $interim\_scale\_mode \leftarrow$ NEED_RESET
    end if
    Interim($ovrhd_{probe}, ovrhd_{vcore}, ovrhd_{mem}$)
end while
```

DecisionTreeFS($metrics$) immediately returns MC^2@N0 when the power objective is requested. For the other objectives, it compares the metric values to the thresholds. Accordingly, it finds the EPNM classification for each VM, $MappingClass_{VMID}$. The EPNM classification results are compared, and then one memory mapping configuration, $MemMapPolicy$, is selected through the predefined memory mapping table.

```
if $objective$ is power then
    $MemMapPolicy \leftarrow MC^2@N0$
    return $MemMapPolicy$
end if

for all $VMID \in \{VM0, VM1\}$ do
    if $r_{\cup as} > threshold\_LWSS$ then
        if $CacheMissRate > threshold\_CacheMissRate$ and $r_{\cap as} > threshold\_CS'$ then
            $MappingClass_{VMID} \leftarrow Expandable$
        else
            $MappingClass_{VMID} \leftarrow Normal$
        end if
    else
```

```
        if $CacheMissRate > threshold\_CacheMissRate$ and $r_{\cap as} > threshold\_CS$ then
            $MappingClass_{VMID} \leftarrow Protected$
        else
            $MappingClass_{VMID} \leftarrow Mergeable$
        end if
    end if
end for

switch $(MappingClass_{VM0}, MappingClass_{VM1})$
case $(Mergeable, Mergeable)$:
    $MemMapPolicy \leftarrow MC^2@N0$
case $(Expandable, Mergeable)$ :
case $(Expandable, Normal)$ :
    $MemMapPolicy \leftarrow MIMC$
case $(Mergeable, Expandable)$ :
case $(Normal, Expandable)$ :
    $MemMapPolicy \leftarrow MCMI$
default:
    $MemMapPolicy \leftarrow MCMC$
end switch
return $MemMapPolicy$
```

6.4.2 Fine-grained Memory Mapping Policy

A fine-grained memory mapping policy is now described for a fine-grained *interleaved* memory mapping. Similar to the parallel work in the previous chapter, the objective for this fine-grained approach is to enhance the performance of the described NAVAR policy. Basically, this approach tries to find an alternative *interleaved* memory mapping, to the modulo-n approach (the baseline *interleaved* memory mapping). Two principles are taken into account for partitioning pages to be mapped to the different NUMA domains. One principle is based on the assumption that pages shared by many vcores are likely to be hot pages. Filtering hot pages is important in order to prioritize pages to be placed in the local node (***highly-shared pages placed in the local node***). Notice that vcores are assumed to be consolidated to a socket in the full-subscription situation. Another principle is that memory-access traffic needs to be evenly split to leverage the memory bandwidth of multiple sockets (***load-balance***). To realize these principles, additional metrics need to be measured. Two key metrics are $LocalPagesBitmap$ and $LocalPageRatio$, each of which respectively reflects the principles.

The following detection mechanism executes after the aggregation process finishes. Each iteration in the loop searches pages that are shared by the number of vcores, $NumOwners$. The loop begins to search pages that are accessed by all vcores. There pares are identified to the hot pages, which are located in the local node as following the first principle (***highly-shared pages placed in the local node***). At the same time, the overall pages need to be distributed across all NUMA domainsfor the second principle (***load-balance***). Therefore, hot pages are searched and $LocalPagesBitmap$ is established until the number of searched pages are equal or less than the threshold, the number of all accessed pages divided by the number of NUMA domains.

```
$accessed\_per\_store \leftarrow \bigcup_{i=0}^{n-1} accessed\_per\_store_i$
$LocalPagesBitmap \leftarrow accessed\_per\_store$
$accessed\_per\_store\_Set \leftarrow \bigcup_{i=0}^{n-1} \{accessed\_per\_store_i\}$
$LocalPageRatio \leftarrow 1$; $NumOwners \leftarrow MaxVCore$
```

```
**while** $LocalPageRatio > 1/(NumofNodes)$ and $NumOwners > 0$ **do**
    $LocalPagesBitmap \leftarrow$ FilteroutNumOwners($accessed\_per\_store\_Set$,
    $NumofOwners$)
    $LocalPageRatio \leftarrow |LocalPagesBitmap|/|accessed\_per\_store|$
    $NumOwners \leftarrow NumOwners - 1$
**end while**$newLocalPagesBitmap \leftarrow LocalPagesBitmap$
```

FilteroutNumOwners($accessed\_per\_store\_Set$, $NumOwners$) sets the bits for pages that are shared by the given number of vcores, $NumOwners$.

FineMemDecisionTreeFS($metrics$) returns selected memory mapping information. Again, this fine-grained approach tries to enhance the coarse-grained approach (the NAVAR policy). Therefore, most procedures are maintained from DecisionTreeFS($metrics$). The following pseudocode reflects the changed portion, which is right after the memory mapping switch statement. $LocalPagesBitmap$ is supposed to be updated at every probe. What the new code does is to check the changed ratio on the $LocalPagesBitmap$ between the $oldLocalPagesBitmap$ and $newLocalPagesBitmap$. The ratio, $LocalPageBitmapRatio$, is defined as follows:

$$LocalPagesBitmapRatio = \frac{weight(newLocalPageBitmap \oplus oldLocalPageBitmap)}{weight(newLocalPageBitmap \wedge oldLocalPageBitmap)}$$

where $weight(bitmap)$ represents Hamming weight (bitmap count)[9] of the $bitmap$, and $\oplus$ and $\wedge$ refer to XOR and AND operations respectively. To differentiate the fine-grained *interleaved* memory mapping from the modulo-n based coarse-grained *interleaved* memory mapping, the following notations are used here:

- **fineMI**: The fine-grained *interleaved* memory mapping. For example, MCfineMI

[9]The number of set bits

refers to the *consolidated* memory mapping for VM0 and the fine-grained *interleaved* memory mapping for VM1.

- **fineMI_Enforced**: Re-map pages for **fineMI**, according to the (updated) $LocalPagesBitmap$. fineMIMC_Enforced refers to refreshing the fine-grained *interleaved* memory mapping for VM0, while the *consolidated* memory mapping is configured for VM1.

If the ratio is above the threshold[10], the policy forces the memory mapping configuration to be fineMIMC ($fineMIMC\_Enforced$) or MCfineMI ($MCfineMI\_Enforced$), otherwise, the memory mapping selected by the original procedure is preserved.

```
...
switch ($MappingClass_{VM0}$, $MappingClass_{VM1}$)
case ($Mergeable$, $Mergeable$):
    $MemMapPolicy \leftarrow MC^2@N0$
case ($Expandable$, $Mergeable$) :
case ($Expandable$, $Normal$) :
    $MemMapPolicy \leftarrow fineMIMC$
case ($Mergeable$, $Expandable$) :
case ($Normal$, $Expandable$) :
    $MemMapPolicy \leftarrow MCfineMI$
default:
    $MemMapPolicy \leftarrow MCMC$
end switch
if $MemMapPolicy = MCfineMI$ or $MemMapPolicy = fineMIMC$ then
    if $LocalPagesBitmapRatio > threshold\_LocalPagesBitmapRatio$ then
```

[10]Currently set to 20%

