Title

Dynamic workload characterization for energy efficient computing

Permalink

https://escholarship.org/uc/item/69x5h7ps

Author

Dhiman, Gaurav

Publication Date

2011

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA, SAN DIEGO

Dynamic Workload Characterization for Energy Efficient Computing

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Gaurav Dhiman

Committee in charge:

      Professor Tajana Rosing, Chair
      Professor Rajesh Gupta
      Professor Tara Javidi
      Professor Dean Tullsen
      Professor Amin Vahdat

2011

The dissertation of Gaurav Dhiman is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____

_____
Chair

University of California, San Diego

2011

DEDICATION

To my parents.

EPIGRAPH

karmany evadhikaras te
ma phalesu kadachana
ma karma-phala-hetur bhur
ma te sango stv akarmani.
–Lord Krishna
(Bhagavad Gita)

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I am indebted to my parents for their unconditional love and selfless support – they have been my pillars of strength. Their unfaltering trust and absolute confidence in my abilities always kept me inspired, focused and optimistic. I thank my younger brother Saurabh for being a perfect buddy. It has been wonderful to see him grow into a mature and responsible person he has become. His relaxed and calm demeanor under all circumstances has amazed and inspired me in equal measure. I thank my wife Kirti for her unflinching support, love and encouragement, which made the tough and relentless phases of my PhD seem a lot easier. I feel extremely privileged to have such a wonderful and loving family.

ics and Design, 2009. Dhiman, G.; Marchetti, G. and Rosing, T.S. The dissertation author was the primary investigator and author of this paper.

Chapters 1 and 5, in part, are reprints of the material as it appears in ACM Transactions on Design Automation of Electronic Systems, 2010. Dhiman, G.; Marchetti, G. and Rosing, T.S. The dissertation author was the primary investigator and author of this paper.

Chapters 1 and 6, in part, are reprints of the material under submission at International Conference for High Performance Computing, Networking, Storage and Analysis, 2011. Dhiman, G.; Kontorinis, V.; Ayoub, R.; Sadler, C.; Tullsen, D. and Rosing, T.S. The dissertation author is the primary investigator and author of this paper.

2002	B. Tech. in Computer Science and Engineering, Indian Institute of Technology, Roorkee, INDIA
2007	M. S. in Computer Science, University of California, San Diego, La Jolla, CA
2011	Ph. D. in Computer Science, University of California, San Diego, La Jolla, CA

PUBLICATIONS

Dhiman, G.; Marchetti, G. and Rosing, T.S., vGreen: A System for Energy Efficient Management of Virtual Machines. In ACM Transactions on Design Automation of Electronic Systems, November 2010.

Dhiman, G.; Kontorinis, V.; Tullsen, D.; Rosing, T.S.; Saxe, E. and Chew, J., Dynamic Workload Characterization for Power Efficient Computing in CMP Systems. In Proceedings of the 15th ACM/IEEE International Symposium on Low Power Electronics and Design, 2010.

Dhiman, G.; Mihic, K. and Rosing, T.S., A System for Online Power Prediction in Virtualized Environments Using Gaussian Mixture Models. In Proceedings of the 47th ACM/IEEE Design Automation Conference, 2010.

Dhiman, G. and Rosing, T.S., System Level Power Management Using Online Learning. In IEEE Transactions in Computer Aided Design of Integrated Circuits and Systems, May 2009.

Dhiman, G.; Marchetti, G. and Rosing, T.S., vGreen: A System for Energy Efficient Computing in Virtualized Environments. In Proceedings of the 14th ACM/IEEE International Symposium on Low Power Electronics and Design, 2009.

Dhiman, G.; Ayoub, R. and Rosing, T.S., PDRAM: A Hybrid PRAM and DRAM Main Memory System. In Proceedings of the 46th ACM/IEEE Design Automation Conference, 2009.

Dhiman, G., Pusukuri, K.K. and Rosing, T. S., Analysis of Dynamic Voltage Scaling for System Level Energy Management. In Proceedings of the Workshop on Power Aware Computing and Systems, 2008.

Dhiman, G. and Rosing, T. S., Dynamic voltage frequency scaling for multi-tasking systems using online learning. In Proceedings of the 12th ACM/IEEE International Symposium on Low Power Electronics and Design, 2007.

Dhiman, G. and Rosing, T. S., Dynamic power management using machine learning. In Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, 2006.

ABSTRACT OF THE DISSERTATION

Dynamic Workload Characterization for Energy Efficient Computing

by

Gaurav Dhiman

Doctor of Philosophy in Computer Science

University of California, San Diego, 2011

Professor Tajana Rosing, Chair

Energy management has become a key issue in the design of computing systems in both mobile as well as enterprise space. For mobile systems it is important from the perspective of battery life, while for large scale systems it directly impacts operational costs, making dynamic energy management methods a critical system component. An effective and conventional mechanism to achieve runtime energy savings is to dynamically reduce the power consumption of the system by transitioning to sleep. This thesis introduces a novel online learning based meta-policy for dynamic power management, where based on characteristics of the device and workload, the system converges to the best suited power management policy for it. We show, that this approach provides superior adaptability across diverse work-

loads compared to state of the art single policy solutions achieving energy savings of up to 61%.