```
        if $MemMapPolicy = MCfineMI$ then
            $MemMapPolicy \leftarrow MCfineMI\_Enforced$
        else
            $MemMapPolicy \leftarrow fineMIMC\_Enforced$
        end if
    end if
    $oldLocalPagesBitmap \leftarrow newLocalPagesBitmap$
else
    $oldLocalPagesBitmap \leftarrow 0$
end if
return $MemMapPolicy$
```

6.5 Experimental Setup and Results

For the full-subscription case, the memory mapping is the only configuration to be considered. The fact that there is only one configuration dimension does not necessarily make the full-subscription case easier than the other cases with their multiple dimensions. Since full-subscription involves to have multiple VMs, the number of mapping cases is larger than that of the memory mapping cases for under-subscription. The choice at the coarse-grained level is whether memory channels are shared for performance and/or energy. The default memory mapping is configured to separately consolidate each VM's pages on each host node; in other words, MCMC is the default configuration. The following are the experimental results for the NAVAR policy and the fine-grained memory mapping approach.

Workload (VM0, VM1)	EPNM class	Mapping prediction
(EQUAKE, FREQMINE)	(Expandable, Mergeable)	MIMC
(EQUAKE, APSI)	(Expandable, Expandable)	MCMC
(LU, ART)	(Expandable, Normal)	MIMC
(LU, MGRID)	(Expandable, Expandable)	MCMC
(EP, FREQMINE)	(Mergeable, Mergeable)	MC^2
(CG, STREAMCLUSTER)	(Normal, Protected)	MCMC

Table 6.5: Predicted memory mapping for the test scenarios, according to the memory mapping table and the classification results.

6.5.1 NAVAR Result

Six pairs of workloads, LU-ART, LU-MGRID, EQUAKE-FREQMINE, EQUAKE-APSI, EP-FREQMINE, and CG-STREAMCLUSTER, are selected for the validation set. Due to the interactions in shared memory channels and controllers, each pair is treated as a distinct case. These cases were not included in the preliminary measurements from which the classification was developed. The experimental tests have two purposes. First, the correctness of the offline-based model (the decision tree) needs to be validated. Second, the adaptive system performance for the fulfillment for the objective, the configurability through the mechanism, and the overhead controllability also have to be checked.

Validation of offline modeling Table 6.5 shows six pairs of test sets. It also includes the expected EPNM classification results for the given pairs and the relevant memory mapping predictions, according to the predefined memory mapping table shown Table 6.4. First, comparing all memory mapping cases with the static mapping will validate the correctness of the predictions. Notice that the mapping strategy is established based on treating energy and performance objectives in the same way. Except for the MC^2 memory mapping, the memory mapping configuration itself does not result in noticeable instantaneous power changes. Therefore, energy and performance are aligned with each other.

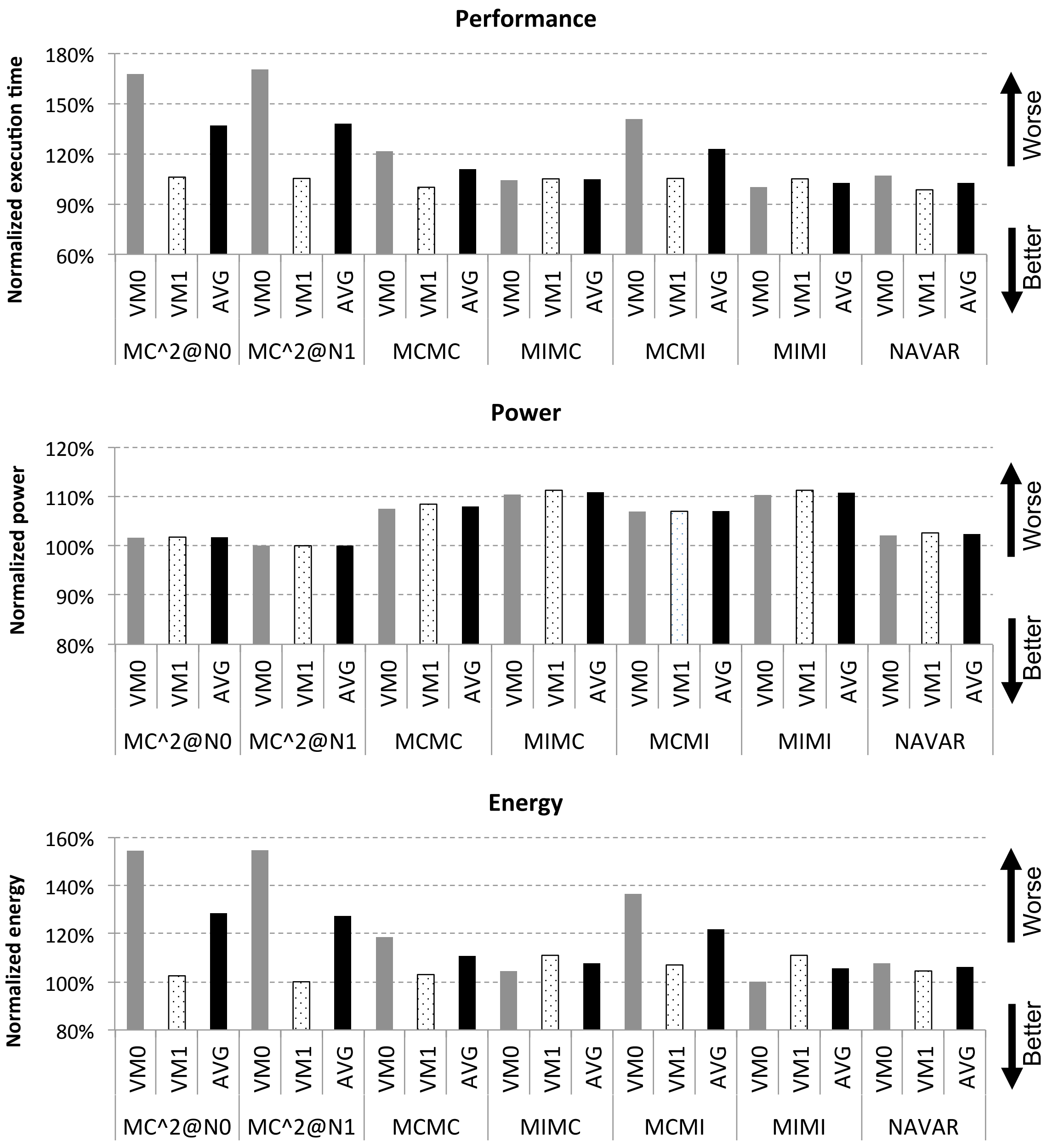


Figure 6.6: Results of LU and ART for the three different objectives. The NAVAR results are compared with all possible static configurations (6 cases). Each objective value is normalized to the minimum value of static configurations for VM0, VM1 and the average (AVG) respectively. This format is all the same for the following result figures.

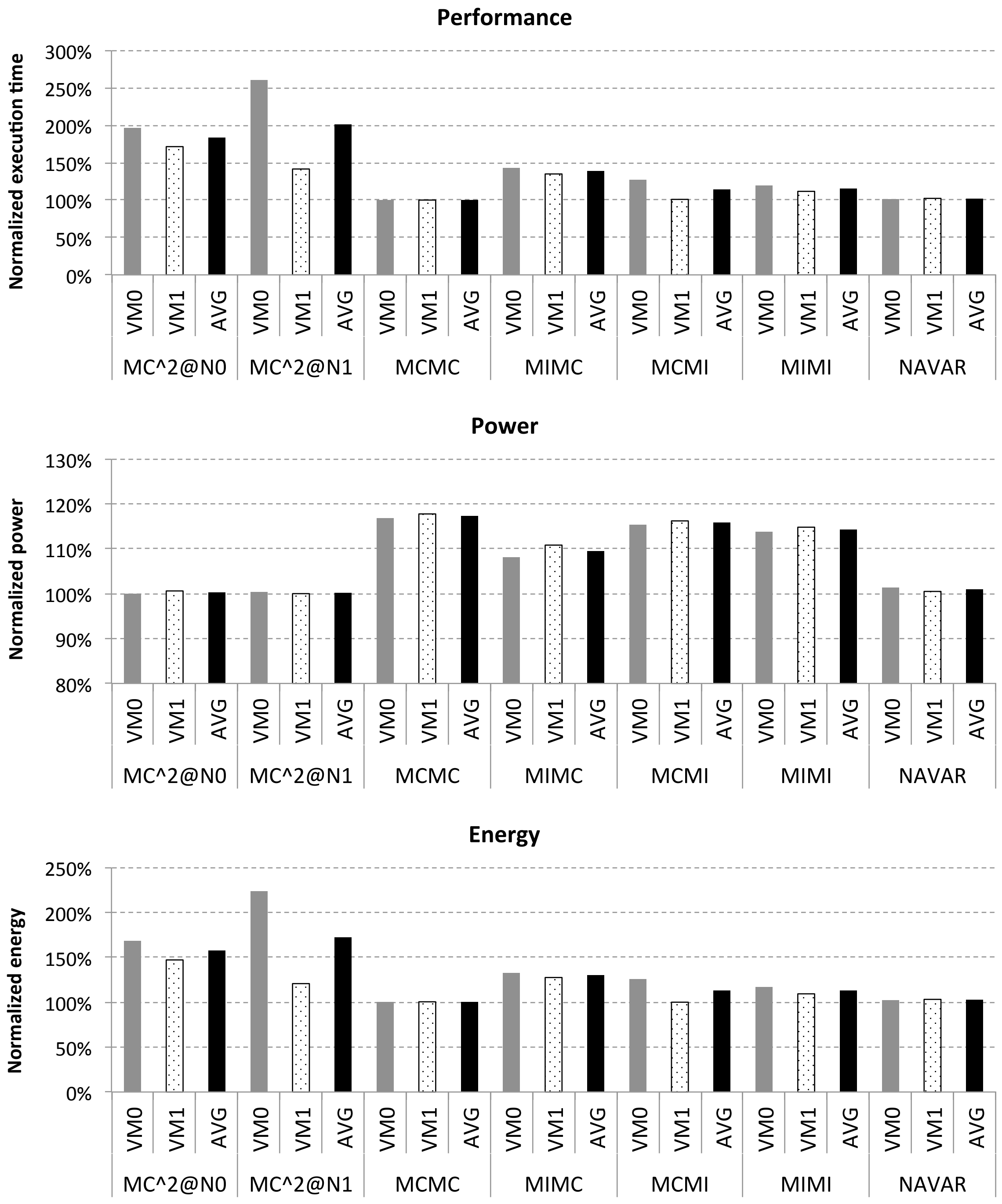


Figure 6.7: Results of LU and MGRID

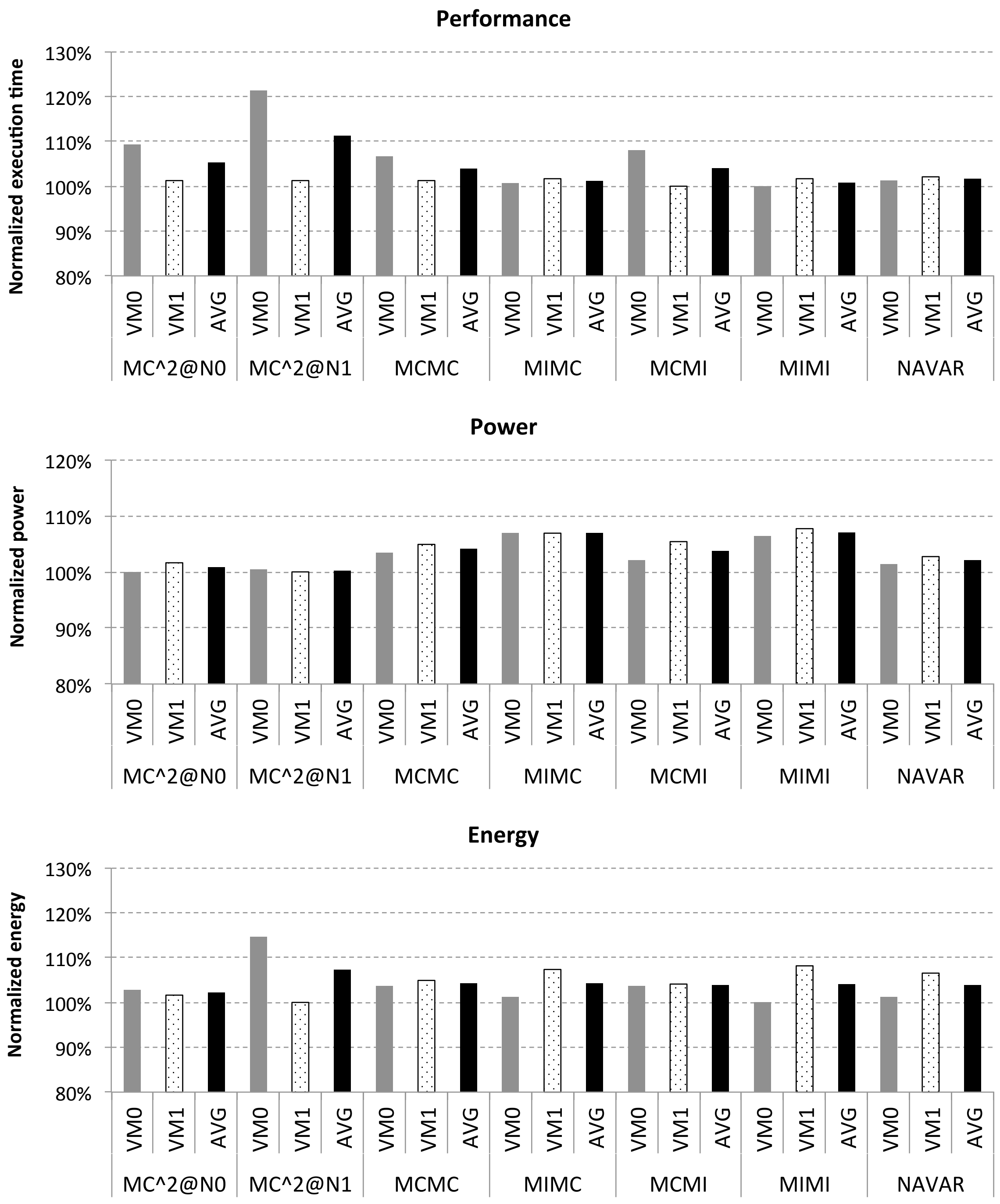


Figure 6.8: Results of EQUAKE and FREQMINE

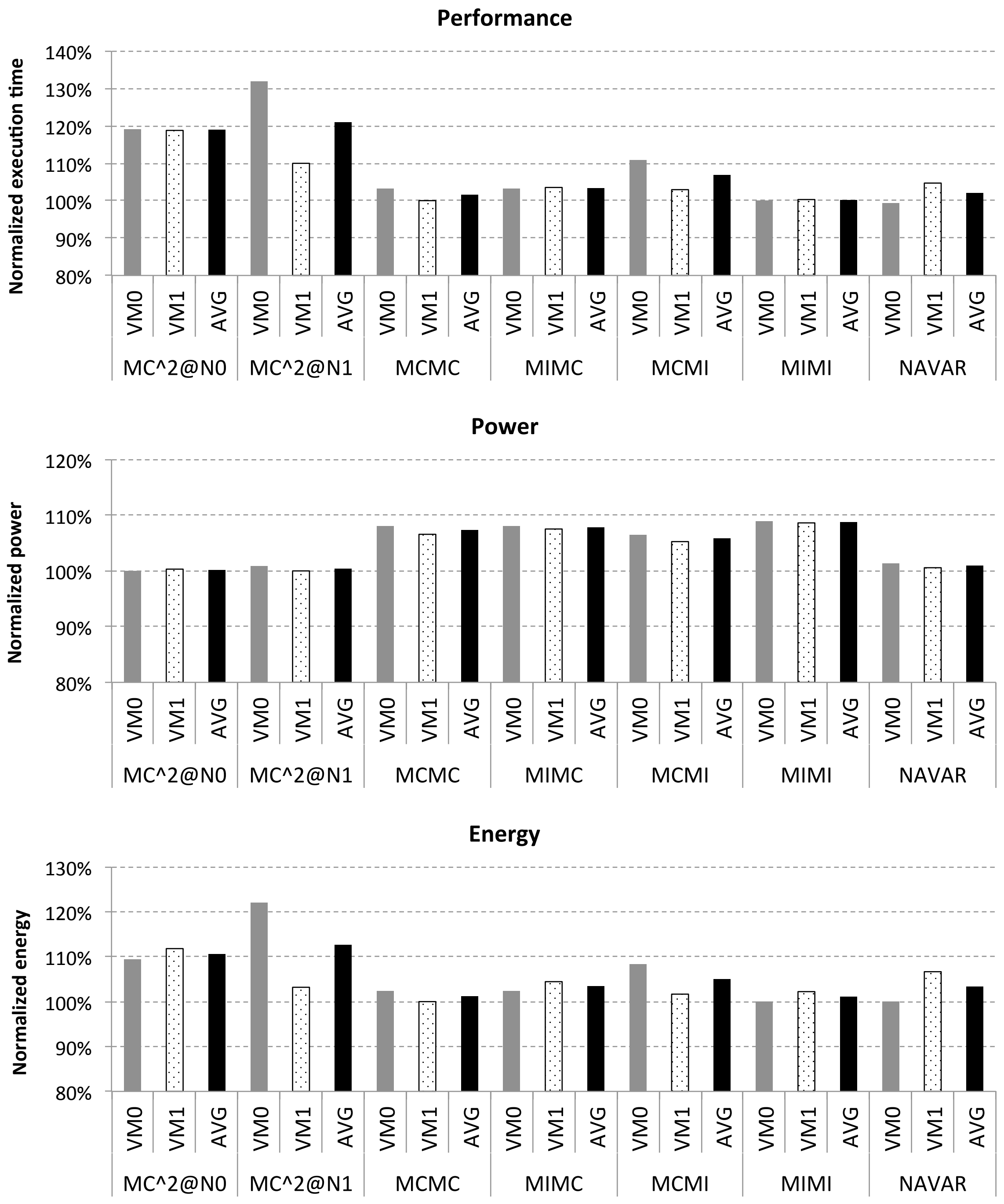


Figure 6.9: Results of EQUAKE and APSI

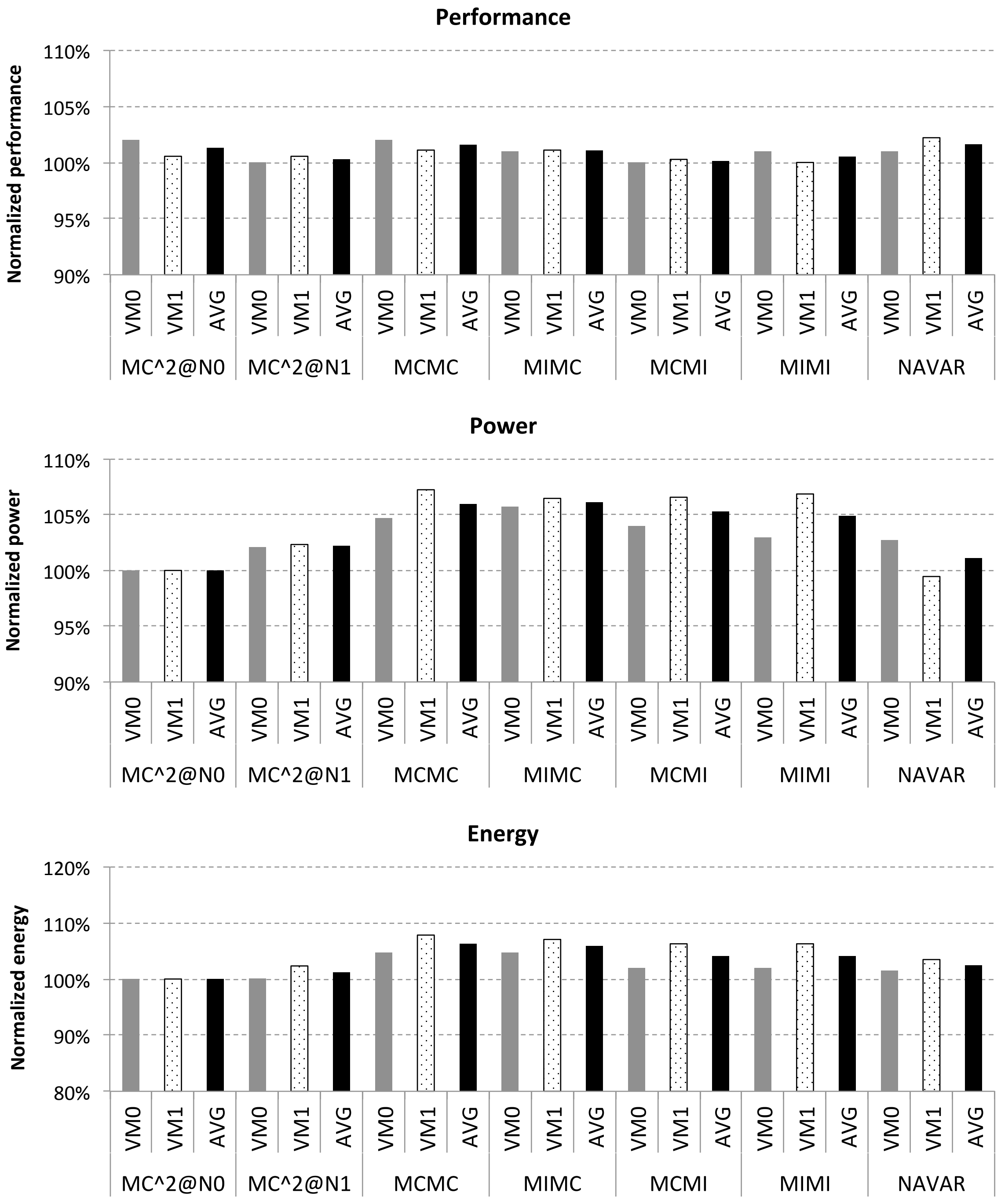


Figure 6.10: Results of EP and FREQMINE

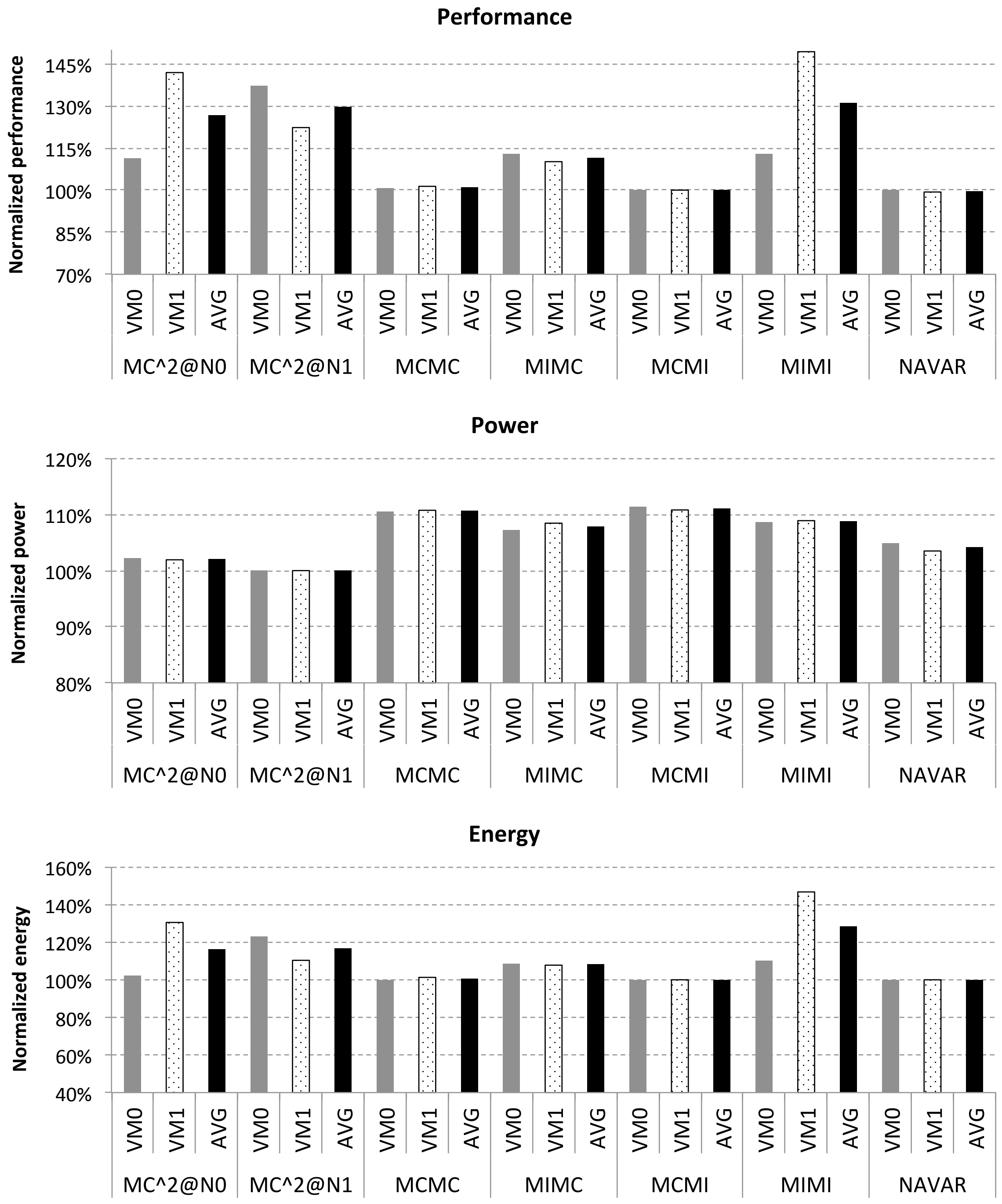


Figure 6.11: Results of CG and STREAMCLUSTER

Additionally, power is always better for the MC^2 memory mapping. As shown in Figure 6.6, 6.7, 6.8, 6.9, 6.10 and 6.11, the best static results for both performance and energy are indeed aligned with the predictions shown in Table 6.5. MIMI looks to be a viable option for most cases except for the CG-STREAMCLUSTER pair. However, MIMI is not an outstanding option over the other selectable memory mapping options in terms of performance. The decision tree, then, is still valid.

NAVAR performance results The second part of the evaluation is whether the NAVAR system works properly or not. For the six cases, NAVAR successfully selects the best mapping. For the full-subscription case, MCMC is set by the default. NAVAR increases performance about 10% and saves about 5% energy, at most, over the default mapping. For the arbitrary static mapping cases, NAVAR achieves up to 80% execution time reductions and 50% energy savings. For the power objective, the MC^2 memory mapping should be selected; thus, the question is on how much NAVAR increases average power over the static result of the MC^2 mapping. Most cases have less than 3% increased average power over the static mapping. The CG-STREAMCLUSTER case has 4% overhead due to the relatively short execution time. Each memory migration consumes more power during the migration and so needs to be amortized. Typically, about 5% additional power is spent on the memory migration than on the normal operation with the current memory migration implementation. This happens only one time for the power objective, since MCMC is taken by the default. The increased portion of power can be compensated for if the execution time takes a relatively longer.

Workload	NAVAR	Fine-grained Approach
EQUAKE-FREQMINE	100%	98.72%
LU-ART	100%	102.78%

Table 6.6: Comparison between the NAVAR and the fine-grained approaches. The execution time of the fine-grained approach (***the smaller, the better***) is normalized to the execution time result of the NAVAR system.

6.5.2 Fine-grained Policy

The NAVAR system as well as the offline-based decision tree are evaluated through the validation test set. Now, the fine-grained approach is discussed. The key point to check is improvement compared to the NAVAR policy. The fine-grained policy is selectively applied to the cases that need the MIMC or MCMI memory mapping, the EQUAKE-FREQMINE and LU-ART workload pairs. The modulo-n-based and the sharing degree-based *interleaved* memory mapping approaches do not show any noticeable difference as can be seen in Table 6.6. Considering the differences and the commonalities between the two approaches, it is likely that the memory-access bandwidth factor is more important than the memory-access distance factor. The memory-access distance impact is not at the level that makes for observable overall performance differences.

6.6 Conclusion

In this chapter, the full-subscription case was investigated. The most case assumes multiple VMs are running and raises the memory mapping problem to be important. In the full-subscription situation, the need to change vcore mapping and scheduling configuration disappears. Almost all hardware threads are taken by vcores, which prevents the *interleaved* vcores. Therefore, the memory mapping becomes the key factor to change the system performance, energy, and power. The test machine has two NUMA domains, and two VMs are assumed to be running. Then six different memory mapping cases are available. Without the adaptive mapping solution, the conventional resource mapping is resource partitioning in which each of the VMs is allocated to an entire NUMA domain. With this resource mapping approach, the static configuration does not guarantee optimization of memory-access traffic paths. The NAVAR system does so by adaptively selecting one of the possible memory mappings. The selection is at least almost equal to the best static mapping. NAVAR improves performance and energy over the conventional resource mapping (MCMC) by 10% and 5% respectively according to the test results. The next chapter addresses the over-subscription cases to extend the coverage of NAVAR.

Chapter 7

Over-subscription

Until now, a one-to-one mapping between vcores and hardware threads has been assumed, and therefore the vcore-scheduling concern has not arisen. In this chapter the scope of the problem is broadened to cover the over-subscription case. In contrast to the previous cases, this case not only has many-to-one mappings between vcores and hardware threads, but also involves many vcores in the system, and this requires vcore scheduling. A coarse-grained vcore mapping choice is not available. The subsequent sections include discussions of the opportunities and tradeoffs in the over-subscription cases, the gang-/co-scheduling implementation to support them, the NAVAR policy design, and the experiment results.

7.1 Tradeoffs and Opportunities

We now discuss tradeoffs in two configuration dimensions, memory mapping and vcore-scheduling. We begin with preliminary studies to select benchmarks for investigating the over-subscription case.

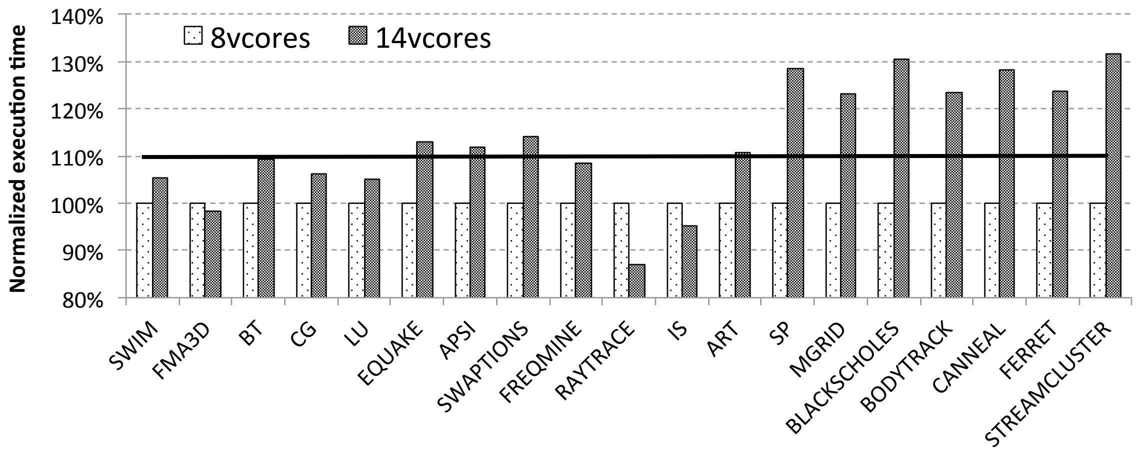


Figure 7.1: Preliminary performance study of the benchmarks. The purpose is to find the applicability of using 14 threads, by comparing execution times between 14 threads (vcores) and 8 threads (vcores). Each case has only one active VM in the test system (R410).

7.1.1 Selection of Workloads

Figure 7.6 shows the execution time differences between 8 vcores and 14 vcores for each benchmark. Each workload thread is mapped to a single vcore, which is mapped to a distinct hardware thread. Given 16 hardware threads, each VM can fully utilize the hardware threads. They also use the *interleaved* memory mapping and the *interleaved* vcore mapping for 8 vcores. Because cache contention may increase as we increase the number of threads in one socket, the cases with more than a 15-pecent-increased execution time are not suitable for the 14 threads configuration. Further, when multiple workloads are running with vcore-scheduling, CG, IS, and ART become unstable in execution, and some other short benchmarks have fluctuations in their execution times. They are excluded. Benchmarks with a short runtime present no issues for execution, but high fluctuations are not good for use as experimental test cases. The fluctuation is assumed to happen when the

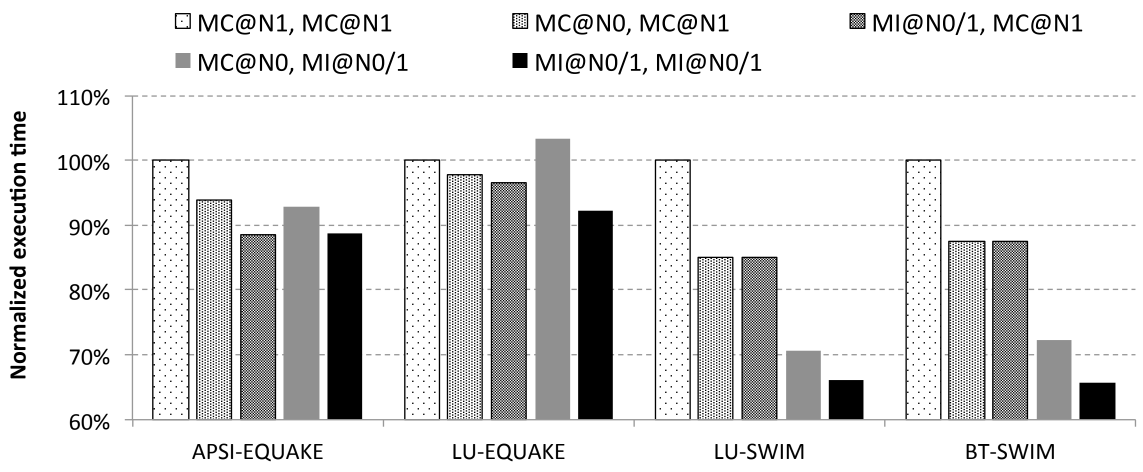


Figure 7.2: Tradeoffs in performance for the 5 possible memory mapping cases, with vcores fixed to the *interleaved* mapping. Each VM has 14 vcores and the hardware threads mapped the vcore mapping of the two VMs are the same; thus, in each hardware thread, two vcores from different VMs run. The *interleaved* memory mapping for both VMs gives the best performance across the test cases.

co-runners have longer run-times and include multiple phases of operation. The workload with a short run-time, then, interacts with the different phases of the co-runner when it iteratively run. Three types of benchmark are excluded from the over-subscription test cases, those that have large performance degradation cases with large number of threads unstable execution when sharing hardware threads with co-runners, and fluctuations in run-time with co-runners.

It is important to understand that the issue these excluded benchmark have is possibly testbed- and platform-specific. The exclusion here aims to minimze noise in experiment results. Five benchmarks, APSI, EQUAKE, SWIM, LU, and BT, are focused in the following experiment.

7.1.2 Tradeoffs of Memory Mapping

Memory mapping effects and tradeoffs are now discussed. Figure 7.2 shows performance variations across different memory mappings. Note that this setup of 14 vcores for each VM with 16 available hardware threads is maintained in this chapter. As seen in the results, there are memory mapping-driven performance impacts and the impacts vary by workloads. Nonetheless, the *interleaved* mapping for both VMs consistently outperforms the other four possible memory mappings. The memory bandwidth effect has been already discussed for the under-subscription and the full-subscription cases. With the fixed *interleaved* vcore mapping, the *consolidated* memory mapping does not provide benefit over the *interleaved* memory mapping. The comparison between the CIMC and the CIMI mappings shown in Chapter 5 offers insight for the analysis of the test results. Once vcore mapping is fixed to the *interleaved* mapping, the *consolidated* memory mapping limits memory-access bandwidth and increases memory-access distances, which are both negative impacts. Furthermore, memory bandwidth demands tend to be increased in the over-subscription case, where workloads in multiple VMs produce more memory traffic. With that, the *interleaved* memory mapping for all VMs is the best option agnostic to the workload. Therefore, there are few tradeoffs in memory mapping.

Since the *consolidated* memory mapping on one NUMA domain saves power with the proposed hardware mechanism, the *interleaved* memory mapping needs to be compared for energy consumption. More details can be found in Section 7.4. The conclusion about memory mapping concerns for both performance and energy is the static configuration with the *interleaved* memory mapping for all VMs is preferrable.

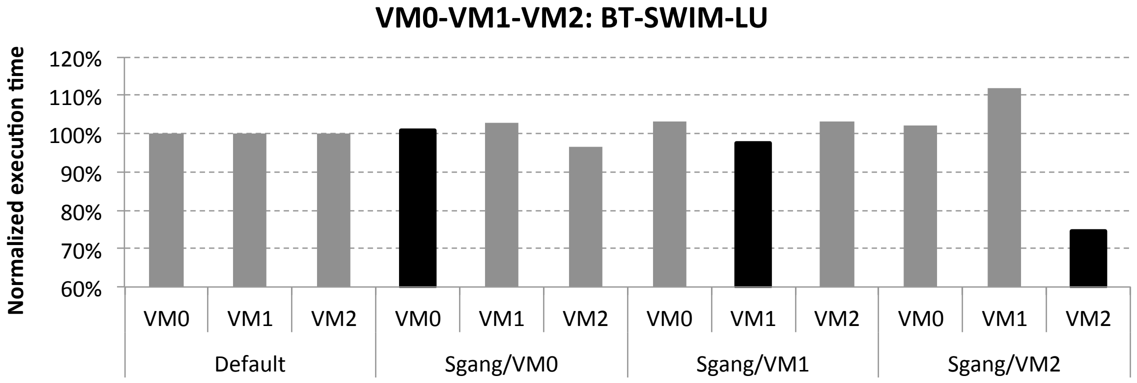


Figure 7.3: Preliminary results for gang-scheduling effects. The initial implementation described in Chapter 3, called strict gang-scheduling (Sgang) here, is used. Three VMs run with 14 vcores for each VM (42 vcores total). The vcore mapping is same for all VMs. The 14 vcores are evenly split across the two NUMA domains of the test machine (R410). Each application's execution time is normalized to the default case (the case without any gang-scheduling). The results shown in Figure 7.4 and 7.5 are also produced from the same setup. LU (VM2) gets 25% reduced execution time when gang-scheduled.

7.1.3 Tradeoffs of Gang-scheduling

Figure 7.3, 7.4, and 7.5 show gang-scheduling tradeoffs. The initial gang-scheduling implementation, the strict gang-scheduler (Sgang) described in Chapter 3, is used. Each figure compares four scheduling cases, including the default case, which does not include gang-scheduling for any VM. The other three cases apply the gang-scheduling to only one VM exclusively. For clarification, one VM is gang-scheduled to check the gang-scheduling effects, while the others are not. As shown, the gang-scheduling effects depends on the workloads. EQUAKE and LU show consistent performance benefits from gang-scheduling across different cases. Therefore it is important to adaptively select the right VM to gang-schedule during runtime to optimize performance and energy of the system. Different from memory mapping, the scheduling choice has significant tradeoffs. Before we talk about the policy design, the scheduling mechanism needs to be improved. The next section explains

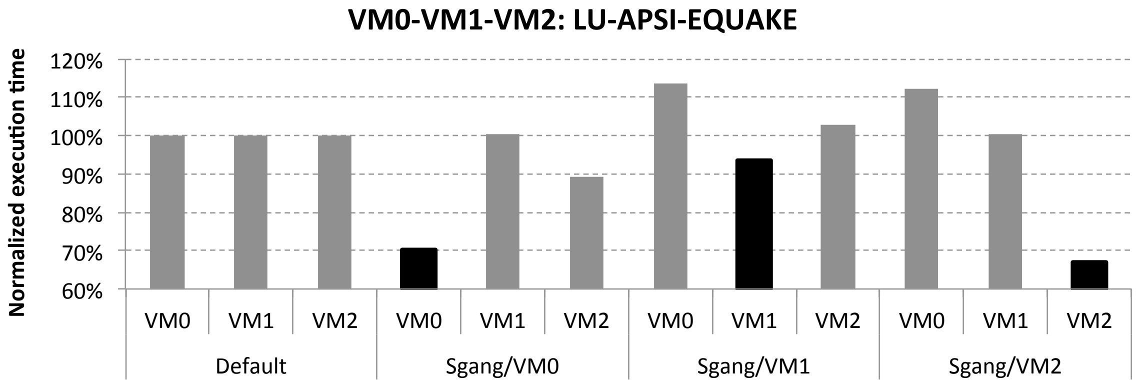


Figure 7.4: Gang-scheduling tradeoffs. LU and EQUAKE have significant performance benefit with gang-scheduling.

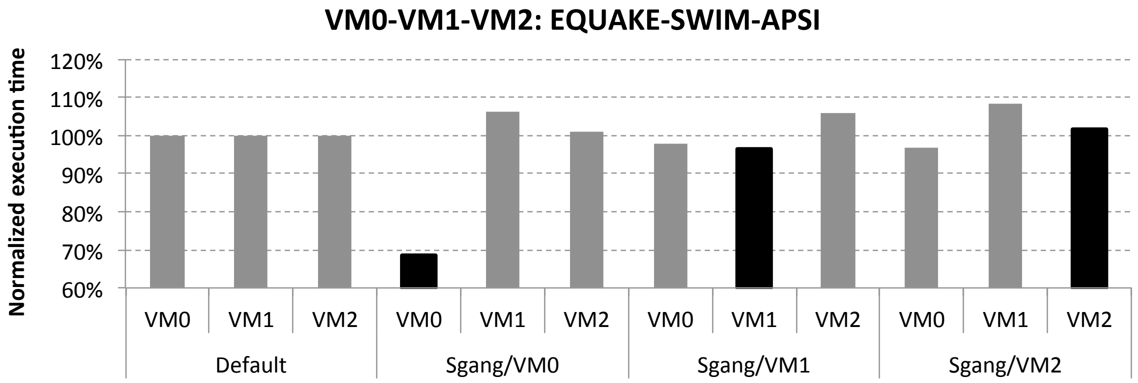


Figure 7.5: Tradeoffs of gang-scheduling. EQUAKE gets about a 30% execution time reduction with gang-scheduling.

the vcore-scheduling mechanism and the revisions.

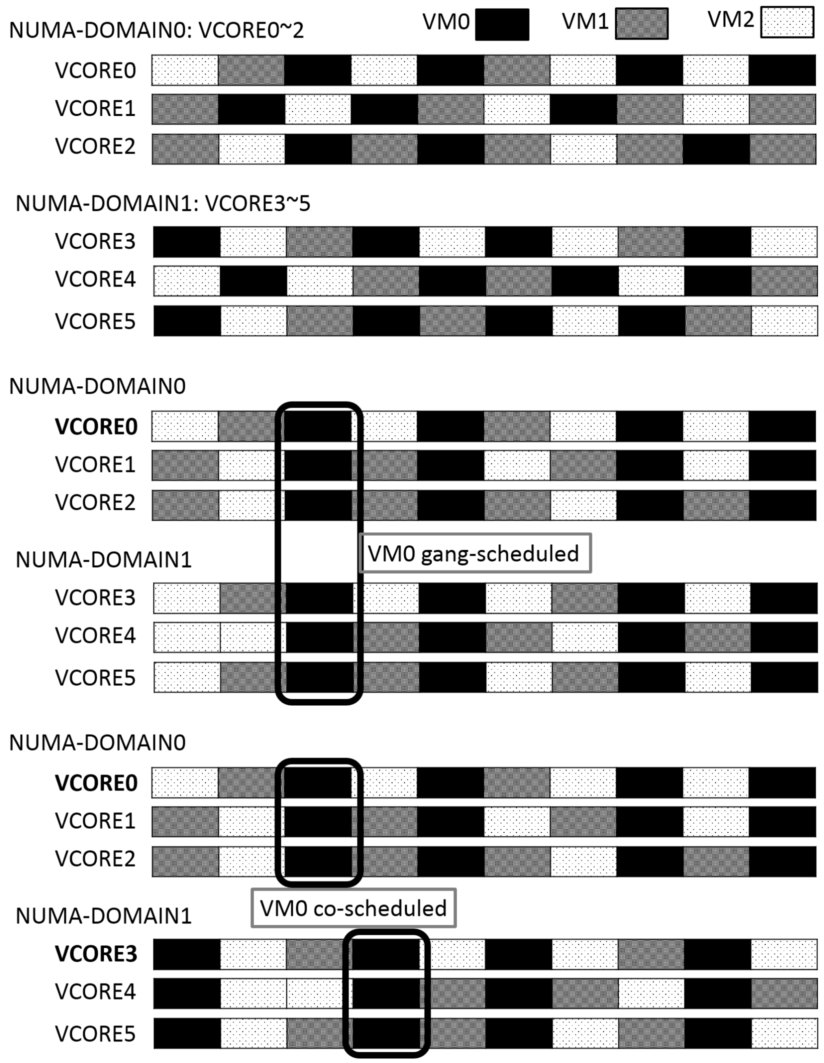


Figure 7.6: Illustration of the NUMA-aware co-scheduling by comparing with default scheduling with the gang-scheduling. The default scheduling model does not have VM-level context information. The gang-scheduling model strictly groups all of vcores to run in the same time slices. Gang-scheduling is known to cause some side effects like CPU fragmentation and priority inversion, while improving concurrent workloads by reducing synchronization waiting times and/or cache contention. The co-scheduling mechanism tries to address the tradeoffs of gang-scheduling.

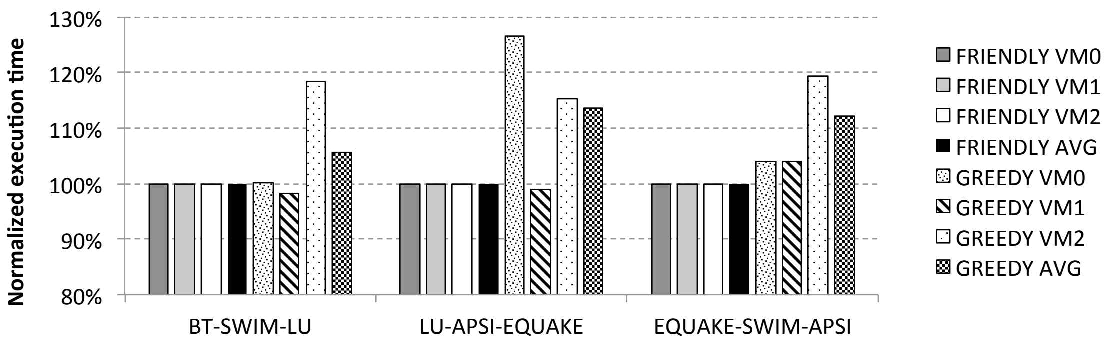


Figure 7.7: Comparison of GREEDY and FRIENDLY on the performance, without gang-scheduling. Each case shows three workloads on the VMs: VM0-VM1-VM2 in a sequence. In the case of BT-SWIM-LU, BT runs on top of VM0.

7.2 NUMA-Aware Co-scheduling

This section describes the vcore-scheduling implementation. Before I design the policy, the vcore-scheduling mechanism was iteratively revised to improve its performance impact. Finally, I developed the NUMA-aware co-scheduling mechanism. The memory mapping and vcore mapping mechanisms also have some effect on performance, but they are quite different from the scheduling mechanism as described in Section 3.3.3. Once vcore and memory mappings are (re-)configured, the performance impact is no longer driven by these mechanisms. However, the vcore-scheduling mechanism continues to affect performance because it continues to make decisions. The scheduling mechanism has been iteratively revised in two ways from the initial version described in Section 3.3.3. One way has the goal of reducing side effects such as CPU fragmentation and priority inversion. The other is more virtualization platform-specific.

7.2.1 Unconditional Yield Effect

The Palacios VMM has two different yield functions: the conditional yield function (CondYield) and the unconditional yield function (UnCondYield). CondYield is called on every exit, and the function checks current time slice expiration. The gang-scheduler implementation shown in Section 3.3.3 is entirely in CondYield. On the contrary, UnCondYield yields unconditionally, and the function is called mostly by the halt exit handler (HLTExtHandler). In the HLTExtHandler, there is a loop that keeps on yielding via CondYield until any type of interrupt is raised. The purpose of this loop is to quickly deliver interrupts to the VM's guest when they arise. In the over-subscription situation, when more than one vcore shares one hardware thread, it is correct to keep on calling CondYield to allow another vcore to run, but performming my suffer.

UnCondYield calls one of the host yield functions. The host Linux kernel provides `schedule()` and `schedul_timeout(timeout)`. CondYield calls `schedule()` by default which we refer to as the GREEDY policy. The other host yield function, `schedul_timeout(timeout)`, puts the calling thread on a wait queue for the given time interval (`timeout`). When UnCondYield is configured to use this host yield function, it is called the FRIENDLY policy.

Figure 7.7 shows the performance impacts of the two UnCondYield implementations in the over-subscription case. Without gang-scheduling, FRIENDLY outperforms GREEDY, but the FRIENDLY policy makes the gang-scheduler[1] unstable. The performance of VM with the gang-scheduling runs slower even than the default (without gang-scheduling). Therefore, the right combination of GREEDY/FRIENDLY for the gang-scheduling needs to be examined. After searching through various combinations, one configuration was found. The configuration is to pick GREEDY unconditionally for the gang-scheduled VM

[1]At this moment, the gang-scheduler version is the initial one (Sgang).

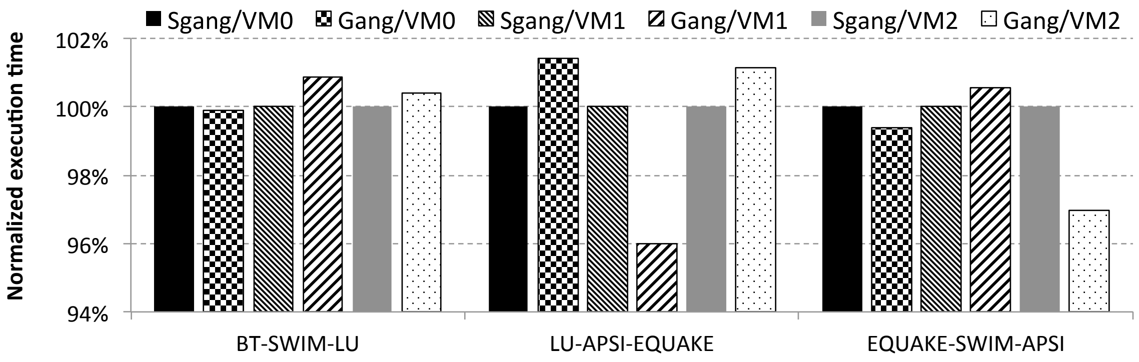


Figure 7.8: Two versions of gang-scheduling implementations are compared for performance. Sgang refers the version with waiting (the barrier) while Gang does not have waiting (no barrier). To get the results, each VM's execution time with gang-scheduling is first normalized over the VM's execution time without gang-scheduling (default). Then the average of the three normalized execution times is again normalized to the average of Sgang. For example, for Gang/VM0 of BT-SWIM-LU, the execution times for the three workloads are measured while gang-scheduling only the VM of BT, then they are normalized to the execution time without any gang-scheduling. The normalized execution time is averaged, and finally the average of Gang/VM0 is normalized to the average of Sang/VM0 to highlight the differences.

and selectively to choose GREEDY or FRIENDLY for the VMs without gang-scheduling. During the time slice when the gang-scheduled VM is run, the other VMs use GREEDY, and for other time slices, FRIENDLY is used by these VMs. Although the desired results are produced by this hybrid configuration, the underlying reason is unclear.

7.2.2 Strict Gang-scheduling vs Gang-scheduling Without Waiting

So far one iteration has been taken to address the UnCondYield function. The next two iterations were made to minimize side effects such as CPU fragmentation. We begin with the question of whether vcores wait for all arrive a barrier before the time slice begins. As described in Section 3.3.3, the initial version of the gang-scheduler (Sgang) keeps the gang-scheduled vcores waiting until all the vcores are ready to run. Given the initial version (Sgang), here is the pseudocode for the version without waiting. The difference from the initial version is that the followers in the leader-driven model execute as soon as the $SchedState$ is changed to the $GangRequested$ state. Previously, the $GangPending$ state made vcores wait until they all got to the barrier.