The thesis next provides analysis of effectiveness of power management based approach for energy management on mobile and server class systems. The study reveals that while the effectiveness is high for mobile systems, it is on a decline for server systems due to their *non energy proportional* design. This motivates energy management through two alternative means – energy proportional design and workload consolidation.

For energy proportional design, the thesis focuses on memory subsystem as an example and proposes a memory hierarchy using phase change memory (PRAM) and conventional DRAM. The proposed system exploits the characteristics of workloads (in terms of their read and write intensity) to intelligently place pages across both the memories, thus reducing overall energy consumption of the memory subsystem by 40% on average compared to conventional DRAM. For workload consolidation, the thesis devises cluster level virtual machine (VM) scheduling and resource management algorithms. We show that fine grained information on how the VMs utilize share the system resources like the CPU pipeline, memory bandwidth etc. can be exploited to perform more intelligent VM consolidation and resource management to maximize the overall energy efficiency. Real-life implementation of such VM management policies out-perform state of the art techniques that primarily use just CPU utilization for VM management by 35% on average.

Chapter 1

Introduction

Energy management has become a key issue in the design of computing systems today. On one end, the increasing popularity of small scale battery driven portable systems necessitates a design that offers longer battery life without compromising the performance of increasingly complex applications (eg. multimedia, web etc.). On the other end, for large scale systems that populate modern data center and enterprise environments, energy efficient design is key to reduction of all energy-related costs, including capital, operating expenses, and environmental impacts. These reasons have made energy efficient design an active area of research for both small and large scale systems.

Modern system designers and architects have designed system components with efficient support for energy management. For instance, modern CPUs and hard drives support sleep states, which consume dramatically lower power than active states. Modern multi-core architectures provide much higher performance per watt through inherent hardware parallelism. Technologies like virtualization enable higher resource utilization through consolidation, which reduces operational costs and increases the energy efficiency of individual systems.

Effectively utilizing these architectures and designs is key to achieving runtime energy efficiency. This thesis shows that in order to do so, it is extremely important to understand and exploit the characteristics of the workloads running on the system. The characteristics here refer to the way the workloads utilize different resources and components of the system such as CPU, caches, memory

etc. Most of the existing state of the art dynamic energy management techniques currently treat CPU utilization as the primary workload characteristic to drive their management decisions, which as we show in this thesis, is sub-optimal. More fine grained information on how intensively workloads use the whole architecture hierarchy – CPU pipeline, shared caches, memory bandwidth etc. can provide opportunities for aggressive energy management even at same CPU utilization levels. This idea forms the over-arching theme of this thesis – developing mechanisms and policies to perform dynamic runtime energy management of the system by leveraging the way workloads use the system resources.

1.1 Dynamic Energy Management

A workload that uses a given system component can be represented by a two state finite state machine in terms of how it uses the component : busy and idle. Busy state corresponds to the times during which the workload uses the component to actively perform some processing. For instance, when an application thread in running on the CPU or the hard disk is spinning to serve a block request. Conversely, the idle state corresponds to the instances when the workload is not generating any requests for the component, as a consequence of which, it is inactive or not being utilized. Energy consumption for any workload on an operational system is the product of the power consumption of the system components and the runtime of the workload:

$$E = \int_{t_0}^{t_1} P(t) \ dt \tag{1.1}$$

Based on this equation, an intuitive way of dynamically achieving energy savings is to reduce the power consumption of the system with minimal impact on the execution time of the workload. This will result in reduction in energy consumption proportional to the decrease in power consumption. This approach towards energy management is referred to as 'active power management', since it is based on actively managing the power consumption of the system to achieve

energy savings. The power consumption can be managed during both the busy as well as the idle states of the workload. Based on the state of the workloads (and hence the component), i.e. busy or idle, during which power management is performed, the active power management techniques can be divided into two categories:

- *Dynamic Power Management (DPM):* When the workload is in the idle state, the system component is inactive and its ability to actively execute workloads or serve user requests is not required. Consequently, modern system components like CPUs, hard drives, network cards etc. support sleep states, which consume lower power but compromise the ability of the component to actively serve workload requests. Dynamically utilizing such sleep states during the idle state to achieve energy savings is referred to as Dynamic Power Management or DPM.

- *Dynamic Voltage Frequency Scaling (DVFS):* In addition to the sleep states, modern CPUs also support low power states (in terms of lower voltage frequency settings) that can be used when it is actively executing the workload. This form of active power management is referred to as Dynamic Voltage Frequency Scaling or DVFS.