```
CondYield(...)
  ...
  switch (SchedState)
  case GangLeader:
    if VMID is the VM gang-schedule selected then
      if all vcores are in Running of LocalState then
        if time slice expires then
          call schedule()
          update SchedState of all the rest HW threads to be GangRequested
        end if
      else
        update SchedState of all the rest HW threads to be GangRequested
      end if
    else
      if time slice expires then
        call schedule()
      end if
    end if
  case GangRequested:
    if VMID is the VM gang-schedule selected then
```

```
        **if** time slice expires **then**
            set $SchedState$ to be $WaitForGang$
            set $LocalState$ to $Wait$
            call **schedule()**
        **end if**
    **else**
        call **schedule()**
    **end if**
**case** $WaitForGang$**:**
    **if** $VMID$ is the VM gang-schedule selected **then**
        call **schedule()**
    **else**
        **if** time slice expires **then**
            call **schedule()**
        **end if**
    **end if**
**end switch**
```

Figure 7.8 shows the comparison results over the three different cases. The difference in the average execution time is less than 5 %. Even though the execution time difference for each VM is not depicted, the difference is similar to the average execution time except for one case. In the LU-APSI-EQUAKE case, LU with strict gang-scheduling (Sgang) for APSI has about 10 % performance degradation compared to gang-scheduling for APSI (the version without waiting). Although waiting helps the gang-scheduled vcores be more synchronized in their time slices, performance degradation is also highly expected. Overall, given the mixture of benefits and costs, small differences, and occasionally better performance for gang-scheduling without waiting are observed. Therefore, I revised gang-

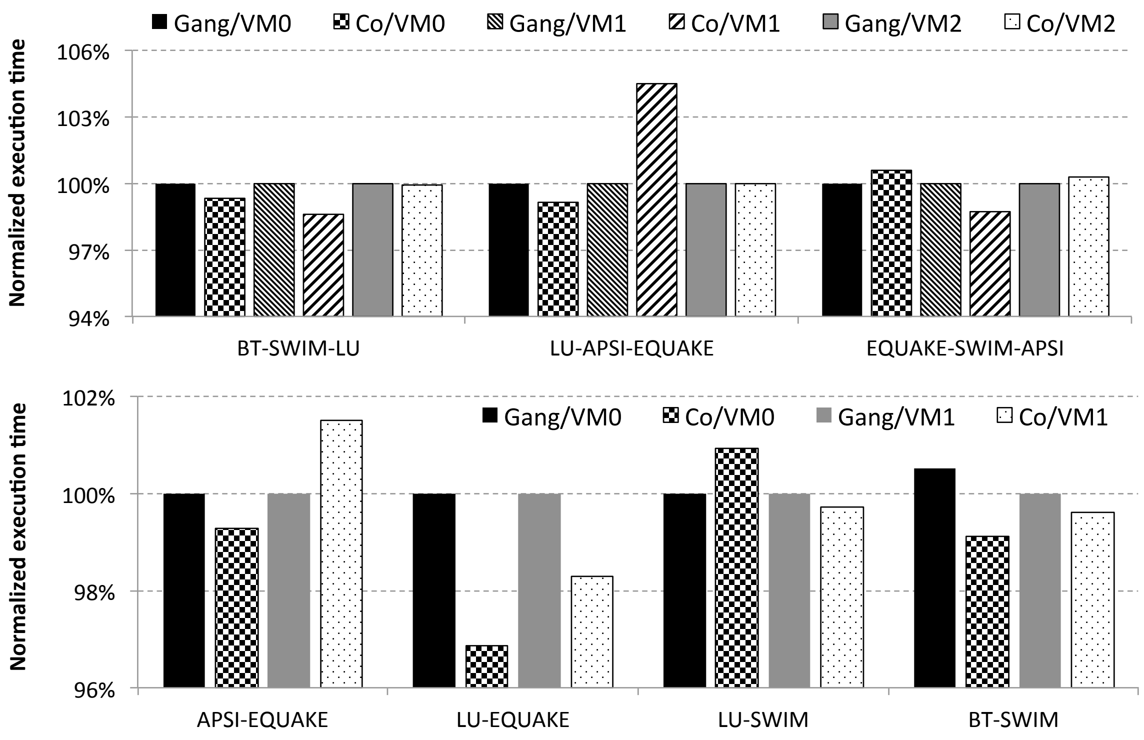


Figure 7.9: Comparison of performance between gang-scheduling (Gang) and NUMA-aware co-scheduling (Co). The way to produce the normalized execution time is same as Figure 7.8. Each case takes an average of normalized execution time for each workload over the default without any VMM-level scheduling, and then the average is normalized again to the average of Gang.

scheduling to avoid waiting (no barrier). revised without having the waiting time.

7.2.3 Gang-scheduling vs. Co-scheduling

The gang-scheduling model still has another hard restriction that defines a gang as all of the vcores of the VM, but combining all vcores across different NUMA domains is a different matter from combining vcores in one symmetric multiprocessor. The idea for co-scheduling is to split the gang based on the NUMA domain. Two points motivate us to do this: the co-scheduling benefit and cache effects. Along with reducing the waiting time for synchronizations in a workload, gang-scheduling forces all the vcores of the gang to execute, which can cause high priority task to be delayed. Co-scheduling addresses this heuristically by partitioning the gang into multiple groups. Given the NUMA architecture, vcores in each NUMA domain share at least the last-level cache. Scheduling the vcores of one VM at the same time reduces cache contention by disallowing vcores from other VMs to run.

The implementation of co-scheduling requires only grouping vcores for the target VM and allowing multiple leader vcores.

Figure 7.9 illustrates performance differences between the gang-scheduler (Gang, the one without waiting) and the NUMA-aware co-scheduler. The figure shows the normalized average execution time for the two and three VM cases of over-subscription. Overall, the results show no huge variations, except for the LU-APSI-EQUAKE case with APSI co-scheduled. LU has about a 10 % execution time increase with co-scheduling compared to the gang-scheduling, and the increase constitutes about a 5 % increase for the average execution time for the three VMs. However, the anticipated result comes from the case when LU is co-scheduled rather than the case where APSI is co-scheduled. The LU co-scheduled result on the same as the gang-scheduling result. Notice that our selected benchmarks are not I/O-intensive, and I/O-intensive benchmarks can have more side effects of gang-scheduling. The NUMA-aware co-scheduler is chosen to provide a more relaxed way of vcore-scheduling.

7.3 NAVAR Policy

The mechanism revision has resulted in the NUMA-ware co-scheduler. Along with that, the UnCondYield operation was accordingly adjusted to integrate with the vcore-scheduler.

LU-EQUAKE	Metric	Sampl1	Sample2	Sample3
Default	L1 miss rate	≈0%	≈0%	≈0%
	L2 miss rate	12%	12%	12%
	L3 miss rate	47%	47%	47%
	CPI	1.38	1.38	1.38
	CPM	643	629	628
LU is co-scheduled	L1 miss rate	≈0%	≈0%	≈0%
	L2 miss rate	13%	13%	13%
	L3 miss rate	48%	47%	47%
	CPI	1.41	1.41	1.41
	CPM	482	476	488

Table 7.1: Correlation check of various metrics for predicting the execution time. The co-scheduled LU has about a 15% reduction in the average of normalized execution times over the default that does not have any co-scheduling. The CPM has high correlation with the execution time. It is important to know that the cache miss rate for each cache level is calculated based on exclusive memory access count. For example, almost all memory accesses hit L1 cache, but missed accesses are only counted as L2 cache access for L2 miss rate: L2 miss rate = (L2 cache miss count) $\div$ (L2 cache access count).

Furthermore, memory mapping for the over-subscription case is set to the *interleaved* mapping for performance. Energy results will be shown in the next section. The policy needs to focus on the VM selection for co-scheduling. The brute-force approach looks feasible for this case because the overhead to switch between different co-scheduling configurations is quite small, and the mechanism does not suspend any vcore. If the brute-force approach is used, the accuracy of the system feedback value in predicting performance is very important.

7.3.1 Metric

Assuming that the brute-force approach is taken, the measured metric becomes crucial part to the correct selection. For this reason, metrics other than CPI are measured and compared with each other. Table 7.1 shows measured values of different metrics for two

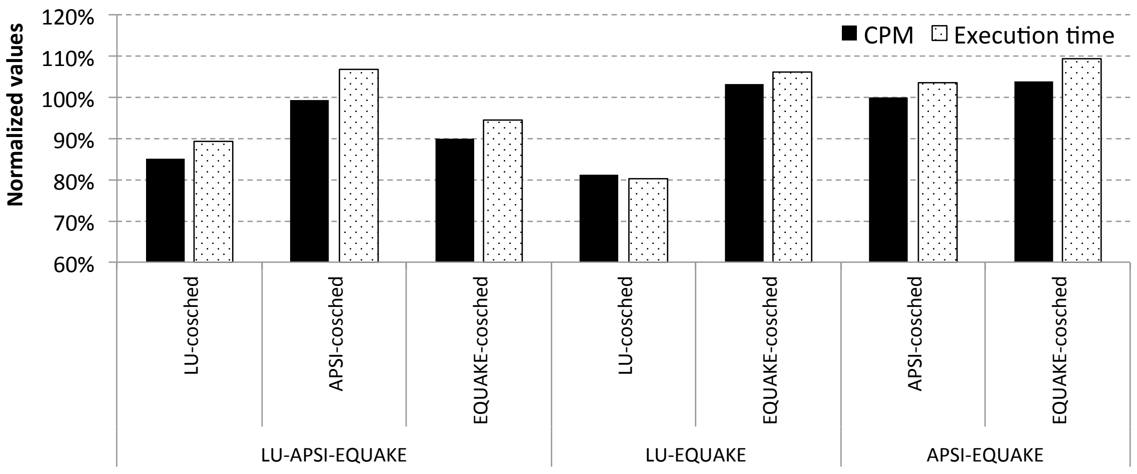


Figure 7.10: CPM has a correlation with the execution time. The execution time is the average of normalized execution time. Each VM's execution time is normalized to the default that does not have any co-scheduling.

different configurations. They are the default without any co-scheduling and the configuration where LU is co-scheduled. When LU is co-scheduled, execution time is reduced by 15% compared with the default case. The metric named CPM has a high correlation to this. The CPM metric is the spent clock cycles divided by the count of memory operations, while a memory operation count is defined as the number of load instructions from L2 cache to main memory[2]. The reason for counting from L2 cache access is because the number of L1 accesses is consistently too high. Filtering L1 access allows us to capture the scheduling effects. CPM is further compared with the execution time as shown in Figure 7.10.

[2]Counting memory access from L2 cache has more correlation than from L3 cache (LLC).

7.3.2 Algorithm

With the high correlated metric and very low reconfiguration time, the brute-force approach is taken for the NAVAR policy for co-scheduling. It also affects the overhead control and the configuration searching frequency. Now there are two differences in this case. One is the closed-loop structure itself, and the other is the style of overhead control. Previously, the control structure is either open-loop or semi-closed-loop. In loop-based policy, the frequency of configuration searching is accordingly adjusted to compensate for the missing feedback. Now the feedback value is available, and the reconfiguration time for the vcore-scheduling is small. As mentioned, the brute-force approach basically compares all possible configuration cases. One concern may arise about the overhead of searching and comparison. Even if reconfiguration time for different targets for co-scheduling is small, the overall time for searching and comparing increases linearly with the number of VMs. However, in practice, the increased number of VMs in an over-subscribed system increases the execution time as well. Thus the ratio of comparison time over the total execution time of VMs is maintained, as the number of VMs increases. Also, during the comparison, VMs are still executing rather than being entirely suspended. The following is the pseudocode of NAVAR for the over-subscription cases. It is also important to realize that the comparison interval and threshold values are not machine-specific. In principle, this policy is immediately applicable to different test machines.

```
$Prev\_Config \leftarrow Default\_Sched$
$interim\_scale \leftarrow 1$

while $1$ do
    if objective is power then
        set memory mapping for all VMs to consolidated in the same NUMA domain
```

```
else
    set memory mapping for all VMs to interleaved
end if

configure Default scheduling
sleep 10 secs
$Default\_CPM \leftarrow$ (measured-CPM)
for all VM with $VMID$ do
    exclusively set co-scheduling for one VM with ($VMID$)
    sleep 10 secs
    $(CPM, VMID) \leftarrow$ (measured-CPM, $VMID$)
end for
search minimum CPM pair: $(minCPM, VMID)$
if $minCPM < 0.95 \cdot Default\_CPM$ then
    do co-scheduling on $VMID$
    $Config \leftarrow VMID$
else
    configure Default scheduling
    $Config \leftarrow Default\_Sched$
end if
if $Config = Prev\_Config$ then
    $interim\_scale \leftarrow 2 \cdot interim\_scale$
    if $interim\_scale > 100$ then
        $interim\_scale \leftarrow 1$
    end if
else
```