The goal of both the active power management techniques – DPM and DVFS, is to maximize the reduction in power consumption with minimal impact on performance, so that the energy consumption (based on equation 1.1) can be minimized. The following discussion will provide details on the existing state of the art approaches for active power management and their applicability and limitations in terms of their effectiveness for energy management.

1.1.1 Dynamic Power Management

Dynamic Power Management (DPM) refers to the mechanism of dynamically exploiting the sleep or low power states of system components during the idle states to reduce their power consumption. The control procedure that performs

the DPM decision is referred to as the DPM policy. DPM has been an active area of research and several system level DPM policies have been proposed in the past. The existing DPM policies can be broadly classified into timeout, predictive and stochastic policies. In a timeout policy, the device is put to sleep if it is idle for more than a specified timeout period [59, 37]. For instance, in [59], the device is put to sleep if it is idle for more than T_{be} (break-even time). T_{be} of a low power state is the minimum length of the idle period, which compensates for the cost associated with entering it. In contrast, predictive policies [92, 53, 21] predict the duration of upcoming idle period and make the shutdown decision as soon as the device goes idle. Such heuristic policies tend to be easy to implement, but do not offer any guarantee on energy and performance delay, since they do not model the statistical properties of the workload.

Stochastic policies model the workload and device power state changes as stochastic processes. Minimizing power consumption and performance delay then becomes a stochastic optimization problem. For instance, in [80], Paleologo et al assume the arrival of requests as a geometric distribution and model power management as a Discrete-Time Markov Decision Process (DTMDP). This model is subsequently improved in [84, 90] using more complex MDP models (for instance the Time Indexed Semi Markov Decision Process or TISMDP model in [90]) to characterize real life workloads. Stochastic policies offer optimality only for stationary workloads. The work in [20] and [88] extends the stochastic model to handle non stationary workloads by switching between a set of pre-calculated optimal stochastic policies. However, these approaches compromise optimality by switching heuristically and are also quite complex to implement.

1.1.2   Dynamic Voltage Frequency Scaling

For components like CPU, power consumption can be also managed while actively executing workload by reducing its operational voltage and frequency setting (v-f setting). This technique is known as Dynamic Voltage Frequency Scaling (DVFS). To see why this is beneficial, the following equation shows the breakdown of power consumption of a typical CPU:

$$P = D + L, \; D = C_L V^2 f, \; L = I_L V \qquad (1.2)$$

where C_L is the load capacitance, V is the supply voltage, I_L is the leakage current, and f is the operational frequency. The first term corresponds to the dynamic power consumption component of the total power consumption (referred to as D), while the second term corresponds to the leakage power consumption (referred to as L). We can clearly see that the dynamic power has a cubic dependence on the product of voltage and frequency while the leakage power a linear dependence on voltage. Thus, the dynamic component of power consumption clearly benefits (gets reduced) more than the leakage component through reduction in v-f setting. This implies that DVFS is more effective for energy savings on CPUs with lower leakage power component ($\frac{L}{P}$), which makes CPU leakage a very important characteristic to be taken into account while designing DVFS policies.

The existing DVFS techniques may be broadly divided into three categories. The first category of techniques target systems, where the task arrival times, workload and deadlines are known in advance [85, 106]. DVFS is performed at task level in order to reduce energy consumption while meeting hard timing constraints. The second category of techniques require either application or compiler support for performing DVFS [7, 104, 22]. The third category comprises of system level DVFS techniques that target general purpose systems that have no hard task deadlines, and expect no support from application/compiler level. The work done in [101, 38, 96] monitor the system workload in terms of CPU utilization at regular intervals and perform DVFS based on their estimate of CPU utilization for the next interval. These approaches, however, do not take the characteristics of the running tasks into account, which as we show in section 2.3.2, determine the potential benefits of performing DVFS. In contrast, the work done in [102, 19, 56] characterize the running tasks at runtime, and accordingly make the voltage scaling decisions. They use dynamic runtime statistics such as cache hit/miss ratio, memory access counts etc. obtained from the hardware performance counters to perform task characterization. However, the policy in [102] is not flexible since it operates for a static performance loss (10%), while [19] and [56] present results only

in a single task environment. Besides, none of these techniques give strategies on how to adapt DVFS policies to different CPU leakage characteristics, which as we discussed above is extremely important for energy savings. The prior algorithms that take leakage into account [58, 24], are targeted towards real time systems, and assume precise knowledge of task deadlines and characteristics in advance.