```
        $interim\_scale \leftarrow 1$
    end if
    $Prev\_Config \leftarrow Config$
    sleep $interim\_scale \cdot$ 100 secs
end while
```

7.4 Experimental Results

The NAVAR policy is described as a brute-force approach with open-loop control. With small overhead for the system-reconfiguration time, this policy incorporates a simple control loop. This section presents experimental results with this policy. Because the policy does not include any offline-based modeling, test sets shown in the preliminary results are reused. At first, performance, power, and, more importantly, the energy results of the static memory mapping are discussed, and then a set of performance results for the dynamic adaptive co-scheduling, along with *interleaved* memory mappings follow.

7.4.1 *Interleaved* Memory Mapping for Performance and Energy

Figure 7.11 compares performance, power, and energy across different memory mappings along with the co-scheduling configurations. The *interleaved* memory mapping for all VMs is expected to boost performance over the *consolidated* memory mapping to one selected NUMA domain for all VMs. However, we need to check whether the *interleaved* memory mapping for all VMs has also the best energy efficiency. Based on the test machine measurements, the estimated power saving with the proposed memory power-off mechanism is about 5 %, as shown for normalized power between MC^2 and MIMI shown in Figure 7.11. The LU-EQUAKE case has the smallest performance difference between

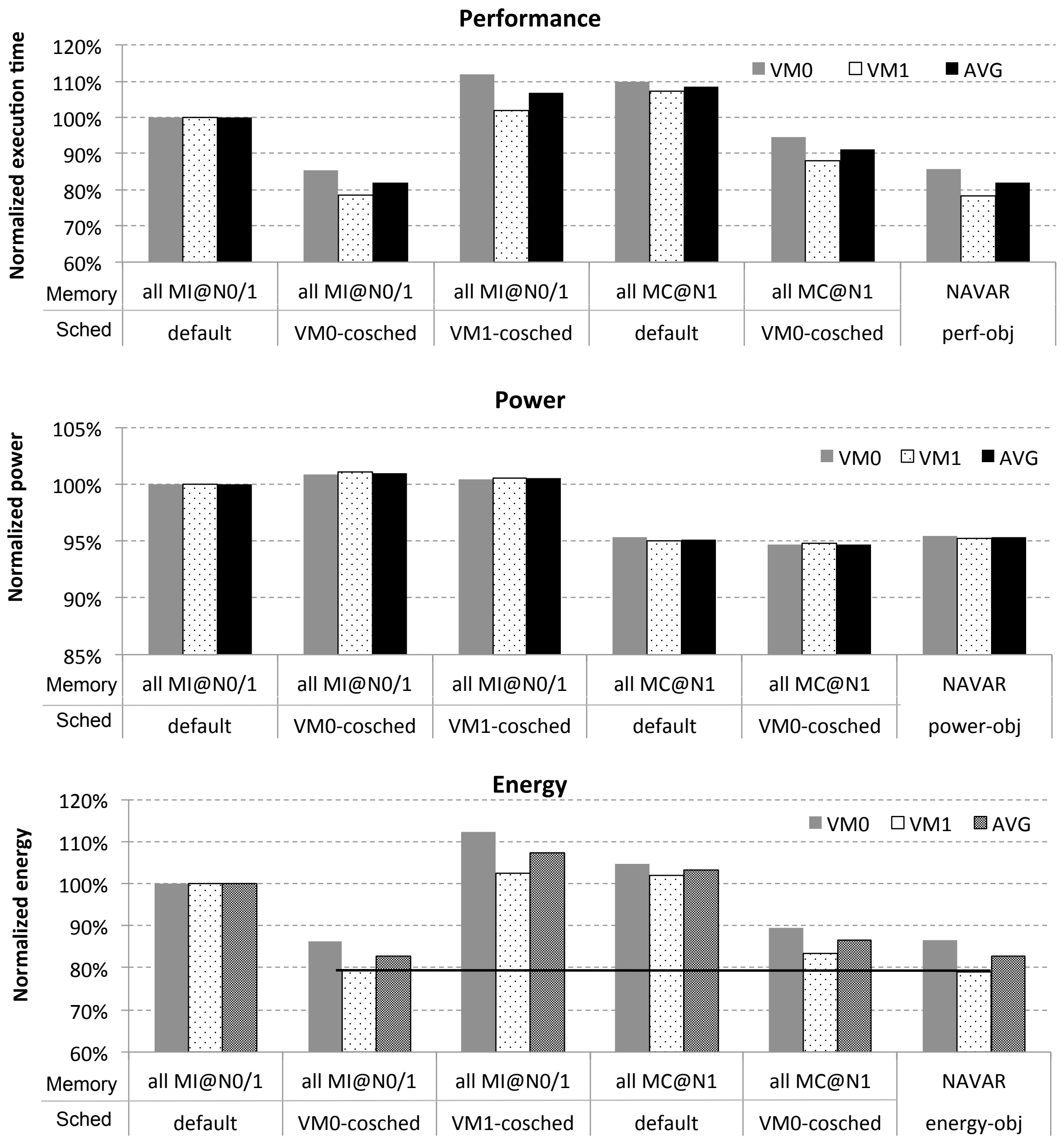


Figure 7.11: Comparison of performance, power, and energy for LU and EQUAKE benchmarks for the MIMI and MC^2 mappings and the default and co-scheduling cases. Energy consumption, between the MIMI and MC^2 mapping for both the default and the the co-scheduled LU, demonstrate the efficiency of the MIMI mapping.

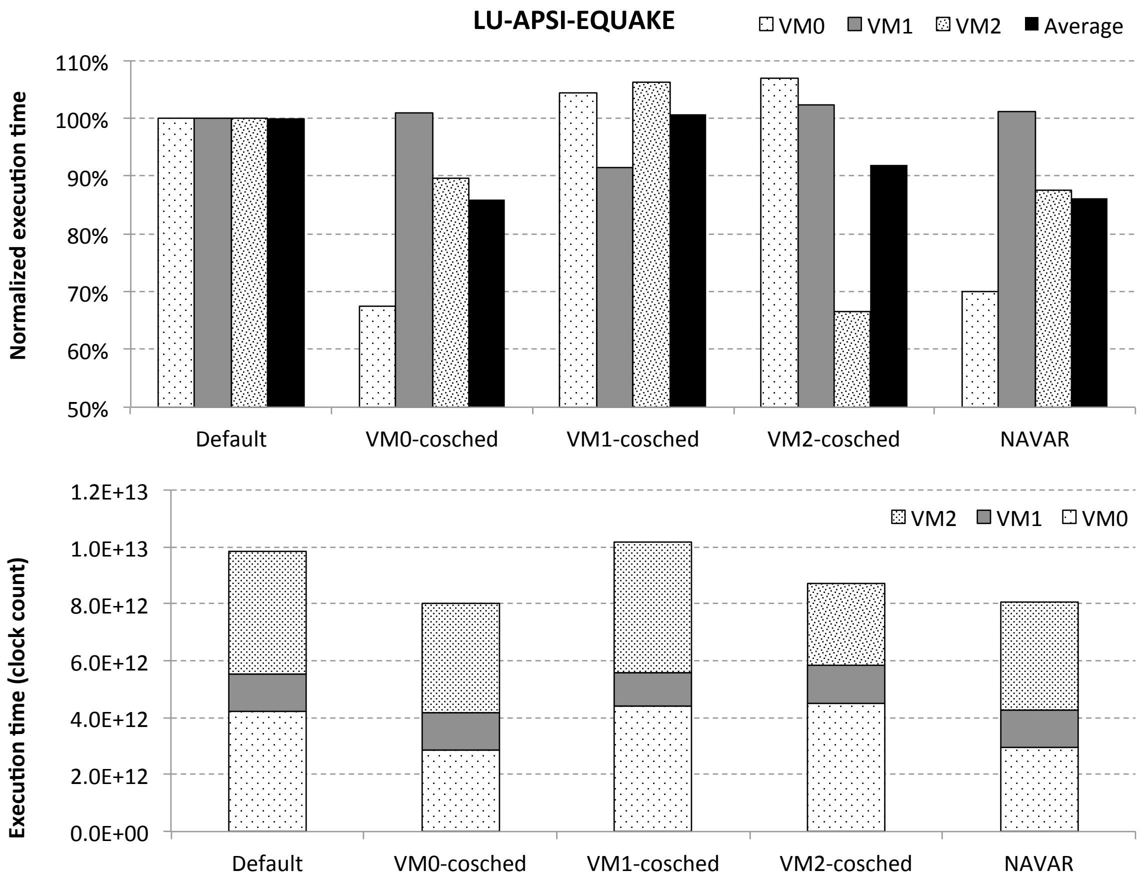


Figure 7.12: The co-scheduling results for the LU, APSI and EQUAKE benchmarks. The top figure shows the normalized execution time over the Default case for each benchmark. The Default case does not have any co-scheduling. The average is calculated over the normalized execution time of the benchmarks. The bottom figure illustrates the raw execution time (clock cycles) in a stacked way. Figures 7.13, 7.14, 7.15, 7.16, 7.17 and 7.18 also these two types of figures.

MIMI and MC^2 in the preliminary results shown in Figure 7.2. Nonetheless, the MIMI configuration has a performance benefit over the MC^2 mapping to overcome the increased power, which is shown in the energy comparison in Figure 7.11. Particularly, the MC^2 mapping and the MIMI mapping with LU co-scheduled, and the NAVAR result demon-

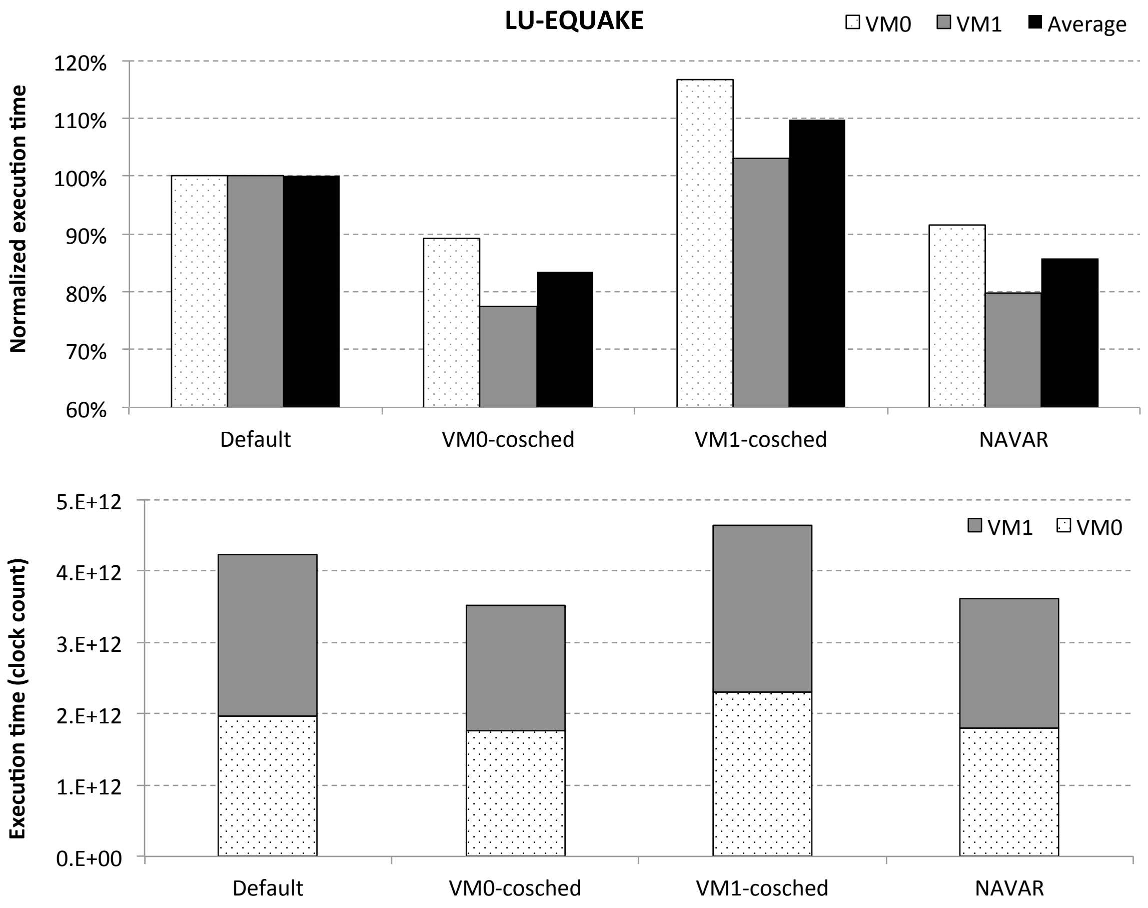
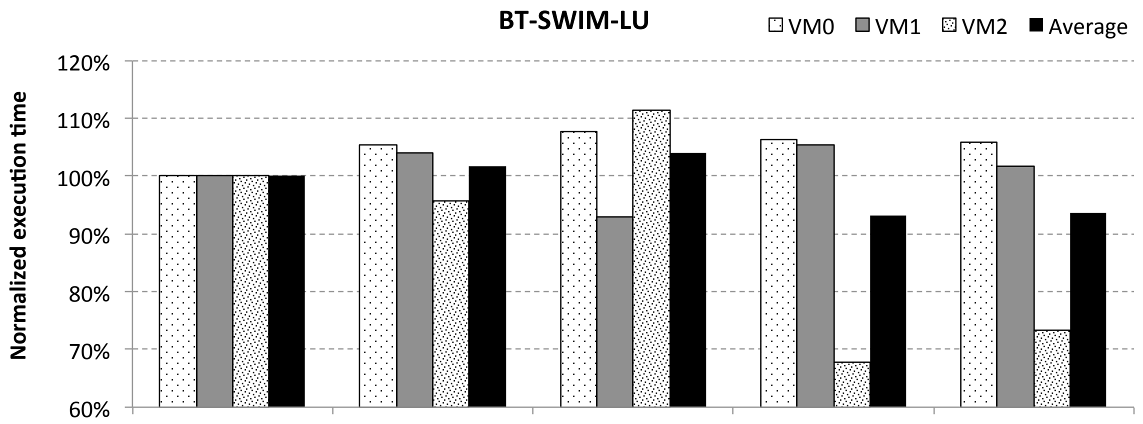


Figure 7.13: The co-scheduling results for LU and EQUAKE benchmarks.

strate that the MIMI mapping should be selected for the energy objective as well. Increasing the number of VM to more than two VMs is also expected to increase the performance of the MIMI mapping over the MC^2 mapping, as memory bandwidth demands increase.

7.4.2 Co-scheduling Results

Energy efficiency is now demonstrated for the MIMI mapping as well. Once MIMI is used for both energy and performance objectives, the ability of NAVAR to pick the correct



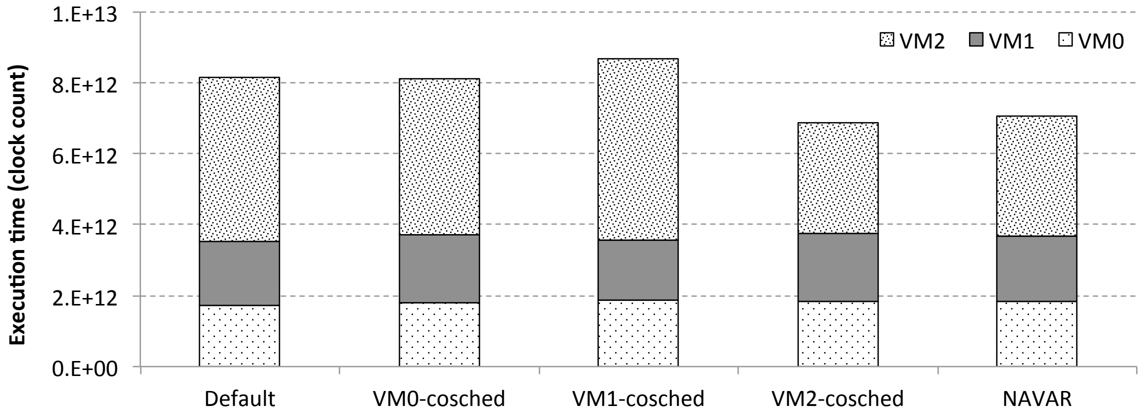


Figure 7.14: The co-scheduling results for BT, SWIM and LU benchmarks.

VM for co-scheduling needs to be checked. Since MIMI is to be selected for the energy objective, energy results are expected to be aligned with performance results. The results for the power objective are only measured for the LU and EQUAKE case because it unconditionally configures memory mapping to the *consolidated* for all guest memory to one NUMA domain. Therefore, the results of the seven cases are for the performance objective only. The execution time for each workload is measured while all benchmarks are running. Therefore all benchmarks are run multiple times. Figures 7.12, 7.13, 7.14, 7.15, 7.16, 7.17,

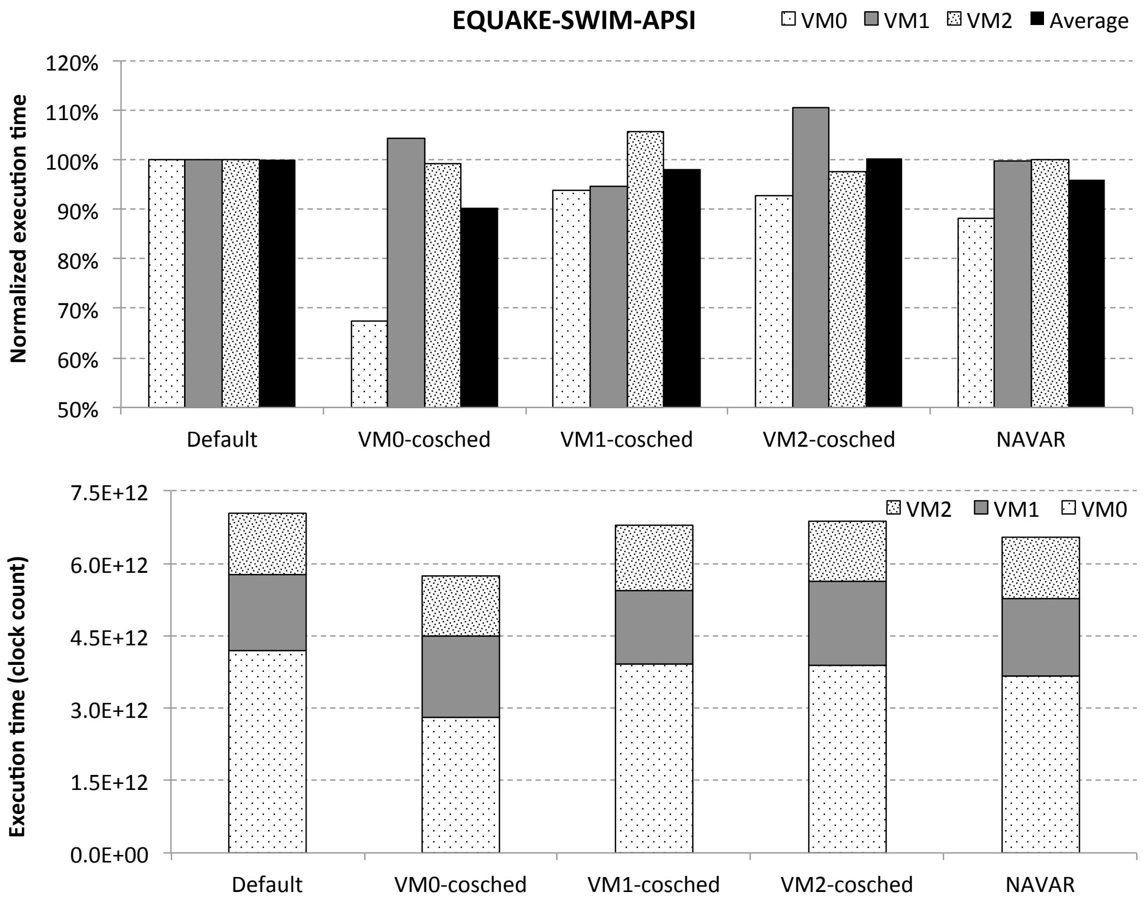


Figure 7.15: The co-scheduling results for the EQUAKE, SWIM and APSI benchmarks

and 7.18 show results for the seven cases. There are two types of graphs shown in each figure. The first type (the top graph) for each figure shows the performance with the same weight over the VMs. The presented average is calculated by the sum of the two or three VMs' "normalized" execution time over the default' sum of normalized execution time. Instead of the normalized execution time, the raw execution time is presented along with the sum of the two or three raw execution times, as depicted in the second type (the bottom graph of each figure). The top graph shows the performance without considering task size

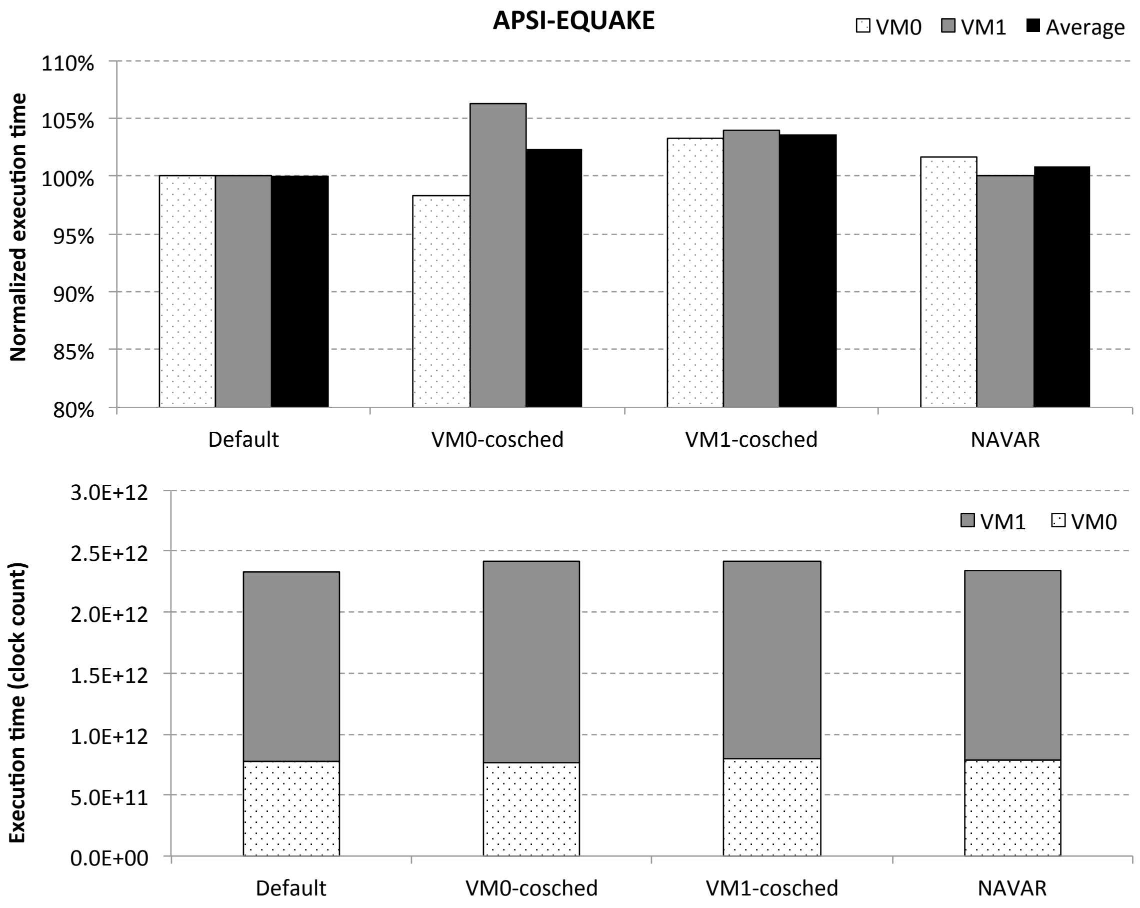


Figure 7.16: The co-scheduling results for APSI and EQUAKE benchmarks.

while the bottom graph highlights the performance individually. These two styles constitute the two possible performance objectives whether to maximize speedup (top graph) or to minimize makespan (bottom graph).

The last three figures, in fact, do not show the co-scheduling performance benefits, but rather it demonstrate that the NAVAR system does not introduce unnecessary overheads. Now the focus needs to be on the first four figures to check whether the NAVAR system selects the right VM, when clear benefits are available. As described, NAVAR

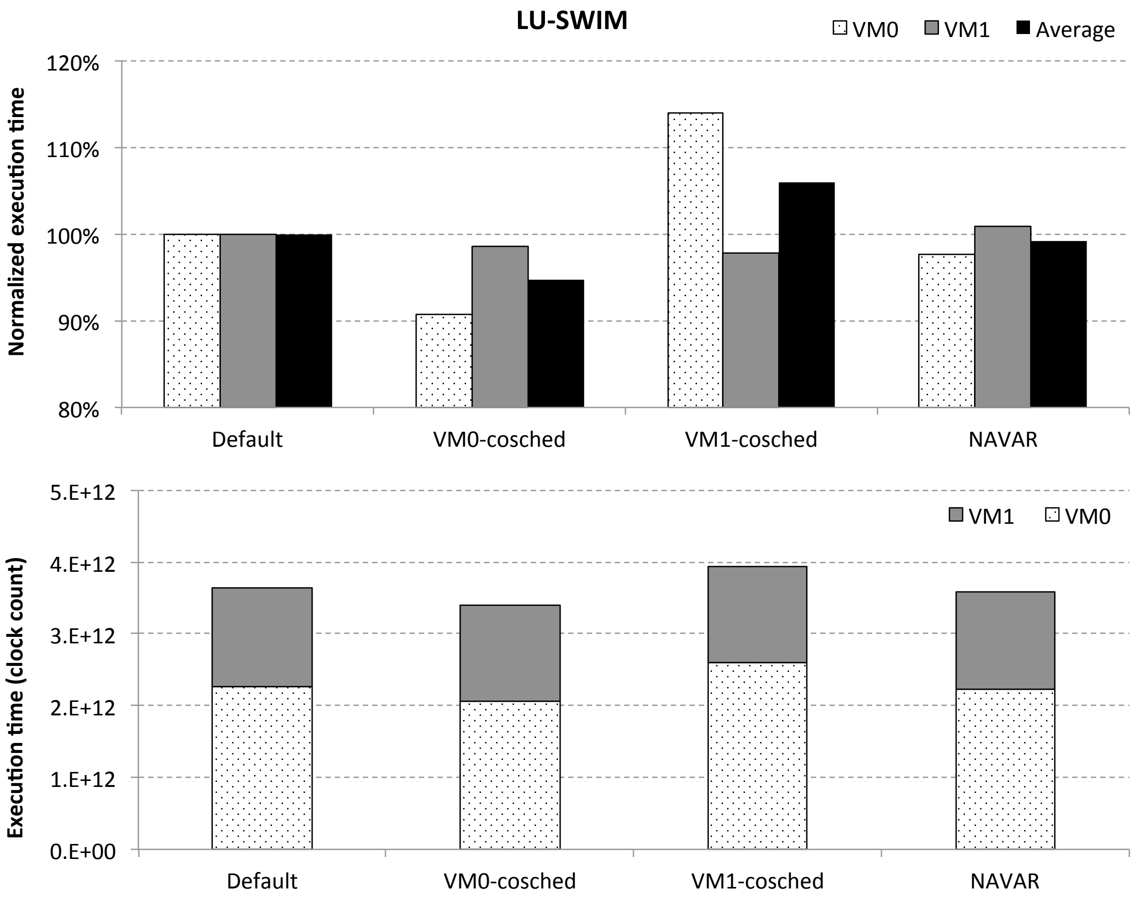


Figure 7.17: The co-scheduling results for LU and SWIM benchmarks.

selects one configuration based on the immediate feedback value observed. Figure 7.12 shows one of best results, for LU, in which NAVAR almost completely replicates the best static result. This result is equally shown in the two types of figures. It is also true for the next two cases, the LU-EQUAKE and BT-SWIM-LU cases, where NAVAR replicates the best performance results of the static configurations. For the EQUAKE-SWIM-APSI case, NAVAR produces a different result than the static configuration results. As shown in Figure 7.15, the NAVAR result is similar to the combination of the two static results, the

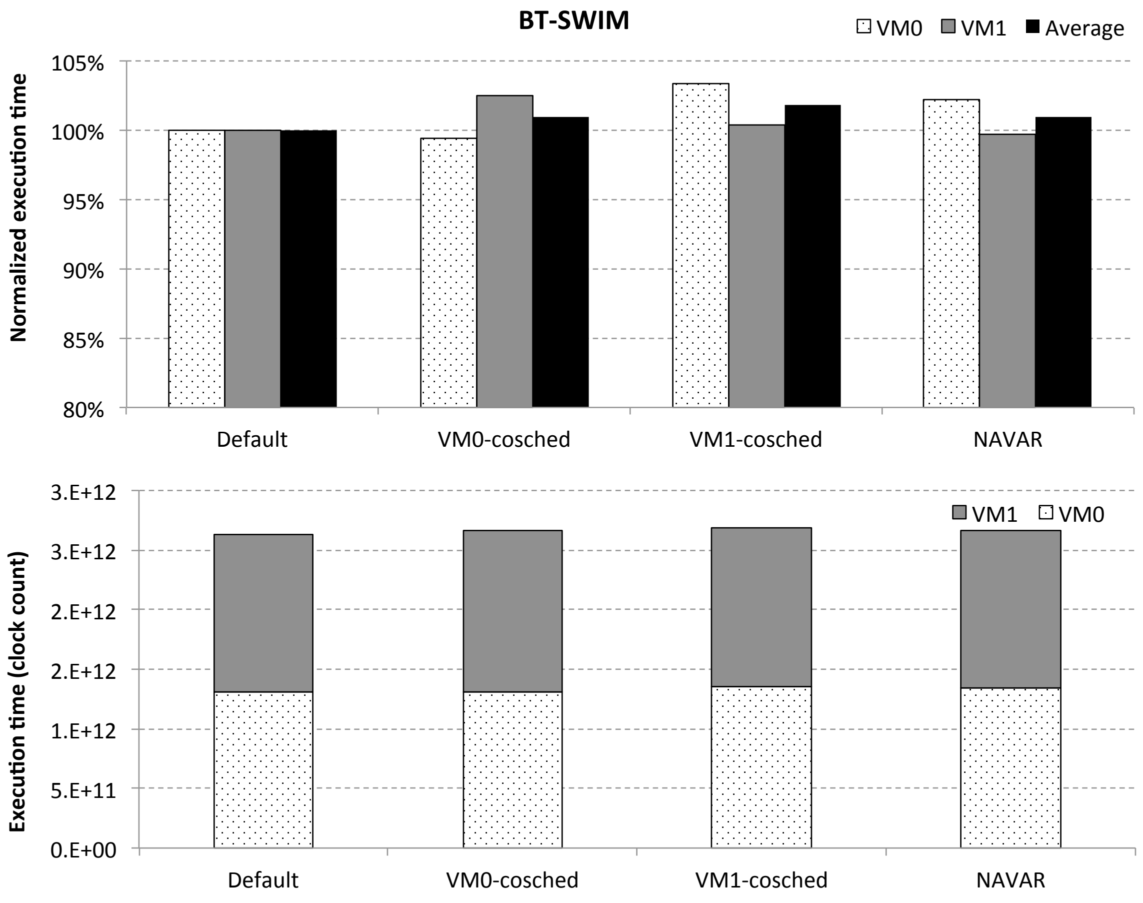


Figure 7.18: The co-scheduling results for BT and SWIM benchmarks.

VM0-cosched and VM2-cosched results. This can be interpreted as follow: in the middle of execution, NAVAR switches the co-scheduling target between the two VMs, VM0 and VM2, as the execution phase changes. As shown in the two types of graph, in this case, NAVAR has slightly increased performance compared to the best static result.

Further discussion In contrast to the approaches attempted for the under-subscription and over-subscription cases, the NAVAR policy here is simple and incorporates no model. One upside for this policy design is its universal applicability, but it is not yet explicable why we have such performance benefits by the co-scheduling. Two possible factors are considered. Decreased waiting time for synchronizations in VMs, and reduced cache contention are expected from having co-scheduling. The synchronization factor looks to have more of an impact than the cache locality factor. As shown in Table 7.1, the cache miss rate is unchanged between default scheduling and the co-scheduled cases for LU. The cache miss rate changes are still small for other benchmark as well. Nonetheless, this does not mean that the cache miss rates are not changed at all. Rather, even if some changes are observed, they are not enough to make the cache locality factor the main reason for the performance benefit of the co-scheduling. Consider a situation in which some vcores wait for a lock-holder vcore. Instructions are still retired, in fact, rapidly. The CPI metric includes all instructions, even if they are in a wait loop. In contrast, the CPM metric can consider only the memory instructions that reache to lower caches. As we showed, CPM has more correlation than CPI to the execution time results. This result also supports the synchronization factor as being dominant. I believe the reductions lock waiting are the major factor in the performance benefit of co-scheduling.

7.5 Conclusion

This chapter has described the system design for the over-subscription case. In the over-subscription case, both computational and memory resources are heavily utilized. Therefore the *interleaved* memory mapping is consistently used to provide sufficient memory-access bandwidth, and the vcore-scheduling configuration is used to dynamically optimize system performance and energy consumption. With a cheap reconfiguration cost and a highly correlated metric, CPM, the brute-force approach is used by the NAVAR policy and we can avoid an elaborate modeling process.

Chapter 8

Generalization

The last four chapters discussed three cases of the overall resource mapping problem in a virtualized NUMA machine, considering cases of vcore mapping, memory mapping, and vcore-scheduling. Each case is defined based on the total number of vcores of the active VMs compared to the available number of hardware threads, and NUMA domains in the machine. The under-subscription case is when the number of vcores are less than the total number of hardware threads divided by the number of NUMA domains, and in this case, the *consolidated* vcore mapping is possible. The full-subscription case is when the number of vcores is above the threshold for the under-subscription case but less than the total number of available hardware threads. Cases other than these types are classified as the over-subscription case. Given this classification, we ran experiments on representative test using a NUMA domain test machine (the R410). For the under-subscription case, a VM ran with eight vcores in the test machine with total 16 available hardware threads. For the full-subscription case, two VMs were set up with six vcores each mapped to each NUMA domain for each VM, and two workloads ran separately on each VM. The test cases chosen for over-subscription case had two or thee VMs run with 14 vcores each, which were evenly distributed over each NUMA domain with identical vcore mappings to hardware threads. Here, depending on the total number of VMs, two or three vcores shared the same hardware thread.

This chapter begins by discussing some missing cases in the perspective of generalizing these results. Nest, we discuss variations and experiment results in applying the methodology to a different test machines (the Dell R415). In the last section, potential constraints and feasibility of applying NAVAR to other machines with more than two NUMA domains are discussed.

8.1 Combined Policy

Given the methodology and policies, given throughout the previous chapters the three different case, this section discusses in combination, considering an increasing number of of vcores. Here we assume a two NUMA domain machine. If there are significant tradeoffs in opportunity, the NAVAR policy or protected policy design is discussed. Otherwise, a static configuration is sufficient.

8.1.1 Under-subscription

The under-subscription case is determined by the number of active vcores and whether they are mappable to one NUMA domain in the balanced mapping; in other words, whether the *consolidated* vcore mapping is an available option or not. The cases shown in Chapter 4 and 5 have one VM with eight vcores. Although the specific case has been considered so far as under-subscription, the NAVAR policy is also applicable for a different number of VMs as long as the same or similar kinds of tradeoffs exist. For example, when two VMs run, and each VM has four vcores, there is also a choice between the *consolidated*

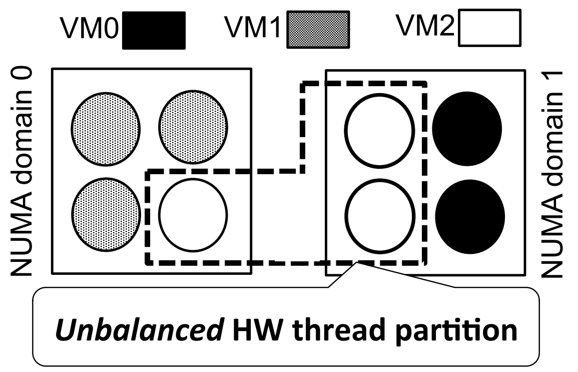


Figure 8.1: Illustration of the unbalanced hardware thread partition. When three VMs run on a two NUMA domain machine, vcores of one VM has to be mapped to the hardware threads in the unbalanced hardware thread partition. This vcore mapping is different from the *interleaved* vcore mapping.

and *interleaved* vcore mapping, and also for the memory mapping. The two VMs can be treated together reducing to the situation with one VM. On the other hand, when the total number of vcores is small, two or three vcores, for example, tradeoffs then are barely expected.

8.1.2 Full-subscription

For the full-subscription case, the specific example considered so far has two VMs where each has almost the same number of vcores, which is as half the number of all available hardware threads. We need to discuss situations with one VM or three, or more VMs.

If one VM is running with as many vcores as the number of hardware threads, both the *interleaved* and *consolidated* memory mappings are the options, and a decision can be made similarly to the way we determine the memory mapping for the under-subscription.

When three or more VMs are running, it becomes a bit different, because it depends on the vcore partitioning. A static vcore mapping may be sufficient, but dynamic adaptive vcore mapping could be necessary. Previously, when two VMs run, it is apparent that we can partition the vcores into two partitions, one for each NUMA domain. Now one and more VM and its vcores are added. First, all vcores should be partitioned into a number of groups that is same as the number of NUMA domains, and then the vcore mapping can be fixed. If vcores from one VM are located in a group of hardware threads (HW threads) across different NUMA domain, the *unbalanced* HW thread partition shown in Figure 8.1, the VM needs to be carefully chosen. Note that the vcore mapping to the hardware threads in the *interleaved partition* is different from the *interleaved* vcore mapping, because vcores in the *interleaved partition* are not guaranteed to be evenly split across NUMA domains. One principle should be applied when selecting a VM: VM in the CS class should not have vcores put in the *unbalanced* HW thread partition. For memory mapping, similar to the approach made for NAVAR, an estimate of the memory-access bandwidth demands determines whether to choose the *interleaved* memory mapping or not.

8.1.3 Over-subscription

For over-subscription, there are some corner cases as well. For example, when the number of vcores across VMs is just above the number of hardware threads, it is still close to the full-subscription situation. In this case, the vcore mapping is not the balanced mapping[1], which violates the **`vcore-scheduling`** problem prerequisite, as defined in Chapter 2. With co-scheduling here, any further improvement is hardly expected.

The other corner case is, for instance, when two VMs run and only some of each VM's vcores share one hardware thread. Co-scheduling may be only available on the hardware threads shared by multiple vcores. According to the NAVAR policy, the measured performance will indicate the performance impact of co-scheduling. Therefore, regardless of the

[1]The balanced mapping does not allow multiple vcores from the same VM to be mapped to the same hardware thread.

amount of hardware thread sharing, the NAVAR policy is expected to configure the vcore-scheduling appropriately. Virtual cores that are alone on their HW threads are always ready to run. For vcore mapping, the number of vcores of one VM may be small enough to allow the *consolidated* vcore mapping, but the *interleaved* vcore mapping is still preferred. The benefit of the *consolidated* vcore mapping is for power, which also leads the energy efficiency in the under-subscription case. In the over-subscription cases, however, vcores of the other VMs are mapped to all the NUMA domains, which does not allow us to turn off power to them. With the *interleaved* vcore mappings, the *interleaved* memory mapping also is desired. In practice, furthermore, it is rare that all the VMs' memory demands are very small enough to allow us to consolidate all guest memory into one NUMA domain. When a workload executes threads that fully utilize almost all of the hardware threads in the machine, the workload's memory bandwidth demand is expected to be large enough to need more than one memory channel.

8.1.4 Combined Policy

The following pseudocode describes the combined policy, which incorporates the discussions so far. Some situations need adjustments to the NAVAR policy. In particular, when the vcore count quite small in the under-subscription case, or when the vcore partitions are not aligned with NUMA domains in the full-subscription case, the adjusted policy is described. The combined policy is mostly based on the NAVAR policies that were evaluated through experimental test cases, which we refer to as NAVAR(US), NAVAR(FS) and NAVAR(OS) for the under-subscription, the full-subscription, and the over-subscription respectively. The pseudocode is:

```
    **if** $number\_of\_overall\_vcores \leq threshold_{sub\_under} \cdot \frac{number\_of\_hardware\_threads}{number\_of\_NUMA\_domain}$ **then**
    use *interleaved* vcore and memory mapping for performance, and *consolidated* vcore
    and memory mapping for energy and power. No scheduling is needed.
else if $number\_of\_overall\_vcores \leq \frac{number\_of\_hardware\_threads}{number\_of\_NUMA\_domains}$ then
    apply NAVAR(US)
else if $number\_of\_overall\_vcores \leq number\_of\_hardware\_threads$ then
    if the partitions of vcores by VMs are not aligned with the HW thread partitions by
    NUMA domains then
        locating a VM in the none-CS class to the *unbalanced* HW thread partition[2].
    end if
    apply NAVAR(FS) to the misaligned VMs
else
    if $number\_of\_VMs > 1$ then
        apply NAVAR(OS)
    end if
end if
```

8.2 Different Machine

All test results so far come from one test machine (the R410). Questions can be raised, such as whether the tradeoffs we have identified are machine-specific, how much NAVAR needs to be modified for a different machine, and whether NAVAR works properly on a different machine. These motivated me deploy the NAVAR on a different test machine (the R415). We begin with a discussion for the difference between the two test machines, and then changes to the adaptive system design, particularly for modeling, are discussed. The section ends with the experimental results.

[2]This VM's vcores are placed in the *unbalanced* HW thread parition, so the vcore mapping is fixed.

	R415	R410
Number of NUMA nodes	2	2
Number of cores per NUMA node	4 cores	4 cores (w/ 2 way SMT per each)
CPU / Memory frequency	2.2GHz/1333MHz	2.4GHz/1066MHz
Last level cache size	6MB	12MB
Performance counter	no mem-inst counting	supports mem-inst counting

Table 8.1: Differences between the two test machines with different CPU architectures. "mem-inst" refers to memory instruction.

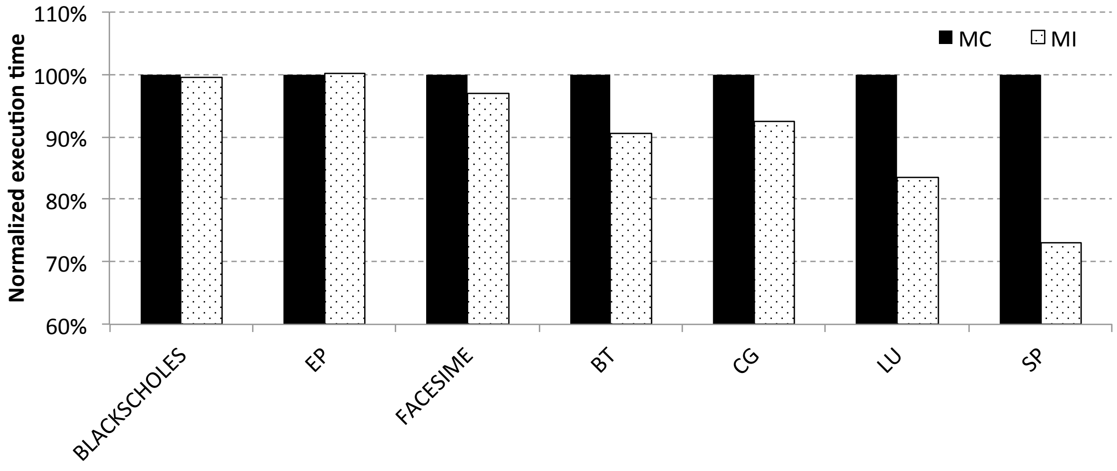


Figure 8.2: Tradeoffs of memory mapping in the under-subscription. Depending on workloads, the performance benefit of *interleaved* memory mapping has great variances.

8.2.1 Tradeoffs and Opportunity

Table 8.1 illustrates the differences and similarities between the two test machines. Both have the same number of sockets. The R415 test machine has a smaller cache size than the R410 machine, and the R415 machine does not incorporate the SMT feature. The two test machines' CPUs have a different clock frequency, as do the memory subsystems. As each NUMA domain of the R415 test machine provides four hardware threads (four physical cores), $\leq$4 vcores is the threshold for the *consolidated* vcore mapping with the balanced mapping. This scale of parallelism is smaller than on the earlier experiments. Thus, the

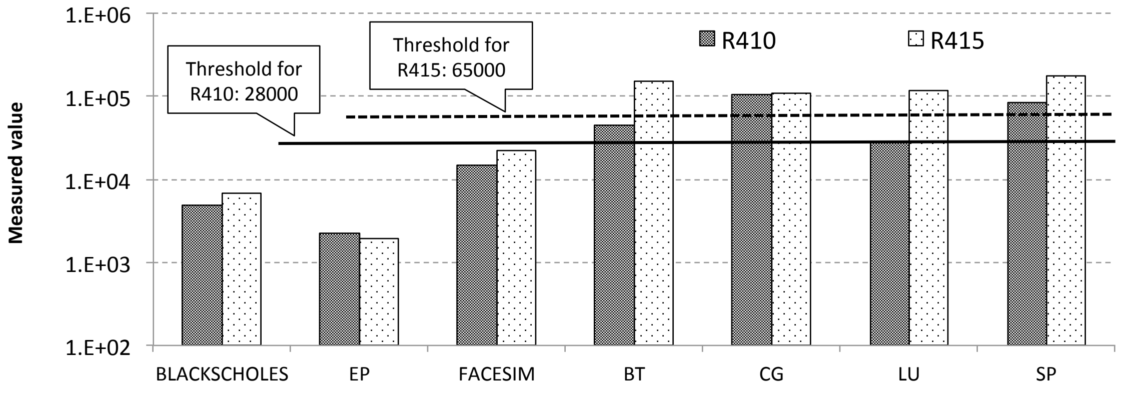


Figure 8.3: Comparison of the thresholds for the LWSS classification in the two different machines. With all differences, workloads are classifiable and even the threshold value of R410 is possibly reusable for R415; however, the threshold for R415 is conservatively set a little higher than that.

number of vcores is set to six, and the vcore mapping is fixed to the *interleaved* mapping. Memory mapping is dynamically adjustable in the coarse-grained level during runtime, according to the preliminary results depicted in Figure 8.2. Most workloads used here come from the NAS benchmark suite[3]. Given that, the tradeoff opportunities for the R415 test machine is in the choice between the *interleaved* and *consolidated* memory mappings in the under-subscription case.

8.2.2 NAVAR Migration to New Machine

The adaptive system is composed of three major components: (1) detection framework, (2) the policy, and the mechanism components. The mechanism component has three kinds of adaptation mechanisms, all of which are machine independent. No modification is needed to deploy them. The other two components, however, need some adjustment.

[3]At this moment, the selected benchmarks are the most stable on the R415 test machine.

Detection framework migration For the detection framework, the scanning interval has to be changed due to incompatibility of the hardware monitor (performance monitor), which defines the scanning interval. The scanning interval was previously determined based on the statistical correlation with the performance results. Accordingly, the interval can be redefined here with one of the supporting hardware events. Nonetheless, the detection framework is immediately available for the scope of the given resource mapping problem, which addresses only the memory mapping concerns. Previously, for the under-subscription case, the resource mapping was established in Table 5.6. Here, with the fixed *interleaved* vcore mapping, the memory mapping is determined by the LWSS classification result. The classifier for the LWSS classification is the $r_{\cup as}$ metric, which is less sensitive to the scanning interval as shown in Figure 5.11. Therefore, to workaround the incompatibility, the scanning interval is set to be the same as the probe duration. Since the probe duration is defined in wall clock time, the metric that is actually used can be expressed in $r_{\cup ac}$ (the average access page ratio per clock counts). The two metrics are approximately equal: $r_{\cup ac} \approx r_{\cup as}$.

Policy migration The NAVAR policy has three different versions depending on the resource demand situation. Particularly, for under-subscription and over-subscription, the policy component embeds the decision tree, the offline-based model. The thresholds, need to be adjusted for the new machine. However, translating a threshold value defined over $r_{\cup as}$ to a value over $r_{\cup ac}$ also needs to be done. Figure 8.3 depicts the similarities and differences of the measured values for the two metrics in the two different machines. The setups of the two machines are different. The differences lie in the architectural attributes (cache size, CPU architecture, CPU and memory frequency), the resource mappings (the CCMC mapping for R410 and the CIMC mapping for R415), and the metrics ($r_{\cup as}$ for R410 and $r_{\cup ac}$ for R415) themselves. With all the differences, the workloads are classifiable to the

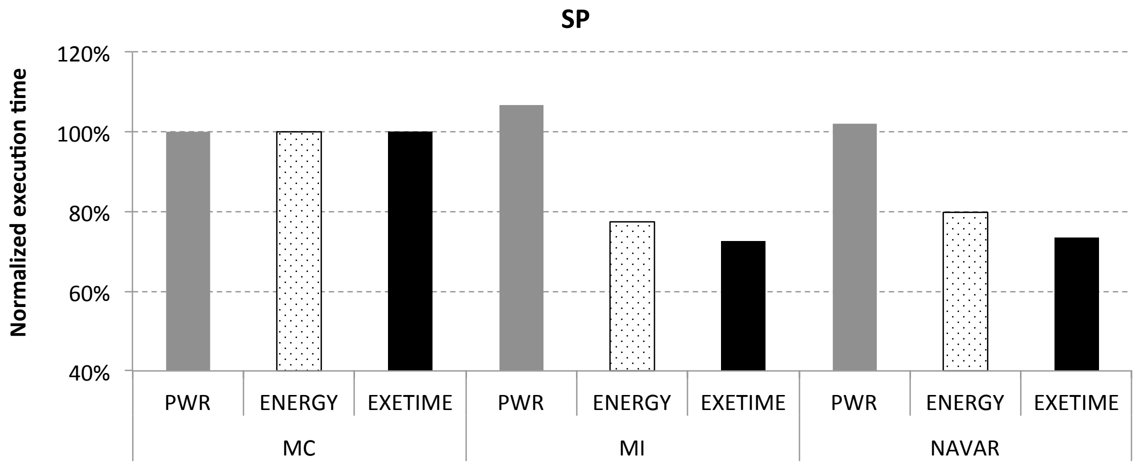


Figure 8.4: Results of the SP benchmark in the R415 test machine. The NAVAR results are compared with the two static configuration results. The vcore mapping is the *interleaved* mapping for 6 vcores with the under-subscription situation. The setup for following three figures, Figure 8.5, 8.6 and 8.7, are the same.

LWSS and !LWSS classes on the R415 machine as well, and also these classification results have high correlation with the preliminary results shown in Figure 8.2. Although the threshold of the R415 is conservatively set a bit higher than that of the R410, the threshold value of the R410 is probably usable for the R15 machine as well. The software monitor based detection scheme incorporates hardware assistance to define an interval for page scanning for each vcore. With that, the part of the decision tree is migrated to the R415 test machine, and similarly the entire decision tree can be recreated for the R415.

8.2.3 Evaluation

So far, the two points, the tradeoffs and opportunities on the different test machine and the policy migration, have been discussed. For the experimental evaluation, four workloads are selected to check the NAVAR performance. The test results are shown in Fig-

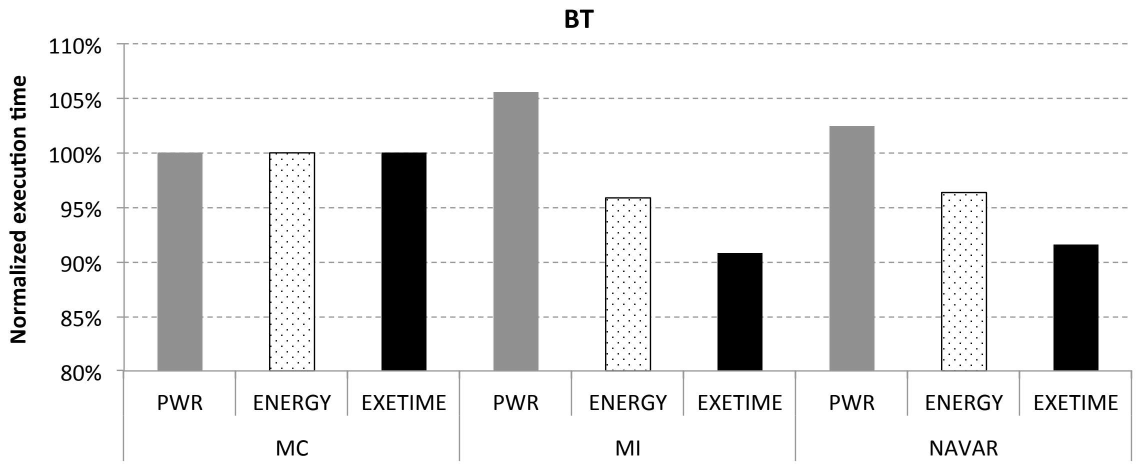


Figure 8.5: Results of BT benchmark. NAVAR selects a correct memory mapping configuration for the three objectives individually.

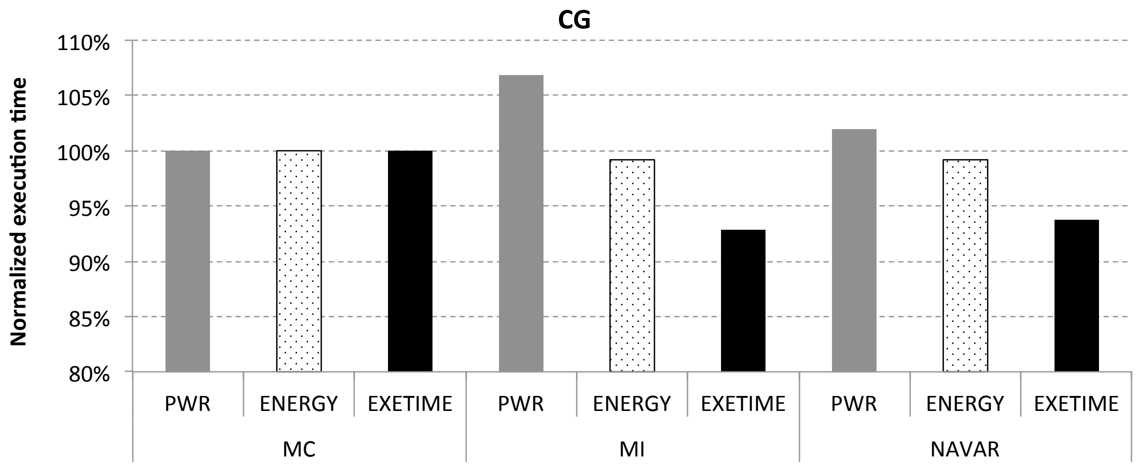


Figure 8.6: Results of CG benchmark. NAVAR selects a correct memory mapping configuration for the three objectives individually.

ures 8.4, 8.5, 8.6 and 8.7. The first three cases show benefit using the *interleaved* memory mapping, whereas the last case, EP, is better using the *consolidated* memory mapping for the energy objective. For the *interleaved* memory mappings, the performance and energy

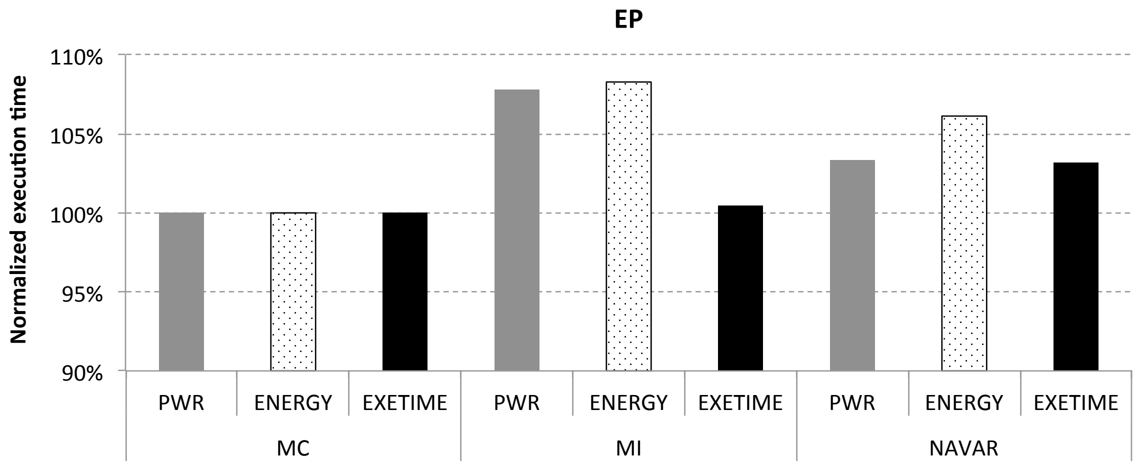


Figure 8.7: Results of EP benchmark. Due to short execution time of EP, the memory migration overhead results in increased power and execution time, which also increases energy.

Mapping	Relevant Factor	2 Nodes Status	>2 Nodes Expectation
Interleaved vcore mapping	Cache contention	UP	Less UP
Interleaved memory mapping	Memory bandwidth	OP	More OP

Table 8.2: Summary of analysis of scaling to more NUMA domains. UP refers to under-provisioning, so does OP to over-provisioning.

results are almost same as the best static configuration's results. Regard the EP benchmark, the NAVAR results perform worse than the best static configuration (MC). With a relatively short execution time, the memory migration cost has more impact on the execution time, power, and energy. Moreover, NAVAR has more cost for the energy consumption objective than the other two objectives, because energy is the combined metric of power and execution time, and both of them are increased separately.

8.3 Scalability of NUMA Domains

Up to now only machine with the two NUMA domains have been considered. We now consider leárger number of NUMA domains. The discussion concerns mostly vcore and memory mapping-related factors rather than vcore-scheduling, because the co-scheduling policy does not incorporate any model. In terms of the relationship between resource provisioning and demands, the vcore and memory mapping have an important difference. The vcore mapping is related to the cache contention factor, and cache space is mostly under-provisioned (UP). Unless the WSS fits in the cache, more cache space is demanded. The memory mapping controls memory bandwidth, which is occasionally over-provisioned (OP). The *interleaved* vcore mapping is expected to decrease cache contention. However, as observed on the two NUMA domain machines, cache contention does not seem to change for some workloads because of their memory-access patterns, those with the large number of hot pages (UP). With the scaled system, this type of workload may have reduced cache contention with the *interleaved* vcore mapping (Less UP). For memory mapping, according to the observation of the two NUMA domain machines, some workloads are sufficiently provisioned with the one memory channel. In this case, the *interleaved* memory mapping does not produce any difference due to the application's small bandwidth demand (OP). When more memory channels are added, the situation results in more over-provisioning (More OP). These two factors and their expected impact on the scaled NUMA domains are summarized in Table 8.2. With less under-provisioning (Less UP), vcores tend to be still interleaved, while memory mapping wants to be more consolidated for energy efficiency, with more over-provisioning (More OP). Overall, the *consolidated* memory mapping along with the *interleaved* vcore mapping increases memory-access distance. Therefore, to compensate, memory mapping conversely needs to be interleaved. Given this, we expect that memory mapping will exhibit increasing tradeoffs as the NUMA system scales.

8.4 Conclusion

In discussing the generalization of NAVAR, three aspects are considered in this chapter. They are about the combination of the NAVAR policies, the applicability of NAVAR to a different machine, and the scalability of NAVAR for the increased NUMA domains. Some corner cases were addressed and the expected impacts from the scaled hardware attributes were speculated about. They are likely for the extensions for this work. Nonetheless, the applicability of NAVAR to a different machine at least demonstrates this adaptive system design is not ad hoc.

Chapter 9

Related Works

I now describe related work.

9.1 Virtual Core or Thread Mapping and Scheduling

Broadly, there are issues with the problem of thread (vcore) mapping and scheduling, particularly in virtualized systems. Synchronization-aware scheduling in a over-subscribed system (time-sharing perspectives) and contention-aware co-scheduling and co-location (space-sharing perspectives) are currently and heavily investigated topics.

Co-scheduling has been advocated in response to the issue of the synchronization mismatch delays between vcores in the same domain (VM) [92, 105, 50]. Some recent papers [116, 17, 115, 107, 72] argue co-scheduling or subset of co-scheduling attributes (balanced scheduling) in VMs should be selectively applied to mitigate CPU fragmentation, a priority inversion, and execution delays. Jiang et al. [64] theoretically formulate co-scheduling problem and suggest an offline-base optimal scheduling algorithm for dual cores. Kim et al. [72] discuss detectability on synchronization events in guests through monitoring inter-process interrupts. The approach is comparable with the para-virtualization based one that Weng et al. [115] proposed. In the sense that detection does not require any instrumentations in a guest, it is more applicable; nonetheless, a question remains of whether a detected synchronization event is always coupled with co-scheduling benefits on entire system performance. Lee et al. [80] address time scheduling policies for soft real-time applications in VMs. To minimize latency, which is an important issue for these workloads, cache-affinity and load-balance to prevent cache thrashing are incorporated. Xu et al. [117] suggest a vcore-scheduling policy that prioritizes latency-senstive vcores of I/O-bound applications and schedules them with a smaller time slice for fairness.

Another group of recent studies [90, 22, 49] address contention issues by space sharing and propose co-scheduling mechanisms to leverage resource sharing for the tasks that are relevant and running on a chip multicore processor. Furthermore, Bhadauria et al. [22] interestingly argue that space-sharing is more important factor than than time-sharing. Therefore, they treat contention-awareness more importantly. In this branch of work, there is much work being done to quantify the amount of contention and address mapping problems. In investigating mapping policies for shared resource contention on caches, Zhuravlev et al. [124] use online classification methods particularly with cache miss rates base characterization. Zhang et al [122] demonstrate a compile-time based static analysis on block usages between threads and suggest combined thread mapping and scheduling strategies leveraging data locality. Tang et al. [109] utilize performance counters to profile memory-access characteristic reflected in last-level-cache (LLC) misses, bus transactions and MESI states of LLC requests, then take a heuristic approach to find the best performing thread to core mappings. Hotmeyr et al. [61] take a white box approach. They implement a framework at a user-level that tries to redistributing threads balanced to processors. Chen et al. [31] propose a profiler design to collect very fine-grained and low-level information such as longest instruction dependency chain in an instruction window, and primarily focus on the partitioning of shared resources to address contention management. Verma et al. [111] address consolidation policies for the energy efficiency for large scaled systems. Scale and constraints of the problem are a bit different from above; nonetheless, significant portions of the work are put on analyzing characteristics and patterns of VMs' workloads. They compare two VM mapping (co-locating) policies and find that the policy based on the correlations in peak-time loads brings better results than the other based on the correlations overall time.

One of the key assumptions about the two issues is either over-subscription or contention in resource utilization. This raises the question of how many threads or VMs are allowable in a system, the admission control problem. Johnson et al. [65] argue that contention management as a matter of controlling the number of running threads should be decoupled from scheduling. A dedicated controller interacts with applications via a *sleep slot buffer*. With that, the controller determines the number of runnable threads which are allowed to use synchronization mechanisms. Gupta and Harchol-Balter [55] investigate admission control design for interactive workloads. They use queuing theory to analyze the performance in terms of response time and propose dynamic control schemes using either fluid control or stochastic dynamic programming. Lee et al. [79] suggest an offline static analysis technique to construct a communication graph; then, dynamic compilation selects and combines some threads to reduce unnecessary synchronization cost.

Energy, power, and/or thermal aware scheduling and mapping are other hot issues, as scalability on multicore processors becomes an issue [47, 59]. A group of people [37, 38, 36] investigate optimal mappings in a multicore system for power and performance. They try to solve this with various approaches by combining performance and power models linearly [37], by using machine learning technique particularly artificial neural networks [38], or by using prediction models created by multivariate regression and runtime data analysis [36]. Li et al. [81] propose a task placement scheme based on analyzing task aggregation patterns and argue that the scheme minimizes energy with performance constraints. Much thermal-aware work is also conducted. Rangan et al. [95] address thermal migrations to cores with fixed voltages and frequencies, instead of doing expensive dynamically changes of voltage and frequency in-situ. Ge et al [51] and Yeo et al [119] investigate thermal-aware core migration and scheduling that reduce spatial temperature variation which increases clock skews and decreases performance and reliability. To address the thermal hot-spot issue, Hanumaiah et al. [58] formulate the problem and do modeling with analytical circuit formulations. Interestingly enough, from a very different aspect, Coskun et al [35] address thermal effects on reliability and chip lifetime. They argue that the thermal cycling effect, which comes from fluctuation of cold and heat, is more negative than peak thermal effects; thus, moderate- or conservative-DVFS, migration and balanced resource usage schemes are suggested.

VMM design for emerging heterogeneous multicores has been discussed either for asymmetric scheduling [75, 70] or for a uniform resource usage model [56].

9.2 Page Allocation and Memory Scheduling Schemes

Page mapping and memory-access scheduling have been recently investigated. Page mappings are particularly addressed in systems with massive scale multicores and/or multiple nodes. Dashit et al [40] investigate memory traffic effects on modern NUMA multicore systems and suggest an adaptive memory mapping policy called *CarreFour*. One point that they put emphasis on is that memory contention on the memory controller and the bus is a more significant factor than memory-access distances in NUMA. This result is aligned with NAVAR for memory mapping concerns. However, NAVAR precisely addresses the tradeoffs in power and performance in leveraging memory bandwidth. *CarreFour* utilizes memories in all nodes, and they try to improve performance further with mechanisms such as memory replication. Blagodurov et al [28] suggest strategies for migrating both threads and pages to manage contention in memory controllers and interconnects in a NUMA machine. Particularly, they argue that page migrations are necessary when co-located threads hurt each other in performance. Li et al [84] address a projection of the increasing gap between memory bandwidth and core counts per a socket. They propose a scheme to extend the memory space over different nodes through page-swapping via VMs or through fine-grained block accesses with hardware modifications. Some suggestions for enhancing the memory scheduler are made, albeit for hardware. Given that previous off-chip memory scheduling policies are too extreme in either maximizing throughput or maximizing fairness, Kim et al [73] propose a hybrid memory scheduling scheme. They suggest a hardware framework that monitors threads, groups them in two classes and schedules with two policies accordingly. Ebrahimi et al [46] study the effects of prefeching, which is a very popular mechanism in modern memory sub-systems. They capture memory starvation for workloads that are not memory-intensive; therefore, hardware base perfetching coordination schemes are proposed. Similarly, Lin et al [85] use partitioning of both threads and pages to address tradeoffs in performance, power, and heat, and apply different scheduling policies and memory power modes to the groups. Additionally, Sudan et al. [106] argue that the co-location of different OS pages (fine-grained data placement) in row buffers increases their utilization, observing that memory-accesses from applications in a scaled multicore system are aligned with contiguous cache blocks.

9.3 Energy Proportionality

Considering energy-aware system design, power, and energy measurements is essential. Modeling based estimation has been heavily addressed in various domains and with various approaches [103, 120, 68, 44, 21]. Interestingly, Kansal et al. [68] suggest power modeling methodologies for VMs.

For energy proportionality, one of the best policies is intelligently turning off components that are and will be idle. However, power demands, particularly in data centers, have frequent bursty patterns [29, 20]; thus, Meisner et al. [87] propose a system that improves power conversion with proposed components' power state transitions (*PowerNap*) and a new power supply design (*RAILS*). Sharma et al. [99], in the meantime, describe cost-driven power optimization that tries to reach the best performance within a given power budget (*power-driven* approach). The cost-driven power optimization has more importance than the *workload-driven* approach that has been mostly pursued, because emerging use of intermittent renewable energy sources is increaseing. Therefore, they argue for an intermediate power state with a middle duty cycle instead of traditional power states, which are either fully active (100%) or inactive (0%), using *PowerNap*'s fast state transition concept. Snowdon et al. [104] experiment with a *workload-driven* approach on various types of systems thoroughly. Lim et al [83] take *power-driven* approaches for virtualized data servers. They use measurement techniques shown in [68], monitoring each VM's power, defining control levels like top-, application-, and tier-level for data center applications and then applying different power controls at each level. The lower the level, the more controllability is given. Meng et al [89] study various runtime power optimizations and propose an analytic modeling based approach to meet a power budget through runtime adaptation of multiprocessor cores. Motivated by the fact that online data-intensive workloads have dynamic load ranges, Meisner et al. [88] investigate each module's energy proportionality and its opportunities. They conclude memory system power in particular and *PowerNap*, at least for their workloads, need improvements.

For this reason, recently various works propose enhanced memory systems for power. David et al [41] and Deng et al [42], almost at the same time, suggested dynamically scaled voltage and frequency mechanisms and a control scheme for a memory sub-system. Similarly, Chen et al [34] applied an advanced optimal MIMO control theory to control both CPU and memory power for achieving system power proportionality. The number of active racks is chosen as the knob for memory power control. Isen et al [63] propose a technique to eliminate unnecessary DRAM refreshes by leveraging program semantic information. They keep track of memory state in context of memory allocations by software. Similarly, Amin et al. [12] try to prolong the idle time of rank idleness. Since the idea is delaying memory-accesses, it invokes reconsideration of both the cache replacement policy and the write buffer; nonetheless, they show results with 5 – 10 % improved energy efficiency over state-of-the-art DRAM design.

Accounting for the total cost of power supply, cooling power is not an insignificant portion any more as heat dissipation keeps increasing. Ahmad et al [7] consider power management schemes for large scale systems. They figure out previous approaches are extreme, either focusing on power reductions by increasing hot spots or balancing loads across machines causing idle power on more machines. They try to find the optimal number of active machines to address tradeoffs, and propose management schemes at both coarse and fine time-granularities to adapt to the pattern of resource demands. Abbasi et al [5] and Pakbaznia et al [94] formulate the problem of thermal-aware active server provisioning with nonlinear binary integer programming, and minimize data center power, including server system power, and cooling power, with linear programming. They solve the problems with proposed heuristics or running LP solver, and both evaluate their approaches using simulation.

9.4 Monitoring

Software based monitoring is particularly attractive for providing context information, expecially when multiple applications or multi-threaded workloads run. Zhang et al [121] show a detection mechanism that scans page tables to identify hot pages in order to manage cache space efficiently. This detection scheme is very similar to that in NAVAR. Zhao et al [123] try to figure out the memory-access patterns of a guest OS in a VM through the interception of a set of pages that are determined to be less frequently accessed. Miós et al [91] propose sharing-aware block devices integrated with guest OS modifications. The suggested mechanism is argued to enable page sharing detection like zero-page detection, page-table sharing, and kernel text sharing. Lu et al. [86] point out resource usage profiling problems on VM is hard; thus, they formulate the problem as a source separation problem and suggested solutions that use direct factor graph base models and runtime calibration mechanisms. They demonstrate their methodology in real systems.

Hardware based detection schemes provide fine-grained information with low overhead; thus, there are cases utilizing hardware monitors to characterize workloads and suggesting software incorporation. Furthermore, extensions or redesign of the performance counters are proposed. Kim et al [73] classify threads in multiple workloads based on memory bandwidth usage leveraging the statistics of L2 cache and memory controller events. Diman et al [43] utilize conventional architectural performance metrics such as cache miss rate and IPC, to identify cache sensitivity. Metrics are used for characterizing and scheduling threads in a heterogeneous set of workloads. Similarly, Bhattacharjee and Martonosi [25] investigate the most correlated metrics to critical threads in multi-threaded workloads. They argue thread criticality is one of most fundamental factors and should be considered in scheduling or load balancing. They propose a new hardware counter that monitors typical PMU events and additionally raises alarms if the combination of this information goes beyond a threshold. Estimating cache contention between threads is important to determine scheduling policies. Zhuravlev et al [124] compare a miss rate scheme with various profiling approaches including the most well-known stack distance profiles [30]. They argue a scheme using only the miss rate is a far more attractive option for online classification. Integraty with performance counters, some recent papers suggest the design of stimulus software. Govindan et al [54] try online estimation of cache pressure by introducing a stimulus benchmark per application. It is motivated by the fact that how to detect cache contention under multiple VMs is unclear. Hackenberg et al [57] develop benchmark suites to measure cache and memory performance, particularly cache coherency.

9.5 Control Scheme

Some recent works design control schemes according to a closed loop feedback structure. Sharifi et al [98] fit their system design into the scheme of a Single-Input, Multiple-Output controller with an Auto-Regressive-Moving-Average (ARMA) system model, to address contention management in multicore systems. They use three separate controllers for core, cache, and bandwidth and multiple outputs reflected in the CPI for each application over time. Padala et al [93] similarly use ARMA for the system model and use a Multi-Input, Multi-Output (MIMO) type of feedback controller in the context of data center with virtualization, where these are as many controls and outputs as these are nodes and applications running. Kalyvianaki et al [67] propose a resource control scheme, for a virtualized cluster, that uses the Kalman filter to monitor utilization with transient fluctuations, assuming that the system is described in a linear model and noises are white and Gaussian, then the control scheme updates allocations. They also suggest variant designs to the input numbers and to the noise patterns. Papers focus on a control scheme itself by using utility functions or models, rather than fitting the system model into the feedback system structure. Xu et al [118] optimize VM placement in datacenters for multiple objectives through cross-layer control. They use a single objective function based on a weighted sum of normalized utility functions for each metric. Chen et al [33] consider constraints of performance, provider's profit and user satisfaction based on Service Level Agreements (SLAs), and build a utility model with straightforward relationships. Lim et al [83] and Wang et al [114] both mention Model Predictive Control (MPC) that combines output prediction, optimization and control into a single algorithm, to address problems like budget driven power management in a virtual cluster, and chip-level power management with constraints for temperature per core.

The following control schemes are MPC based approaches. Gong et al [52] estimates resource demands in distributed systems by a hybrid approach, first employing a fast Fourier transformation to identify repeating patterns, and if a signature is not found, applying a discrete-time Markov chain. Then, they address prediction error handling and correction in [101]. Sheikh et al [100] use a Byesian approach to estimate database query time on virtualized servers, noticing the previous Gaussian Processing model costs too much. Urgankar et al [110] use Lyapunov techniques to get approximately optimized online control over the power supply to solve a power cost management problem in data centers. They are motivated by the fact that a popular approach using either Markov Decision theory or Dynamic Programming is unsuitable with large scale. Chen et al [32] address the chip-level resource management problem using performance models through gradient performance gains and augment a conventional stack distance model. With that, they estimate resource demands in more architectural-level to configure memory bandwidth and branch predictor size.

Machine learning technique based modeling approaches are also investigated for adaptive system design. Particularly, this approach is useful for modeling nonlinear system characteristics. Rao et al [96] use a reinforcement learning (RL) based approach to automate the VM configuration process. Tan et al [108] also use RL for system power management in a partially observable environment. Instead of selecting one among existing solutions, they argue instead to learn the policy through a model-free approach. Bitirgen et al [27] use Artificial Neural Networks (ANNs) to address the resource allocation of heterogeneous threads to cores. Ge et al [51] use neural network predictor to build thermal models while arguing previous studies that are mostly history-based are inaccurate and have overhead. Cochran et al [39] identify multiple objectives for energy efficiency. To address this, they proposed a multinomial logistic regression online classifier. Also they use L_1-regularization techniques to select inputs for minimizing overhead and over-fitting.

This chapter discusses related work in the five areas: virtual core (thread) scheduling, memory allocation and mapping, energy efficiency, workload monitoring, and control scheme for dynamic adaptive system. Much related works can be found. Especially the top of virtual core scheduling (co-scheduling) and NUMA-aware memory mapping have been and are actively researched; nonetheless, my investigation of the system optimizations (including energy efficiency) for the virtualized NUMA multicore system (for both virtualization and the NUMA system) is unique. The contributions will be mentioned in the next chapter, along with a summary of all the previous chapters and potential future works.

Chapter 10

Conclusion

In the context of a virtualized NUMA multicore system, significant tradeoffs are observed on resource mapping and scheduling. The tradeoffs can be reduced to three major problems: **`vcore-mapping`**, **`page-mapping`**, and **`vcore-scheduling`**. I claim that the inference-based approach can solve these resource optimization problems. Different from the guest OS, the VMM can provide scalability and also have observability of the context information of the guest OS. Compared to the host OS, the VMM also has advantages in capturing the VM context as well as the guest OS context, with the same scalability.

Since the three problems occur in combination, the situations that create specific problems were classified to be solved. The three classes are as under-subscription, full-subscription, and over-subscription, as defined by the ratio of active vcores to the available number of hardware threads in the physical machine. In the under-subscription case, the **`vcore-mapping`** and **`page-mapping`** problems are found to be the key problems. In the full-subscription case, the **`page-mapping`** problem is critical. The **`vcore-scheduling`** problem arises in the over-subscription case, where the memory mapping problem can be addressed by a static policy.

The policy component for the NAVAR adaptive system was designed for each resource demand class. For the under- and full-subscription cases, the model-based prediction approach works well. For the over-subscription case, a brute-force approach is taken for the policy. For the under- and full-subscription cases, the brute-force approach, a simple heuristic, was also tried and compared, but it failed due to the overhead of system reconfigurations.

Along with this summary of the work, this chapter lists the contributions that my dissertation gives and discusses some future work.

10.1 Contributions

The five major contributions made by my dissertation are:

- **A proof-of-concept of the inference-based approach in the VMM context for optimizing a virtualized NUMA multicore system.** As claimed, the black box based approach can solve the system optimization problem. The white box based approach may improve the system performance with speculative resource managements; nonetheless, the inference-based model is viable in the coarse-grained level of resource management problem. Moreover, this approach is likely to be beneficial for larger scale system management.

- **The design, implementation, and validation of the NAVAR adaptive system.** An adaptive system was actually designed, implemented, and evaluated. When an offline based model (the proof-of-concept) is embedded into the system, the system works, resulting in performance, energy, or power, comparable to the best static configuration for the workload. At runtime, the system manages its own overhead and the overhead of system reconfigurations.

- **Identification of the tradeoffs in performance, power, and energy in a virtualized NUMA multicore system.** It is well-known that the NUMA multicore system incorporates memory access tradeoffs, and recently studies of the tradeoffs have focused on the memory access bandwidth as the remote memory access penalty is decreasing with hardware developments [40]. However, tradeoffs in energy are not yet well studied. These are clarified and addressed through my experiments.

- **Design and implementation of NUMA-aware vcore scheduling (co-scheduling)** The NUMA-aware co-scheduling model was proposed and evaluated. The vcore scheduler is also optimized to reduce possible side effects.

- **Proposal and quantification of a new hardware feature that allows powering-off the memory of a NUMA domain, if it is unused.** With real measurements, the potential energy savings by the proposed hardware mechanism were quantified. This proposed hardware mechanism is leveraged by the NAVAR adaptive system, which demonstrates the effects clearly.

10.2 Future Work

Following are potential items for future work. In a short term, deploying the NAVAR adaptive system on large scale NUMA multicore systems would validate for its scalability, and generality.

Large scale NUMA multicore systems Our testbeds incorporate only two NUMA domains. The discussions of Chapter 8 lay out the expected factors by having on increased number of NUMA domains. Nonetheless, experimental tests will disclose the real interactions between workloads' memory traffic and the increased number of NUMA domains, and the effectiveness of NAVAR.

Workloads In principle, the NAVAR policy is not specific to the workloads. The NAVAR adaptive system could achieve node-level energy efficiency even when resource demand is fluctuating, which would make it suited if nodes in the datacenter. Evaluating NAVAR in such a domain would be an important study.

Comparison with a guest-level approach Besides more evaluations, the NAVAR adaptive system could be compared with a different black box-based approach. In particularly, commodity OSes increasingly support NUMA-aware system optimizations themselves. Exposing the NUMA topology from the VMM can activate the guest OS's optimizations. Comparisons between the VM model with respect to performance results may give insight into the merits of both approach.

Bibliography

[1] Pinpoints. http://www.pintool.org/pinpoints.html.

[2] Puppy linux. http://puppylinux.org.

[3] SPEC CPU. http://www.spec.org.

[4] *Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes 1, 2A, 2B, 2C, 3A, 3B, and 3C*, June 2013.

[5] Zahra Abbasi, Georgios Varsamopoulos, and Sandeep K. S. Gupta. Thermal aware server provisioning and workload distribution for internet data centers. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC)*, June 2010.

[6] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems (ASPLOS)*, October 2006.

[7] Faraz Ahmad and T. N. Vijaykumar. Joint optimization of idle and cooling power in data centers while maintaining response time. In *Proceedings of the 15th edition of*

ASPLOS on Architectural support for programming languages and operating systems (ASPLOS), March 2010.

[8] A.R. Alameldeen and D.A. Wood. IPC Considered Harmful for Multiprocessor Workloads. *IEEE Micro*, 26(4):8–17, July–August 2006.

[9] AMD, Inc. *Software Optimization Guide for AMD Family 10h and 12h Processors*, February 2011.

[10] AMD, Inc. *AMD64 Architecture Programmer's Manual Vol 2: System Programming*, May 2013.

[11] AMD, Inc. *BIOS and Kernel Developer's Guide for AMD Family 10h Processors*, January 2013.

[12] Ahmed M. Amin and Zeshan A. Chishti. Rank-aware cache replacement and write buffering to improve DRAM energy efficiency. In *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design (ISLPED)*, August 2010.

[13] Vishal Aslot, Max Domeika, Rudolf Eigenmann, Greg Gaertner, Wesley B. Jones, and Bodo Parady. SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. In *Proceedings of the Workshop on OpenMP Applications and Tools (WOMPAT)*, July 2001.

[14] Chang Bae, John R. Lange, and Peter A. Dinda. Comparing approaches to virtualized page translation in modern vmms. Technical Report NWU-EECS-10-07, Department of Electrical Engineering and Computer Science, Northwestern University, April 2010.

[15] Chang Bae and Jamel Tayeb. Energy-aware memory management through database buffer management. In *Third Workshop on Energy-Efficient Design (WEED)*, June 2011.

[16] Chang S. Bae, John R. Lange, and Peter A. Dinda. Enhancing virtualized application performance through dynamic adaptive paging mode selection. In *Proceedings of the 8th ACM international conference on Autonomic computing (ICAC)*, June 2011.

[17] Yuebin Bai, Cong Xu, and Zhi Li. Task-aware based co-scheduling for virtual machine system. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC)*, March 2010.

[18] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM Symposium on Operating Systems Principles (SOSP)*, October 2003.

[19] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Translation caching: skip, don't walk (the page table). In *Proceedings of the 37th annual international symposium on Computer architecture (ISCA)*, June 2010.

[20] Cullen Bash and George Forman. Cool job allocation: measuring the power savings of placing jobs at cooling-efficient locations in the data center. In *Proceedings of the 2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, June 2007.

[21] Ramon Bertran, Marc Gonzalez, Xavier Martorell, Nacho Navarro, and Eduard Ayguade. Decomposable and responsive power models for multicore processors using performance counters. In *Proceedings of the 24th ACM International Conference on Supercomputing (ICS)*, June 2010.

[22] Major Bhadauria and Sally A. McKee. An approach to resource-aware co-scheduling for CMPs. In *Proceedings of the 24th ACM International Conference on Supercomputing (ICS)*, June 2010.

[23] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating two-dimensional page walks for virtualized systems. In *in Proceedings of the 13th international conference on Architectural support for programming languages and operating systems (ASPLOS)*, March 2008.

[24] Abhishek Bhattacharjee and Margaret Martonosi. Characterizing the TLB behavior of emerging parallel workloads on chip multiprocessors. In *Proceedings of the 8th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, October 2009.

[25] Abhishek Bhattacharjee and Margaret Martonosi. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. In *Proceedings of the 36th annual international symposium on Computer architecture (ISCA)*, June 2009.

[26] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, October 2008.

[27] Ramazan Bitirgen, Engin Ipek, and Jose F. Martinez. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *2008 41st IEEE/ACM International Symposium on Microarchitecture (Micro)*, December 2008.

[28] Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. A case for NUMA-aware contention management on multicore systems. In *Proceedings of the 2011 conference on USENIX Annual technical conference (USENIX ATC)*, June 2011.

[29] Pat Bohrer, Elmootazbellah N. Elnozahy, Tom Keller, Michael Kistler, Charles Lefurgy, Chandler McDowell, and Ram Rajamony. Power aware computing. chapter The case for power management in web servers, pages 261–289. Kluwer Academic Publishers, Norwell, MA, USA, May 2002.

[30] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPDC)*, June 2005.

[31] Jian Chen and Lizy Kurian John. Predictive coordination of multiple on-chip resources for chip multiprocessors. In *Proceedings of the international conference on Supercomputing (ICS)*, June 2011.

[32] Jian Chen, Lizy Kurian John, and Dimitris Kaseridis. Modeling program resource demand using inherent program characteristics. In *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems (SIGMETRICS)*, June 2011.

[33] Junliang Chen, Chen Wang, Bing Bing Zhou, Lei Sun, Young Choon Lee, and Albert Y. Zomaya. Tradeoffs Between Profit and Customer Satisfaction for Service Provisioning in the Cloud. In *Proceedings of the 20th international symposium on High performance distributed computing (HPDC)*, June 2011.

[34] Ming Chen, Xiaorui Wang, and Xue Li. Coordinating processor and main memory for efficientserver power control. In *Proceedings of the international conference on Supercomputing (ICS)*, June 2011.

[35] Ayse K. Coskun, Richard Strong, Dean M. Tullsen, and Tajana Simunic Rosing. Evaluating the impact of job scheduling and power management on processor lifetime for chip multiprocessors. In *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems (SIGMETRICS)*, June 2009.

[36] Matthew Curtis-Maury, Filip Blagojevic, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Prediction-Based Power-Performance Adaptation of Multithreaded Scientific Codes. *IEEE Trans. Parallel Distrib. Syst.*, 19(10):1396–1410, October 2008.

[37] Matthew Curtis-Maury, James Dzierwa, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In *Proceedings of the 20th annual international conference on Supercomputing (ICS)*, June 2006.

[38] Matthew Curtis-Maury, Karan Singh, Sally A. McKee, Filip Blagojevic, Dimitrios S. Nikolopoulos, Bronis R. de Supinski, and Martin Schulz. Identifying energy-efficient concurrency levels using machine learning. In *Proceedings of the 2007 IEEE International Conference on Cluster Computing (CLUSTER)*, September 2007.

[39] Matthew Curtis-Maury, Karan Singh, Sally A. McKee, Filip Blagojevic, Dimitrios S. Nikolopoulos, Bronis R. de Supinski, and Martin Schulz. Identifying energy-efficient concurrency levels using machine learning. In *Proceedings of the*

2007 IEEE International Conference on Cluster Computing (CLUSTER), September 2007.

[40] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic management: a holistic approach to memory placement on numa systems. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems (ASPLOS)*, March 2013.

[41] Howard David, Chris Fallin, Eugene Gorbatov, Ulf R. Hanebutte, and Onur Mutlu. Memory power management via dynamic voltage/frequency scaling. In *Proceedings of the 8th ACM international conference on Autonomic computing (ICAC)*, June 2011.

[42] Qingyuan Deng, David Meisner, Luiz Ramos, Thomas F. Wenisch, and Ricardo Bianchini. MemScale: active low-power modes for main memory. In *Proceedings of the 16th international conference on Architectural support for programming languages and operating systems (ASPLOS)*, March 2011.

[43] Gaurav Dhiman, Vasileios Kontorinis, Dean Tullsen, Tajana Rosing, Eric Saxe, and Jonathan Chew. Dynamic workload characterization for power efficient scheduling on CMP systems. In *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design (ISLPED)*, October 2010.

[44] Gaurav Dhiman, Kresimir Mihic, and Tajana Rosing. A system for online power prediction in virtualized environments using Gaussian mixture models. In *Proceedings of the 47th Design Automation Conference (DAC)*, June 2010.

[45] Mian Dong and Lin Zhong. Self-constructive high-rate system energy modeling for battery-powered mobile systems. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, June 2011.

[46] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. Prefetch-aware shared resource management for multi-core systems. In *Proceedings of the 38th annual international symposium on Computer architecture (ISCA)*, June 2011.

[47] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*, June 2011.

[48] Stijn Eyerman and Lieven Eeckhout. System-Level Performance Metrics for Multiprogram Workloads. *IEEE Micro*, 28:42–53, May 2008.

[49] Stijn Eyerman and Lieven Eeckhout. Probabilistic job symbiosis modeling for SMT processor scheduling. In *Proceedings of the 15th edition of ASPLOS on Architectural support for programming languages and operating systems (ASPLOS)*, March 2010.

[50] Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Parallel job scheduling – a status report. In *Proceedings of the 10th international conference on Job Scheduling Strategies for Parallel Processing (JSSPP)*, June 2005.

[51] Yang Ge, Parth Malani, and Qinru Qiu. Distributed task migration for thermal management in many-core systems. In *Proceedings of the 47th Design Automation Conference (DAC)*, June 2010.

[52] Zhenhuan Gong, Xiaohui Gu, and J. Wilkes. PRESS: PRedictive Elastic ReSource Scaling for cloud systems. In *Proceedings of International Conference on Network and Service Management (CNSM)*, Octoboer 2010.

[53] Mel Gorman and Patrick Healy. Performance characteristics of explicit superpage support. In *Proceedings of the 6th Annual Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA)*, June 2010.

[54] Sriram Govindan, Jie Liu, Aman Kansal, and Anand Sivasubramaniam. Cuanta: quantifying effects of shared on-chip resource interference for consolidated virtual machines. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC)*, October 2011.

[55] Varun Gupta and Mor Harchol-Balter. Self-adaptive admission control policies for resource-sharing systems. In *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems (SIGMETRICS)*, June 2009.

[56] Vishakha Gupta, Karsten Schwan, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. Pegasus: coordinated scheduling for virtualized accelerator-based systems. In *Proceedings of the 2011 conference on USENIX Annual technical conference (USENIX ATC)*, June 2011.

[57] Daniel Hackenberg, Daniel Molka, and Wolfgang E. Nagel. Comparing cache architectures and coherency protocols on x86-64 multicore SMP systems. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (Micro)*, December 2009.

[58] Vinay Hanumaiah, Ravishankar Rao, Sarma Vrudhula, and Karam S. Chatha. Throughput optimal task allocation under thermal constraints for multi-core processors. In *Proceedings of the 46th Annual Design Automation Conference (DAC)*, July 2009.

[59] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31(4):6–15, July–August 2011.

[60] Giang Hoang, Chang Bae, John Lange, Lide Zhang, Peter Dinda, and Russ Joseph. A case for alternative nested paging models for virtualized systems. *Computer Architecture Letters*, 9(1):17–20, January 2010.

[61] Steven Hofmeyr, Juan A. Colmenares, Costin Iancu, and John Kubiatowicz. Juggle: proactive load balancing on multicore computers. In *Proceedings of the 20th international symposium on High performance distributed computing (HPDC)*, June 2011.

[62] IBM. Kernel Virtual Machine (KVM) Best Practices for KVM. Technical report, April 2012.

[63] Ciji Isen and Lizy John. ESKIMO: Energy savings using Semantic Knowledge of Inconsequential Memory Occupancy for DRAM subsystem. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (Micro)*, December 2009.

[64] Yunlian Jiang, Xipeng Shen, Jie Chen, and Rahul Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *Proceedings of the 17th international conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[65] F. Ryan Johnson, Radu Stoica, Anastasia Ailamaki, and Todd C. Mowry. Decoupling contention management from scheduling. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2010.

[66] Vivek Kale. Towards using and improving the nas parallel benchmarks: a parallel patterns approach. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns (ParaPLoP)*, March 2010.

[67] Evangelia Kalyvianaki, Themistoklis Charalambous, and Steven Hand. Self-adaptive and self-configured CPU resource provisioning for virtualized servers using Kalman filters. In *Proceedings of the 6th international conference on Autonomic computing (ICAC)*, June 2009.

[68] Aman Kansal, Feng Zhao, Jie Liu, Nupur Kothari, and Arka A. Bhattacharya. Virtual machine power metering and provisioning. In *Proceedings of the 1st ACM symposium on Cloud computing (SOCC)*, June 2010.

[69] Paul A. Karger. Performance and security lessons learned from virtualizing the alpha processor. In *Proceedings of the 34th annual international symposium on Computer architecture (ISCA)*, June 2007.

[70] Vahid Kazempour, Ali Kamali, and Alexandra Fedorova. AASH: an asymmetry-aware scheduler for hypervisors. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual Execution Environments (VEE)*, March 2010.

[71] Samir Khuller, Jian Li, and Barna Saha. Energy efficient scheduling via partial shutdown. In *Proceedings of the 21th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, January 2010.

[72] Hwanju Kim, Sangwook Kim, Jinkyu Jeong, Joonwon Lee, and Seungryoul Maeng. Demand-based coordinated scheduling for smp vms. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems (ASPLOS)*, March 2013.

[73] Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (Micro)*, December 2010.

[74] Brian Kocoloski, Jiannan Ouyang, and John Lange. A case for dual stack virtualization: consolidating hpc and commodity applications in the cloud. In *Proceedings of the Third ACM Symposium on Cloud Computing (SOCC)*, October 2012.

[75] Youngjin Kwon, Changdae Kim, Seungryoul Maeng, and Jaehyuk Huh. Virtualizing performance asymmetric multi-core systems. In *Proceedings of the 38th annual International Symposium on Computer Architecture (ISCA)*, June 2011.

[76] Jack Lange, Peter Dinda, Kyle Hale, and Lei Xia. An introduction to the palacios virtual machine monitor—release 1.3. Technical Report NWU-EECS-11-10, Department of Electrical Engineering and Computer Science, Northwestern University, October 2011.

[77] John Lange, Kevin Pedretti, Peter Dinda, Chang Bae, Patrick Bridges, Philip Soltero, and Alexander Merritt. Minimal-overhead virtualization of a large scale supercomputer. In *Proceedings of the 2011 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, March 2011.

[78] John Lange, Kevin Pedretti, Trammell Hudson, Peter Dinda, Zheng Cui, Lei Xia, Patrick Bridges, Andy Gocke, Steven Jaconette, Mike Levenhagen, and Ron Brightwell. Palacios and kitten: New high performance operating systems for scalable virtualized and native supercomputing. In *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, March 2010.

[79] Janghaeng Lee, Haicheng Wu, Madhumitha Ravichandran, and Nathan Clark. Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA)*, June 2010.

[80] Min Lee, A. S. Krishnakumar, P. Krishnan, Navjot Singh, and Shalini Yajnik. Supporting soft real-time tasks in the Xen hypervisor. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems (ASPLOS)*, March 2010.

[81] Dong Li, Dimitrios S. Nikolopoulos, Kirk W. Cameron, Bronis R. de Supinski, and Martin Schulz. Power-aware mpi task aggregation prediction for high-end computing systems. In *Proceedings of the 24th IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, April 2010.

[82] Jacob Liberman and Garima Kochhar. *Optimal BIOS Settings for High Performance Compting with PowerEdge 11G Servers*. Dell Product Group, August 2010.

[83] Harold Lim, Aman Kansal, and Jie Liu. Power budgeting for virtualized data centers. In *Proceedings of the 2011 conference on USENIX Annual technical conference (USENIX ATC)*, June 2011.

[84] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, June 2009.

[85] Chung-Hsiang Lin, Chia-Lin Yang, and Ku-Jei King. PPT: Joint Performance/Power/Thermal Management of DRAM Memory for Multi-Core Systems. In *Proceedings of the 14th ACM/IEEE international symposium on Low Power Electronics and Design (ISLPED)*, December 2009.

[86] Lei Lu, Hui Zhang, Guofei Jiang, Haifeng Chen, Kenji Yoshihira, and Evgenia Smirni. Untangling mixed information to calibrate resource utilization in virtual machines. In *Proceedings of the 8th ACM International Conference on Autonomic Computing (ICAC)*, June 2011.

[87] David Meisner, Brian T. Gold, and Thomas F. Wenisch. PowerNap: eliminating server idle power. In *Proceedings of the 14th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2009.

[88] David Meisner, Christopher M. Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F. Wenisch. Power management of online data-intensive services. In *Proceeding of the 38th Annual International Symposium on Computer Architecture (ISCA)*, June 2011.

[89] Ke Meng, Russ Joseph, Robert P. Dick, and Li Shang. Multi-optimization power management for chip multiprocessors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques (PACT)*, October 2008.

[90] Andreas Merkel, Jan Stoess, and Frank Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *Proceedings of the 5th European conference on Computer systems (EuroSys)*, April 2010.

[91] Grzegorz Miós, Derek G. Murray, Steven Hand, and Michael A. Fetterman. Satori: enlightened page sharing. In *Proceedings of the 2009 conference on USENIX Annual technical conference (USENIX ATC)*, June 2009.

[92] J. Ousterhout. Scheduling techniques for concurrent systems. In *Third International Conference on Distributed Computing Systems (ICDCS)*, June 1982.

[93] Pradeep Padala, Kai-Yuan Hou, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. Automated control of multiple virtualized resources. In *Proceedings of the 4th ACM european conference on Computer systems (EuroSys)*, March 2009.

[94] Ehsan Pakbaznia and Massoud Pedram. Minimizing data center cooling and server power costs. In *Proceedings of the 14th ACM/IEEE international symposium on Low power electronics and design (ISLPED)*, December 2009.

[95] Krishna K. Rangan, Gu-Yeon Wei, and David Brooks. Thread motion: fine-grained power management for multi-core systems. In *Proceedings of the 36th annual international symposium on Computer architecture (ISCA)*, June 2009.

[96] Jia Rao, Xiangping Bu, Cheng-Zhong Xu, Leyi Wang, and George Yin. VCONF: a reinforcement learning approach to virtual machines auto-configuration. In *Proceedings of the 6th international conference on Autonomic computing (ICAC)*, June 2009.

[97] Suzanne Rivoire, Parthasarathy Ranganathan, and Christos Kozyrakis. A comparison of high-level full-system power models. In *Proceedings of the 2008 Workshop on Hot Topics in Power-aware Computing and Systems (HotPower)*, December 2008.

[98] Akbar Sharifi, Shekhar Srikantaiah, Asit K. Mishra, Mahmut Kandemir, and Chita R. Das. METE: meeting end-to-end QoS in multicores through system-wide resource management. In *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems (SIGMETRICS)*, June 2011.

[99] Navin Sharma, Sean Barker, David Irwin, and Prashant Shenoy. Blink: managing server clusters on intermittent power. In *Proceedings of the 16th international conference on Architectural support for programming languages and operating systems (ASPLOS)*, March 2011.

[100] Muhammad Bilal Sheikh, Umar Farooq Minhas, Omar Zia Khan, Ashraf Aboulnaga, Pascal Poupart, and David J. Taylor. A bayesian approach to online performance modeling for database appliances using gaussian models. In *Proceedings of the 8th ACM international conference on Autonomic computing (ICAC)*, June 2011.

[101] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. CloudScale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC)*, October 2011.

[102] David B. Shmoys and Éva Tardos. An approximation algorithm for the generalized assignment problem. *Math. Program.*, 62(3):461–474, December 1993.

[103] Alex Shye, Benjamin Scholbrock, and Gokhan Memik. Into the wild: studying real user activity patterns to guide power optimizations for mobile architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (Micro)*, December 2009.

[104] David C. Snowdon, Etienne Le Sueur, Stefan M. Petters, and Gernot Heiser. Koala: a platform for OS-level power management. In *Proceedings of the fourth ACM european conference on Computer systems (EuroSys)*, March 2009.

[105] Peter Strazdins and John Uhlmann. A Comparison of Local and Gang Scheduling on a Beowulf Cluster. In *the 2004 IEEE International Conference on Cluster Computing (Cluster)*, September 2004.

[106] Kshitij Sudan, Niladrish Chatterjee, David Nellans, Manu Awasthi, Rajeev Balasubramonian, and Al Davis. Micro-pages: increasing DRAM efficiency with locality-aware data placement. In *Proceedings of the 15th edition of ASPLOS on Architectural support for programming languages and operating systems (ASPLOS)*, March 2010.

[107] Orathai Sukwong and Hyong S. Kim. Is co-scheduling too expensive for SMP VMs? In *Proceedings of the 6th conference on Computer systems (EuroSys)*, April 2011.

[108] Ying Tan, Wei Liu, and Qinru Qiu. Adaptive power management using reinforcement learning. In *Proceedings of the 2009 International Conference on Computer-Aided Design (ICCAD)*, November 2009.

[109] Lingjia Tang, Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. The impact of memory subsystem resource sharing on datacenter applications. In

Proceedings of the 38th annual international symposium on Computer architecture (ISCA), June 2011.

[110] Rahul Urgaonkar, Bhuvan Urgaonkar, Michael J. Neely, and Anand Sivasubramaniam. Optimal power cost management using stored energy in data centers. In *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems (SIGMETRICS)*, June 2011.

[111] Akshat Verma, Gargi Dasgupta, Tapan Kumar Nayak, Pradipta De, and Ravi Kothari. Server workload analysis for power minimization using consolidation. In *Proceedings of the 2009 conference on USENIX Annual technical conference (USENIX ATC)*, June 2009.

[112] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *in Proceedings of the 5th symposium on Operating systems design and implementation (OSDI)*, December 2002.

[113] Chengwei Wang, Karsten Schwan, Vanish Talwar, Greg Eisenhauer, Liting Hu, and Matthew Wolf. A flexible architecture integrating monitoring and analytics for managing large-scale data centers. In *Proceedings of the 8th ACM international conference on Autonomic computing (ICAC)*, June 2011.

[114] Yefu Wang, Kai Ma, and Xiaorui Wang. Temperature-constrained power control for chip multiprocessors with online model estimation. In *Proceedings of the 36th annual international symposium on Computer architecture (ISCA)*, June 2009.

[115] Chuliang Weng, Qian Liu, Lei Yu, and Minglu Li. Dynamic adaptive scheduling for virtual machines. In *Proceedings of the 20th international symposium on High performance distributed computing (HPDC)*, June 2011.

[116] Chuliang Weng, Zhigang Wang, Minglu Li, and Xinda Lu. The hybrid scheduling framework for virtual machine systems. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE)*, March 2009.

[117] Cong Xu, Sahan Gamage, Pawan N. Rao, Ardalan Kangarlou, Ramana Rao Kompella, and Dongyan Xu. vslicer: latency-aware virtual machine scheduling via differentiated-frequency cpu slicing. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing (HPDC)*, June 2012.

[118] Jing Xu and José Fortes. A multi-objective approach to virtual machine management in datacenters. In *Proceedings of the 8th ACM international conference on Autonomic computing (ICAC)*, June 2011.

[119] Inchoon Yeo and Eun Jung Kim. Temperature-aware scheduler based on thermal behavior grouping in multicore systems. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, April 2009.

[120] Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P. Dick, Zhuoqing Morley Mao, and Lei Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the 8th IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS)*, October 2010.

[121] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM european conference on Computer systems (EuroSys)*, March 2009.

[122] Yuanrui Zhang, Mahmut Kandemir, and Taylan Yemliha. Studying inter-core data reuse in multicores. In *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems (SIGMETRICS)*, June 2011.

[123] Weiming Zhao and Zhenlin Wang. Dynamic memory balancing for virtual machines. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE)*, March 2009.

[124] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the 15th edition of ASPLOS on Architectural support for programming languages and operating systems (ASPLOS)*, January 2010.

Appendix A

Dynamic Adaptive Virtualized Virtual Memory (DAV^{2}M)

A critical component of the overhead of a modern virtual machine monitor (VMM) for x86 or x64 hardware is the virtualization of address translation. Conceptually, a VMM introduces an additional layer of indirection that maps addresses that the guest OS believes are physical addresses, but called here guest physical addresses (GPAs), to the actual host physical addresses (HPAs). Address translation then is effectively from guest virtual addresses (GVAs) to GPAs and then to HPAs.

There are two general approaches to achieving this translation, shadow paging and nested paging. Both approaches have several variants. Shadow paging is a software-centric approach that flattens GVA$\rightarrow$GPA$\rightarrow$HPA translation to GVA$\rightarrow$HPA translation and implements this translation in a single page table hierarchy that is under the exclusive control of the VMM and combines guest and VMM intent. In contrast, nested paging is a hardware-centric approach in which the hardware provides for a second page table hierarchy that the VMM can use to separately maintain the GPA$\rightarrow$HPA mapping without having to involve itself in the guest's GVA→GPA decisions. Section A.1 describes these approaches, and their variants on the x86/x64 in more detail.

Although both approaches can achieve very low overhead, there is not a single best approach for minimizing overheads and hence maximizing application performance. Rather, the best approach depends on the paging workload of the application running in the VM. Further, even for a single application, the best approach may not be static, but may rather vary over time. For example, an HPC application with multiple computation phases might have a preferred paging approach for each phase. As another example, a long-running VM might very well have different applications execute during its lifetime, each of which may prefer a different paging approach. In Sections A.2 and A.3 we report on a study, based largely on HPC workloads, that supports these claims. The virtualized application performance, relative to a native environment, can vary by as much as 105% between the two modes.

Most modern VMMs for x86/x64 hardware, including the Palacios VMM [78] in which this work is implemented, offer several variants of both shadow and nested paging. In these VMMs, the selection of paging approach is made at the time the VM is instantiated and holds for the VM's lifetime. As explained in Section A.4, the Palacios VMM is extended to dynamically change the paging approach at *run-time*. As part of the handling of any exit to the VMM, it is possible to switch to a different approach.

A prototype policy is developed, which uses this mechanism to enhance the performance of applications as they execute. The policy is based on two metrics of paging performance. When shadow paging is in use, the metric is the rate of VMM exits related to paging, which is extremely easy to measure in the context of exit handling. When nested paging is used, the metric is the TLB miss rate, which is measured at virtually zero cost using a hardware performance counter. A third metric, cycles per instruction (CPI), is used to measure application performance. These metrics are further described in Section A.3. The metrics are valuable apart from the policy since they succinctly capture the performance of each paging approach and the effect on application performance.

The prototype policy, dynamically adaptive virtualized virtual memory ($\mathrm{DAV^2M}$), uses the mechanism and metrics to probe application performance using the current and alternate paging modes. Probing is triggered when the paging performance under the current mode exceeds a threshold. Based on this probing, the thresholds associated with each paging approach are also adjusted. Performance testing takes into account transient effects due to the paging approach switch itself, and probes are temporally limited to avoid potential oscillations in this control algorithm. When probing identifies a clearly superior paging approach, it is made the current one. $\mathrm{DAV^2M}$ is described in detail in Section A.5.

$\mathrm{DAV^2M}$ is implemented in Palacios and evaluated using the application benchmarks described in Section A.2 that are most sensitive to particular paging approaches, including benchmarks that are insensitive for comparison. Detailed results are shown in Section A.6. The most salient points are as follows. First, for workloads which have a single best paging approach, $\mathrm{DAV^2M}$ is able to quickly converge on that approach. Because of this quick convergence and the reasonably low overhead of the mechanism for switching the paging approach, the performance of these workloads under $\mathrm{DAV^2M}$ is nearly identical to that we would have seen if we chose the paging approach correctly when the VM was configured ($\leq 1\%$). A second important result is that, for a benchmark whose best paging approach varies over its execution, $\mathrm{DAV^2M}$ is able to dynamically change the paging approach to match changing circumstances, without oscillatory behavior.

A.1 Paging Approaches

In a virtualized environment, paging is complicated because there are essentially two levels of address translation. Conceptually, the guest OS controls the translation from guest virtual addresses (GVAs) to guest physical addresses (GPAs) by manipulating page tables in its address space. The VMM controls the translation from GPAs to host physical address (HPAs) by manipulating some other structure that implements the mapping. The two structural forms we consider here are shadow paging and nested paging.

Shadow paging Shadow paging is a form of virtualized paging that is implemented in software. In order to understand shadow paging, it is helpful to differentiate the privilege level of the guest page tables and the VMM page tables. Because the VMM runs at a higher privilege level, it has the ultimate control over the control registers used to control the normal paging hardware on the machine. Because of this, it can always ensure that the page tables in use contain the correct mapping of guest addresses to host addresses. These page tables, the shadow page tables, contain mappings that integrate the requirements of the guest and the VMM. The shadow page tables implement a mapping from GVA to HPA and are in use whenever the guest is running.

The VMM must maintain the shadow page tables' coherence with the guest's page tables. A common approach to do so is known as the virtual TLB model [4, Chapter 32]. The x86's architected support for native paging requires that the OS (guest OS) explicitly invalidate virtual address (GVAs) from the TLB and other page structure caches when corresponding entries change in the in-memory page tables. These operations (including INVLPG and INVLPGWB instructions, CR3 (the pointer to the current page table) writes, CR4.PGE writes, and others) are intercepted by the VMM and used to update the shadow page tables. The interception of guest paging operations can be expensive as each one requires at least one exit from the guest, an appropriate manipulation of the shadow page table, and one reentry into the guest. These operations are especially expensive when context switches are frequent. A typical exit/entry pair, using hardware virtualization support, requires in excess of 1000 cycles on typical AMD or Intel hardware.

In this work, shadow paging is implemented with caching, as is common in many VMMs. Shadow paging with caching attempts to reuse shadow page tables. Ideally, the reuse is sufficiently high enough that a context switch can be achieved essentially with only one exit, to change the CR3 value. The VMM maintains in memory both shadow page tables corresponding to the current context, and shadow page tables corresponding to other contexts. The distinction is not perfect—it is perfectly legitimate for the guest to share guest page tables among multiple contexts, or even at different levels of the same context. Furthermore, the guest kernel has complete access to all of the guest page tables, for all contexts, at any time.

Nested paging Nested paging is a hardware mechanism that attempts to avoid the overhead of the exit/entry pairs needed to implement shadow paging by making the GVA$\rightarrow$GPA and GPA$\rightarrow$HPA mappings explicit and separating the concerns of their control, making it possible to avoid VMM intervention except when a GPA$\rightarrow$HPA change is desired. Both AMD and Intel support nested paging.

In nested paging, the guest page tables are used in the translation process to reflect the GVA$\rightarrow$GPA mapping, while a second set of page tables, visible only to the VMM, are used to reflect the GPA$\rightarrow$HPA mapping. Both the guest and the VMM have their own copy of the control registers, such as CR3. When a guest tries to reference memory using a GVA and there is a miss in the TLB, the hardware page-walker mechanism performs a two dimensional traversal using the guest and nested page tables to translate the GVA to HPA. When the page walk completes, the result is that the translation is cached in the TLB.

It is important to realize that with nested paging every step of the page walk in the guest's page tables requires a traversal of the nested page tables—the guest and nested page walk lengths do not add, they *multiply*. The consequence is that a TLB miss can be very expensive to handle. However, hardware page walk caching has been extended to ameliorate this, and the VMM can further ameliorate it by using large pages (short walks) in the nested page tables. In this paper, we use the AMD nested paging implementation on the Opteron 2350. Palacios can use large pages for nested page tables.

Comparison A TLB miss under nested paging potentially incurs a very high cost compared to shadow paging because of the two dimensional page walk that is needed. In the worst case, using 4-level page tables in the guest and VMM, the cost for a TLB miss is 24 memory references. In contrast, the cost of a TLB miss for shadow paging is the same as that of the native case. In the extreme where there is little locality of reference, it is likely that nested paging will underperform shadow paging.

On the other hand, a shadow page fault is very expensive due to the necessary involvement of the VMM. This is primarily because the cost of VM exit is quite high, as previously noted. Furthermore, a guest page fault will often produce a pair of exits: the first to inject the page fault into the guest, and the second to fill the shadow page table entry appropriately. This leads to a situation in which a guest that frequently modifies its page tables will perform better using a nested rather than shadow paging approach.

A.2 Workloads

A range of benchmarks are selected based on previously conducted studies on virtualized paging [14, 60]. It is important to note that for many workloads, the paging approach makes little difference to performance. In this work, we focus on specific benchmarks that highlight the differences in performance between the two approaches. These are typically benchmarks that stress the TLB.

The widely-used benchmarks, such as SPEC CPU (2000 and 2006) [3] and PARSEC [26, 24], were surveyed under a native environment, which found TLB-intensive

CPU	Opteron 2350 2 GHz
Cache	L1 DCache (2-way): 64KB L1 ICache (2-way): 64KB L2 (16-way): 512KB L3 (32-way): 2MB
TLB entries (page size)	L1 DTLB (full): 48 (4K, 2M, 1G) L2 DTLB: 512 (4K) 4-way, 128 (2M) 2-way, 16 (1G) 8-way L1 ITLB (full): 32 (4K), 16 (2M) L2 ITLB (4-way): 512 (4K)
BUS	1 GHz
Memory	2GB 667 MHz (DDR2)

Figure A.1: Features of primary test machine.

workloads as shown along the x-axis of Figure A.2. Our measurements were done on a Dell PowerEdge SC1435 described in Figure A.1. The benchmarks ran under Linux 2.6.27 (64-bit). The PARSEC benchmarks used the distributed precompiled binaries. GCC 4.3.2 was used to compile SPEC CPU. Oprofile 0.9.4 is used for measuring TLB misses and clock cycle counts, PinPoints (Pin 2.8/SimPoint 3.2) [1] to profile memory access patterns, and libhugetlbfs 2.4 [53] to examine the effect of large pages.

Large page effects Given this set of TLB-sensitive benchmarks, the effect of the use of large pages is also considered. One expectation is that large pages will reduce the TLB miss rate simply by reducing contention. A second expectation is that because large pages imply fewer levels on the page hierarchy, the number of page table entries are decreased, making it more likely that entries will appear in the data cache. Many of the benchmarks showed increased performance (by 10–20%) using large pages. However, there are also cases, MCF and CACTUSADM, where just the opposite occurs.

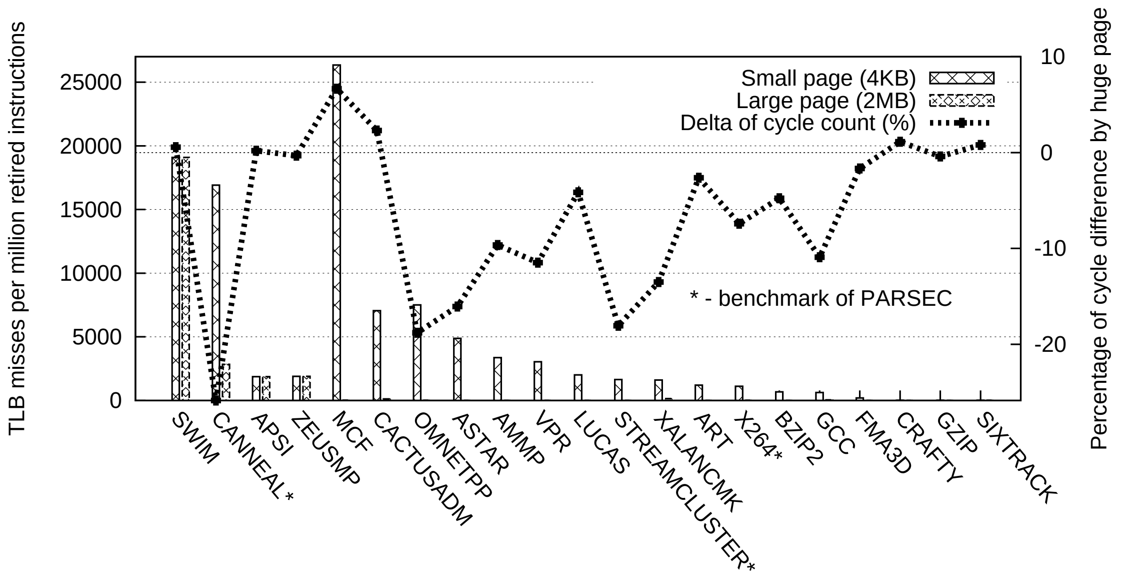


Figure A.2: TLB-intensive workloads surveyed for this work and their native performance with large and small pages. The bars indicate TLB misses (smaller is better), while the line shows the percentage performance difference between large and small pages for each benchmark (smaller is better).

Locality in 2-level page entries To better understand why there are benchmarks where there is a high TLB miss rate even when using large pages, SWIM, VPR, MCF and a "worst case" microbenchmarkf[1] are considered. In Figure A.3, the degree of locality is presented, which holds in the first level ("PDE") of the page hierarchies for these benchmarks. Two benchmarks have more locality than the worst case. However, it is important to note that SWIM shows a very random memory-access pattern with a large working set, which helps

[1]The microbenchmark scans pages in a manner designed to maximize the TLB miss rate by forcing misses at every level of the page hierarchy.

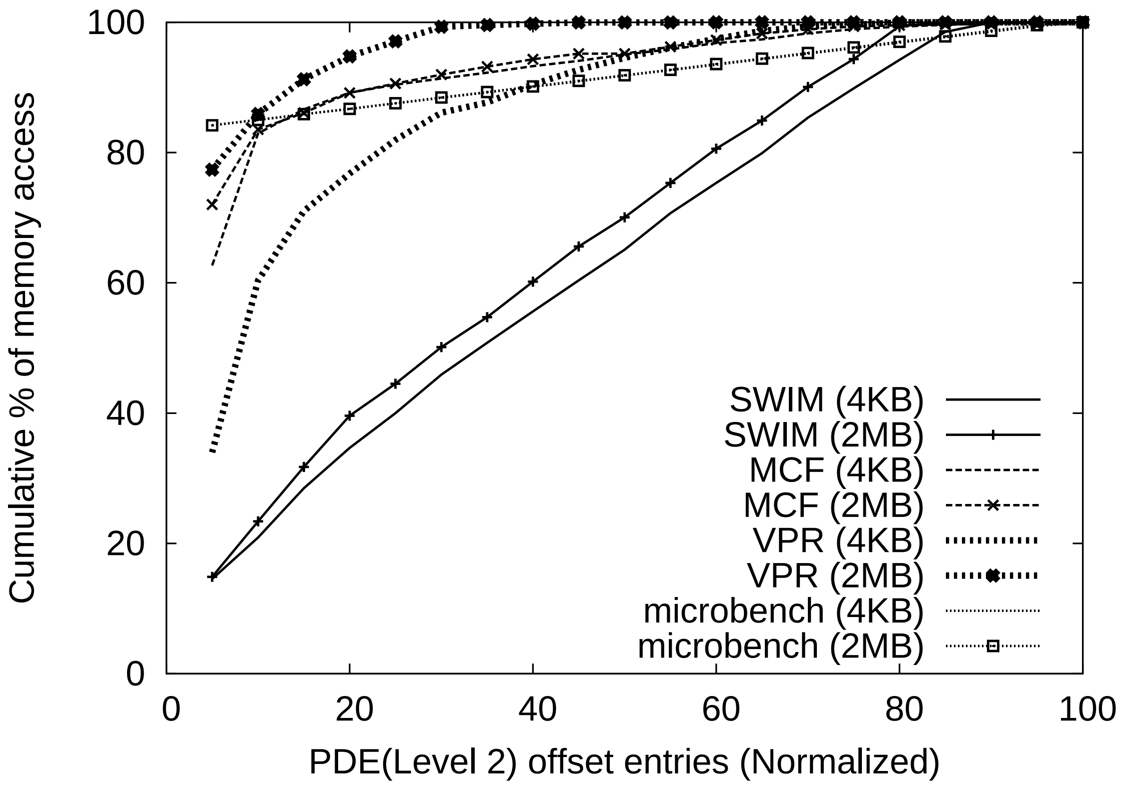


Figure A.3: Locality of reference analysis of benchmarks. The memory-access pattern of SWIM is so random that TLB misses are frequent, even when large pages are used. However, ∼20 % of the page table covers >80% of memory-accesses without large pages (MCF) or with large pages (VPR).

to explain why large pages do not enhance its performance significantly.

A.3 Behavior and Metrics

We now consider the selection of metrics for quickly measuring the performance of shadow and nested paging.

Focused benchmark set Workloads that are TLB-intensive will produce the largest differences between the performance of shadow and nested paging. We therefore have selected the following benchmarks from Figure A.2 for further study: SWIM, APSI, ZEUS-MP, GCC and GZIP. For comparison, FMA3D and CRAFTY are also included, which are much less TLB-intensive. GCC and GZIP use multiple sequential inputs, and thus are likely to show phase behavior in page translation. The different inputs are expected to result in different page mappings. Hence, these benchmarks stress shadow paging, requiring many exits to the VMM to repair the shadow page tables when phase changes occur.

Palacios VMM The shadow and nested paging implementations are implemented in the Palacios VMM. Palacios is is an OS-independent, open source, BSD-licensed, publicly available type-I VMM designed as part of the V3VEE project (`http://v3vee.org`). For this work, Palacios embeds in the Kitten lightweight kernel. Detailed information about Palacios and Kitten can be found elsewhere [78] with code available from the project web site. Specifically these commits are used: Palacios commit – hash: #1cd2958b5eb63b2ac63ced17447ba3b45c43f51a and Kitten commit – hash: #738:02e673-de9a2e. The machine used is as described in the previous section.

Guest OS Benchmarks ran on a guest (and native) OS based on Puppy Linux 3.01 [2], with a 2.6.18 Linux kernel, running on a single core 32 bit guest environment.

Conservative shadow paging performance Although shadow paging with caching is employed, the implementation is likely to produce conservative performance compared to nested paging for several reasons:

- 32 bit guest addressing. 32 bit guest addressing results in the guest page tables being at most 2 levels deep. This means that a nested page walk is much shorter than if 64 bit addressing (4 levels) were used in the guest.

- Small pages. The shadow paging implementation uses small pages, while the nested paging implementation can use large pages at the nested level. As described in Section A.2, using large pages provides an opportunity to reduce TLB contention, thus favoring nested paging. However, it is important to note that benchmarks such as SWIM, APSI, and ZEUSMP are relatively insensitive to the choice of large or small pages.

- TLB tags [9, Chapter 12] and VMCB caching [10, Chapter 15.15] are not used. These features benefit shadow paging more than nested paging as they reduce exit costs.

Performance comparison Figure A.4 shows the performance, compared to native, of nested and shadow paging, for the selected benchmarks. Neither approach is consistently superior.

Metrics The selection of metrics is now considered for measuring application performance and the performance of the current paging approach. Note that application performance may decline for reasons unrelated to paging, so it is essential to measure it independently. The metrics must be very inexpensive to measure. The selected metrics are following:

- Application performance: Cycles per instruction (CPI), measured as the number of CPU cycles needed to execute a window of instructions. The CPI value is smoothened with a 10 step moving average.

- Nested paging performance: TLB miss rate, measured over a window of instructions. The TLB miss rate is smoothened with a 10 step moving average.

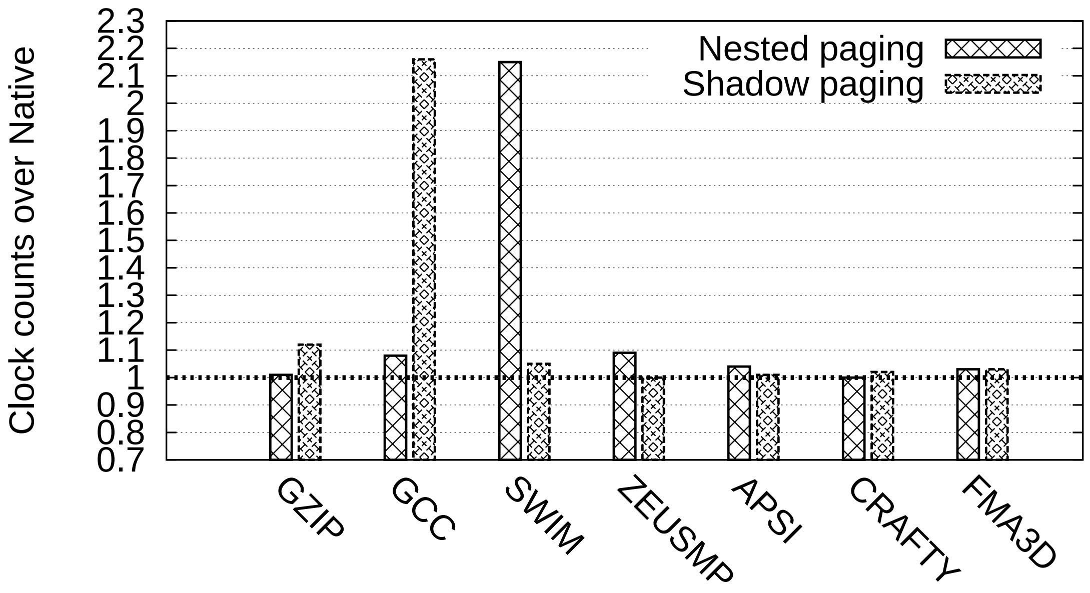


Figure A.4: Performance, compared to native, of selected benchmarks using nested and shadow paging. Lower numbers are better. Neither approach is consistently superior.

- Shadow paging performance: VM exit rate due to paging, measured over a window of instructions.

To capture the CPI and TLB miss rate, the hardware performance counter unit (PMU) [11, Chapter3] is used. The counters used are the cycles outside of halt states, the count of retired instructions, the count of L1 DTLB misses, and the count of L2 DTLB misses. The first two are combined to create the application performance metric (CPI), and the last is used for the nested paging performance metric. VM exits related to paging are counted in Palacios's exit handler. Instructions and TLB misses are counted in the context of the guest, but the cycle count is measured under both VMM and guest context. Because of this, we are measuring these metrics, as they affect the guest, over "wall clock" time (not virtual time). Measurements are made in the context of VM exits that are already occurring, with the window and sampling interval chosen so that there is $<1\%$ overhead.

A.4 Mechanism

The facility to switch between shadow paging and nested paging in the middle of handling any exit is implemented in Palacios. There are essentially two elements to this mechanism, (a) management of the paging and TLB-related aspects of the hardware-specific virtualization extensions, particularly the VMCB [10] or VMCS [4], and (b) management of the paging state for each mechanism in a manner such the guest and VMM intent represented in one can readily be translated to the other.

Palacios maintains a relatively stable GPA$\rightarrow$HPA mapping. Because of this, maintaining nested paging state is quite straightforward. In essence, unless the guest physical memory map changes, the nested page tables can simply be kept cached. Thus, when switching from shadow to nested paging, they can be simply reused. In contrast, the shadow page tables include guest intent, which is not controlled. If guest page table updates are tracked while using nested paging, the performance benefits of nested paging need to be obviated. Instead, the shadow page tables are simply flushed when switching from nested paging to shadow paging. Notice that shadow page caching is still used while shadow paging is used. It is only for the transition from shadow paging to nested paging, and then return to shadow paging in which the shadow paging cache contents is losted. Because of the asymmetry in tracking and reconstructing paging state, it is considerably cheaper to switch from shadow paging to nested paging than the reverse, all other things being the same. Of course, the actual switching cost also depends on the workload.

A.5 Policy

$\mathrm{DAV}^2\mathrm{M}$ is a threshold-based policy. The performance of the current paging mode is compared with a threshold. If the threshold is exceeded, we switch to the alternative mode. A naive approach to setting these thresholds might be to make them fixed. However, as observed, the selected metrics vary widely across workloads, and also across time within an individual workload. Furthermore, even if a threshold were correct for a workload, fixing it would make possible oscillatory behavior, bouncing between the alternative modes.

$\mathrm{DAV}^2\mathrm{M}$ uses dynamic thresholds that are adjusted whenever switching modes. The adjustments to the thresholds are based on the performance difference that is seen, as measured using CPI. If performance increases due to the switch, the threshold is adjusted so that switch will occur at a lower threshold. Otherwise, the threshold makes the switch less probable in the future.

States $\mathrm{DAV}^2\mathrm{M}$ uses five states, as shown in Figure A.5, and described in the following.

- **Shadow** is the state when shadow paging is used. The VM exits related to paging are being counted in $count_{vexit}$. When $window_{vexit}$ instructions are retired, the counter is reset. When it is necessary to leave the state due to $count_{vexit}$ exceeding its threshold, $threshold_{vexit}$, the counter and cpi_{shadow}, computed from the number of retired instructions and the number of cycles during $window_{vexit}$ are stored.

- **PreNested** is the state in which the performance of nested paging is probed, after deciding that shadow paging performance is insufficient. Specifically, the previous cpi_{shadow} is compared with the current cpi_{nested}. Nested paging is being used. PreNested has a limited duration that expires when at least $window_{vexit}$ instructions have been retired.

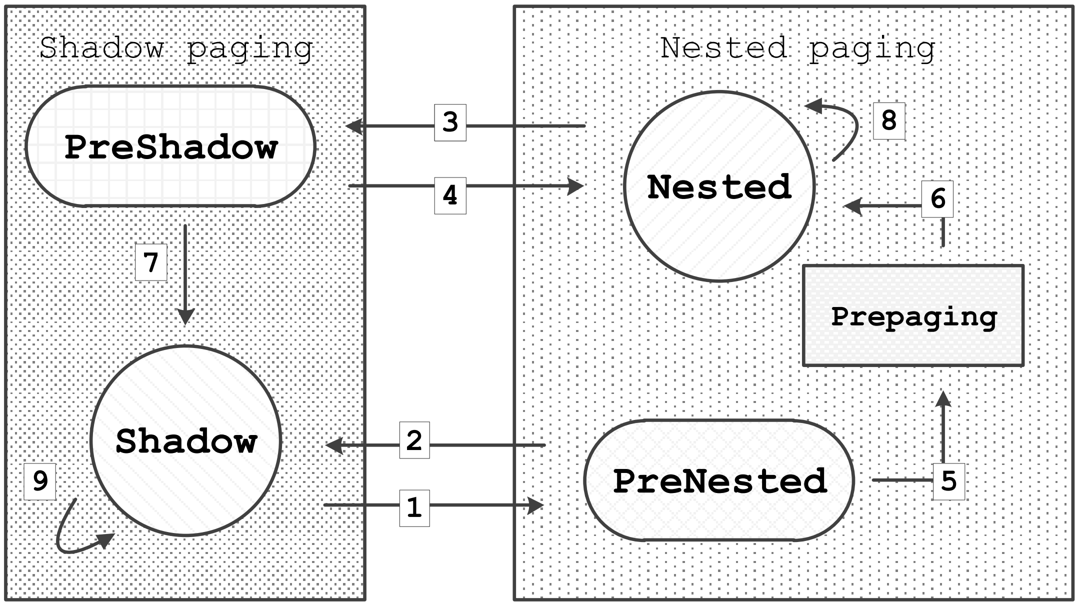


Figure A.5: State transition diagram.

- **Prepaging** is the state that is in force after the PreNested and before Nested. The purpose of this state, during which nested paging is also in force, is to allow the TLB miss rate behavior to quiesce before switching to Nested. Without this state, the high TLB miss rate is easily misinterpretable right after a shadow-to-nested switch and switch back to shadow. The workloads GZIP and GCC are examples where such oscillation would occur.

- **Nested** is the state in which nested paging is used. Here, TLB misses are counted in $count_{tmiss}$, reseting every $window_{tmiss}$ instructions. When this count exceeds $threshold_{tmiss}$, cpi_{nested} is updated and this state is passed.

- **PreShadow** is the state in which the previous cpi_{nested} is compared with the current cpi_{shadow}, while running with shadow paging active. Like PreNested, PreShadow

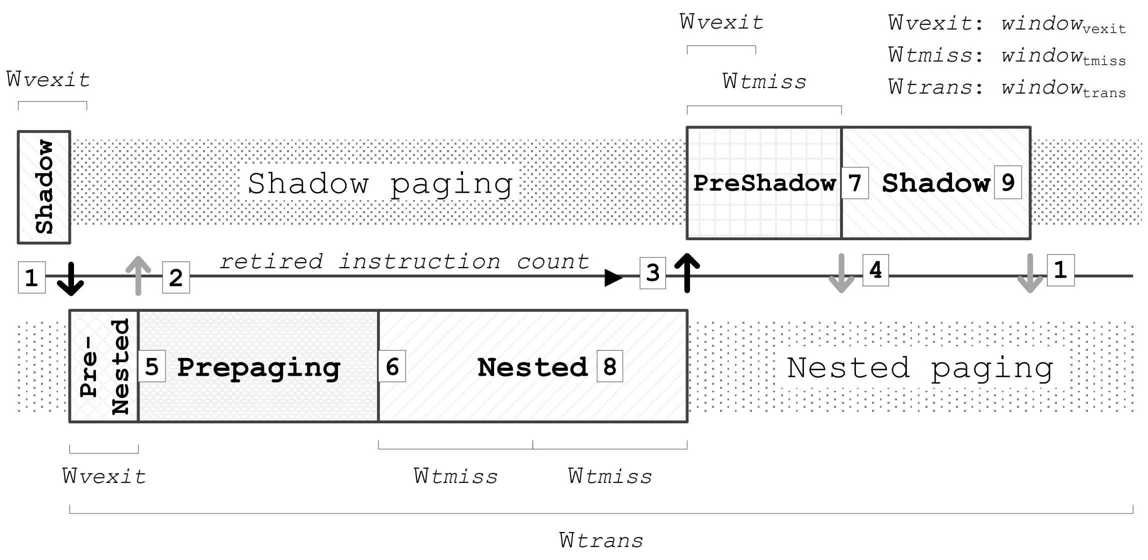


Figure A.6: State transition timeline.

holds for a limited duration. As in Shadow, VM exits are being monitored.

Because the measurement granularity is courser when operating with shadow paging (due to operating over exits), the transient effects of switching from Nested to PreShadow are generally over by the time the PreShadow period is over. Thus no state analogous to PrePaging is needed.

Figure A.6 illustrates an example timing of state transitions. Transitions are labeled by number as in the state diagram.The figure begins in the Shadow state purely for convenience. The time is in retired instructions.

To avoid repeated switching between nested and shadow paging, the thresholds is modified as transitioning from state to state. Furthermore, in testing thresholds, the compared metric ratio is multiplied by a factor, $pFactor$. Finally, the intervals between transitions from nested to shadow ($count_{n2s}$) and shadow to nested ($count_{s2n}$) is considered and which is compared to a window threshold $window_{trans}$. When a transition occurs within this window, both $threshold_{vexit}$ and $threshold_{tmiss}$ are increased, which will damp the system.

Algorithm specifics DAV^{2}M advances through handling the following two events:

```
VM exit for a shadow page fault:
    $count_{vexit} \leftarrow count_{vexit} + 1$
    $determineState_{next}(state_{cur}, state_{next})$
    $transState(state_{cur}, state_{next})$
VM exit for a PMU overflow:
    $count_{inst} \leftarrow 0$
    $count_{punit} \leftarrow count_{punit} + 1$
    $count_{s2n} \leftarrow count_{s2n} + 1$
    $count_{n2s} \leftarrow count_{n2s} + 1$
    $determineState_{next}(state_{cur}, state_{next})$
    $transState(state_{cur}, state_{next})$
```

The PMU is set for $window_{inst}$ instructions as the as sampling period, as Section A.4. For every retired instruction beyond this, an overflow occurs, causing a VM exit. These exits are counted in $count_{punit}$.

$transState(state_{cur}, state_{next})$ transitions to the next state and updates the thresholds. It is implemented as:

```
if $state_{cur} \neq state_{next}$ then
    if transit from PreNested to Shadow or
      from Nested to PreShadow then
        switch paging mode
        if $count_{n2s} < window_{trans}$ then
```

```
            increase $threshold_{tmiss}$ and $threshold_{vexit}$
            $count_{n2s} \leftarrow 0$
        **end if**
    **else if** transit from Shadow to PreNested or
       from PreShadow to Nested **then**
        switch paging mode
        **if** $count_{s2n} < window_{trans}$ **then**
            increase $threshold_{tmiss}$ and $threshold_{vexit}$
            $count_{s2n} \leftarrow 0$
        **end if**
    **end if**
**end if**
```

$determineState_{next}(state_{cur}, state_{next})$ determines the next state. It is implemented as:

```
$state_{next} \leftarrow state_{cur}$
**if** $state_{cur}$ is PreShadow or Shadow **then**
    **if** $count_{vexit} > threshold_{vexit}$ **then**
        update $cpi_{shadow}$
        $state_{next} \leftarrow$ PreNested
        $count_{punit} \leftarrow 0$
    **else if** $state_{cur}$ is PreShadow **then**
        **if** $count_{punit} = window_{tmiss}$ **then**
            update $cpi_{shadow}$
            **if** $cpi_{shadow} > cpi_{nested} * pFactor$ **then**
                $state_{next} \leftarrow$ Nested
            $count_{tmiss} \leftarrow 0$
            $count_{punit} \leftarrow 0$
            increment $threshold_{tmiss}$
          **end if**
        **else if** not $(count_{punit}$ mod $count_{vexit})$ **then**
          $count_{vexit} \leftarrow 0$
        **end if**
      **else if** $state_{cur}$ is Shadow and
        $count_{punit} = window_{vexit}$ **then**
        $count_{vexit} \leftarrow 0$
        $count_{punit} \leftarrow 0$
      **end if**
    **else if** $state_{cur}$ is either PreNested, Prepaging or Nested **then**
      **if** $state_{cur}$ is PreNested and
        $count_{punit} = window_{vexit}$ **then**
        update $cpi_{nested}$
        **if** $cpi_{nested} > cpi_{shadow}$ * $pFactor$ **then**
          increment $threshold_{vexit}$
          $state_{next} \leftarrow$ Shadow
        **else**
          $state_{next} \leftarrow$ Prepaging
        **end if**
      **else if** $state_{cur}$ is Nested and
        $count_{punit} = window_{tmiss}$ **then**
        update $count_{tmiss}$
        if $count_{tmiss} > threshold_{tmiss}$ then
            update $cpi_{nested}$
            $state_{next} \leftarrow$ PreShadow
        else
            $count_{punit} \leftarrow 0$
        end if
    else if $state_{cur}$ is Prepaging and
        $count_{punit} = window_{prepaging}$ then
        $count_{punit} \leftarrow 0$
        $state_{next} \leftarrow$ Nested
    end if
end if
```

Several different windows have been introduced. Their relationships must be

$$window_{inst} \leq window_{vexit} < window_{tmiss} < window_{trans}.$$

A.6 Results

An evaluation of $\mathrm{DAV^2M}$ is now presented, which uses the selected benchmarks described in Section A.3.

Parameters and initial settings The parameters and starting values for $\mathrm{DAV^2M}$, described in Section A.5, were set as follows:

- $pFactor = 1.1$
- $window_{inst} = 10^9$ instructions

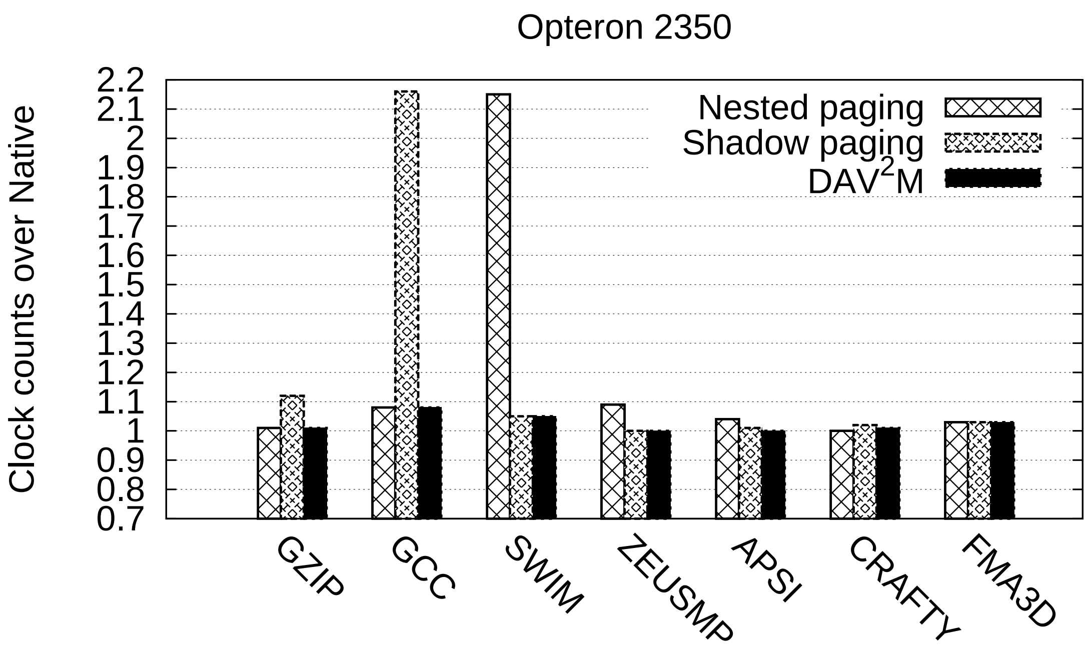


Figure A.7: Performance of $\mathrm{DAV^2M}$˙$\mathrm{DAV^2M}$ is able to provide virtually identical performance to the best static policy for every benchmark.

- $window_{vexit} = 10^9$ instructions
- $window_{tmiss} = 10 \times window_{inst}$
- $window_{trans} = 100 \times window_{inst}$
- $threshold_{vexit} = 10^4$
- $threshold_{tmiss} = 10^5$

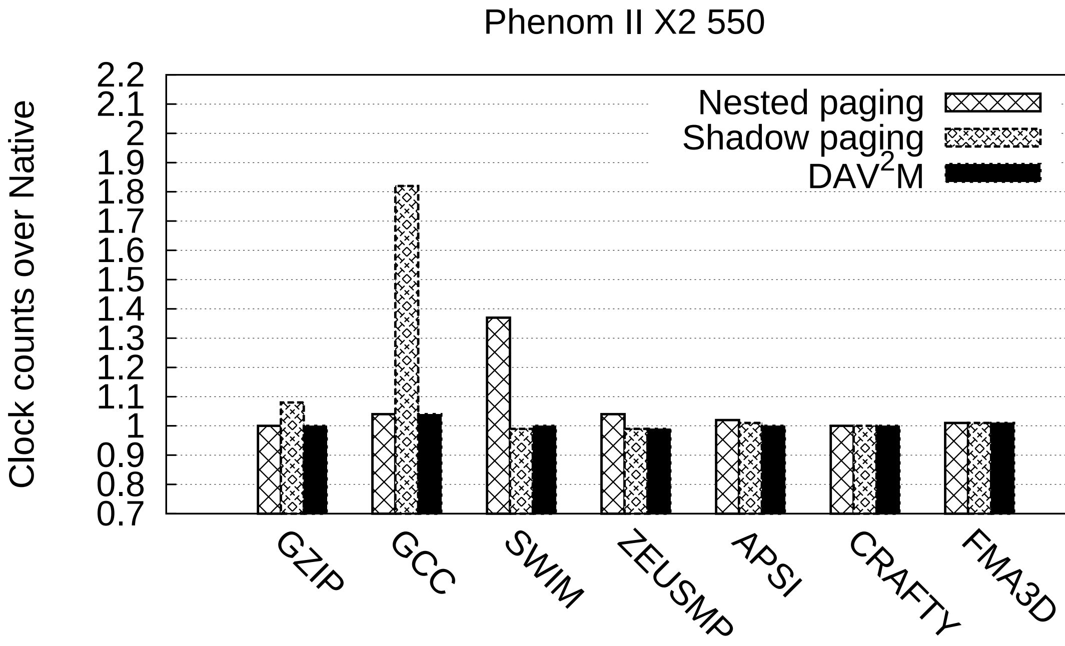


Figure A.8: Performance of DAV^{2}M on the second test machine. DAV^{2}M is also able to produce the same performance of the best static policy.

Application performance Figure A.7 and A.8 present the application performance results comparing DAV^{2}M with the static approaches of either shadow paging or nested paging. The format of each graph is identical to that of Figure A.4. The bars compare to native performance, lower bars are better. The left hand graph presents an evaluation on the same Opteron 2350 as previously described, while the right hand graph presents the same evaluation done on a newer machine. The newer machine is equipped with with an AMD Phenom II X2 550 processor, 4GB RAM, 3GHz clock speed, and 6 MB of L3 cache. The most important observation is that DAV^{2}M is able to provide virtually identical performance to the best static paging approach on all of the benchmarks on both machines.

There are two important things to point out at this point. First, the measurements given in Figure A.7 (and Figure A.4) are of the number of cycles needed to run the benchmark—they reflect the total execution times of the benchmarks. These should not be confused with the CPI metric (Section A.3) that $\mathrm{DAV^2M}$ uses internally to heuristically determine application execution rate. Secondly, recall that our evaluation focused on a set of benchmarks that induced the most significant differences between the two paging approaches (Section A.2). For benchmarks where there is little difference (two are included), $\mathrm{DAV^2M}$ correctly does not affect performance.

Deeper analysis Now focusing on the results for the Opteron 2350 machine, it is illustrated on how $\mathrm{DAV^2M}$ is working, and what its overheads are.

It is possible to group the benchmarks into three sets based on which virtual paging mode is best for performance:

1. SWIM and ZEUSMP are best under shadow paging.

2. GCC and GZIP are best under nested paging.

3. APSI, CRAFTY and FMA3D perform similarly.

For (1) and (2), $\mathrm{DAV^2M}$ quickly chooses the right approach. For (3), $\mathrm{DAV^2M}$ quickly chooses *an* approach and avoids switching.

Figure A.9 illustrates the number of switches of paging mode that occur for each benchmark. Even in GCC, only 13 transitions occur. GCC has phase behavior in which short phases where shadow paging is preferable occur. Note that despite this switching, GCC under $\mathrm{DAV^2M}$ performs as well as the best static policy: the costs of switching are counterbalanced by the increased performance in those phases. For the other benchmarks, very little switching is observed, as expected.

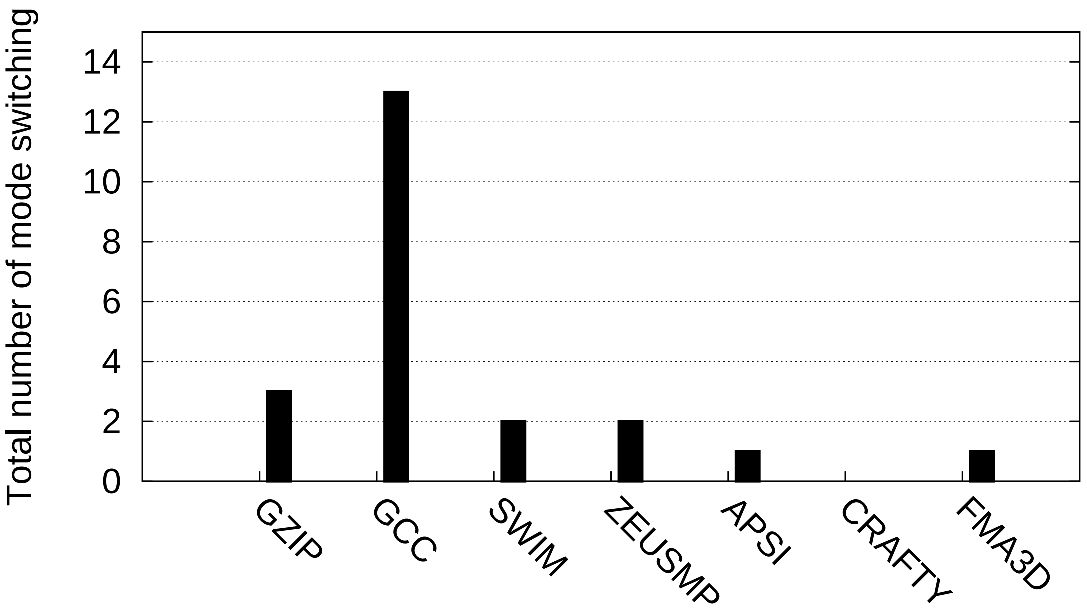


Figure A.9: Number of transitions seen during execution.

Figure A.10 shows the percentage of time that the benchmarks spend in each mode when run under $\mathrm{DAV^2M}$. Here the three sets of benchmarks are quite evident. For the third set, in which the paging approach does not matter, we see that $\mathrm{DAV^2M}$ has made opposite decisions (SWIM vs. ZEUSMP), but with no real consequences for performance. At most one switch occurred.

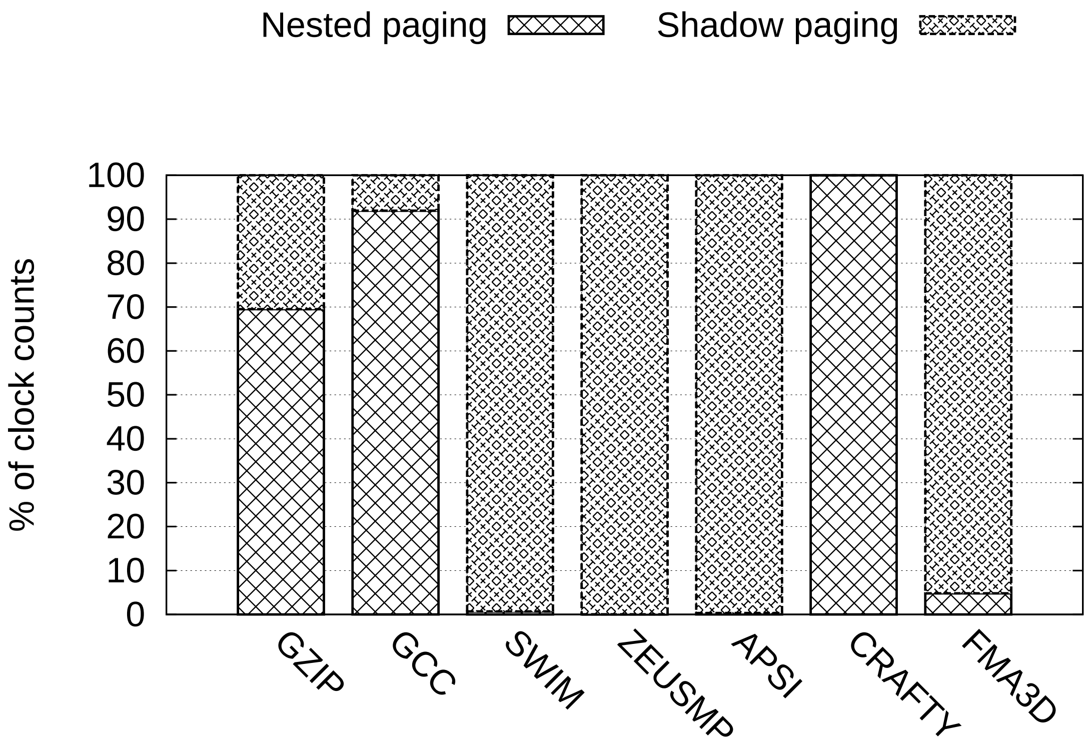


Figure A.10: Percentage of time spent in each mode under $\mathrm{DAV^2M}$.

A.7 Related Work

The virtualization of paging has a history as long as virtual machine monitors themselves. For the x86 platform, the initial descriptions of the software-based approaches of shadow paging in VMware [112] and paravirtualized shadow paging in Xen [18] set the stage. Bhargava et al [23] give a detailed treatment of the hardware-based approach of nested paging and its optimization. Barr et al [19] propose new approaches to page walk caching that could be used to further accelerate nested paging. Adams and Ageson [6] compare hardware and software techniques in x86 virtualization, while Karger [69] compares x86 and DEC Alpha virtualization, a comparison that includes a excellent treatment of the aspects of x86 paging that make it particularly challenging to virtualize with high performance.

Wang et al [113] also propose, implement, and evaluate an adaptive approach to paging approach selection. $\mathrm{DAV^2M}$ uses a different mechanism and policy, and is evaluated in the context of a different VMM. The work in this Appendix was previously described at ICAC 2010 [16]. Both papers find adaptive paging to be highly promising.

A.8 Conclusions

In this work, a case is made, in which no single approach to virtualizing virtual memory is best for maximizing application performance, focusing on the choice between shadow paging and nested paging. Rather, the choice is workload-dependent, and it may even vary over the life of a virtual machine. In response, a mechanism is created in our Palacios VMM for changing the paging approach at any time, and a policy, $\mathrm{DAV^2M}$, for driving that mechanism to increase application performance. It is demonstrated that $\mathrm{DAV^2M}$ is able to adapt to workload, providing performance at least as good as the best statically chosen paging approach for the workload. Although the implementation is available in the Palacios VMM, the general idea could be applied in any VMM that supports multiple paging approaches.