The existing work on DPM and DVFS policies falls short on three major fronts: (1) Most of the DPM policies are not designed to adapt to diverse device and workload characteristics. As we show in this thesis, some policies can perform very well for certain workloads, but can be easily outperformed by other policies across diverse workloads. (2) Most of the DVFS policies do not take into account both the leakage and workload characteristics for general purpose systems, which, as this thesis shows, is key to performing effective DVFS across diverse workloads. More fine grained information on how intensively the workload uses the CPU pipeline (as opposed to just CPU utilization) and CPU leakage provides additional opportunities for DVFS policies to achieve higher energy savings. (3) None of the DVFS policies are designed to take into account the impact of their decisions on possible energy savings due to DPM. This is extremely important since both of them can be used to save energy for CPUs, and it is critical to devise an approach that can symbiotically perform DVFS and DPM during busy and idle times to save energy.

These observations and limitations serve as our motivation for developing novel DPM and DVFS policies taking into account the workload characteristics to address these concerns.

1.1.3   Applicability of Active Power Management

Active power management is an effective approach to achieve energy savings only in systems where majority of power consumed is contributed by components that are *'energy proportional'* [42]. Energy proportionality refers to the property of components where their power consumption is proportional to their utilization. This property is extremely important for active power management since it relies on moving components into lower power states when their utilization is low.

For instance, CPUs are highly energy proportional since their power consumption can be reduced in proportion to their utilization through DPM and DVFS states commonly supported by commercial vendors like Intel and AMD. In contrast, components like fans, power supply and memory are not energy proportional since their utilization and power consumption curve is non linear.

The usage pattern of mobile systems is bimodal in nature, high levels of activity interspersed with long periods of inactivity [90, 42]. The majority of power consumption for these systems comes from devices like CPU, screen and radio interface, which can be easily turned off during idle periods. These reasons make mobile systems attractive candidates for aggressive active power management.

However, the same is not true for the server class systems that populate modern data centers. The usage model for servers has very different characteristics. Figure 1.1a shows the distribution of CPU utilization levels for thousands of servers during a six-month interval at Google [42]. The key observation that can be made from the figure is that servers are rarely completely idle and seldom operate near their maximum utilization. Instead, servers operate most of the time at between 10 and 50% of their maximum utilization levels. As they have few windows of complete idleness, it implies that the servers cannot take advantage of system level DPM policies that mobile devices otherwise find so effective for dynamic energy management. Furthermore, we show in section 3 that the effectiveness of DVFS for energy management on server systems has significantly diminished.

In terms of power consumption, close to 50% power consumed by such systems is contributed by non energy proportional components like fans, power supplies, memory etc. [42]. Figure 1.1b plots the power consumption and the energy efficiency ($\frac{utilization}{power}$) for a server with active power management enabled. We can clearly see that even an energy efficient server still consumes about half its full power when doing virtually no work. Moreover, the peak energy efficiency occurs at peak utilization and drops quickly as utilization decreases. Notably, energy efficiency in the 20 to 30% utilization range – the point at which servers spend most of their time – has dropped to less than half the energy efficiency at peak performance. Thus, the applicability of active power management for energy
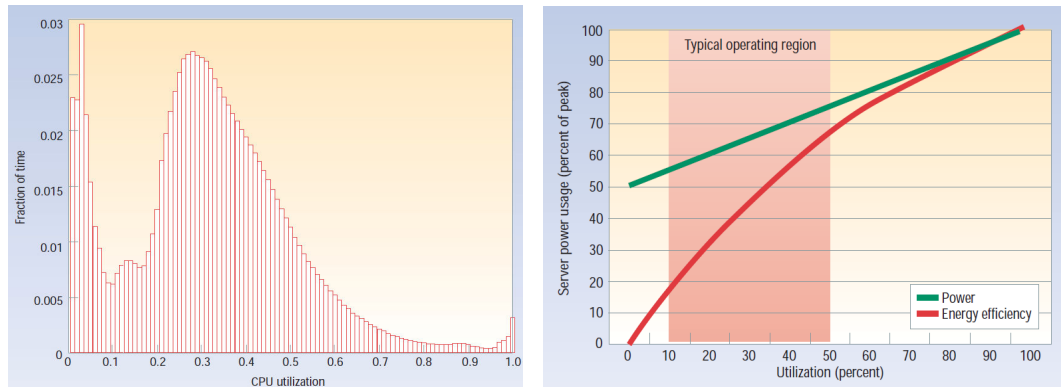
(a) Average CPU utilization [42]

(b) Server power usage and energy efficiency at varying utilization levels [42]

Figure 1.1: Average CPU utilization, power and energy efficiency of 5000 servers during six-month period.

management on such systems is pretty low.

1.1.4   Energy Efficiency for Server Systems

For server class systems which, as shown in the previous discussion are not energy proportional, there are two ways of achieving energy efficiency: (1) Design systems to be more energy proportional. (2) Push the existing systems towards more energy efficient zone of operation by increasing their utilization (see Figure 1.1b).

The first approach requires changes at hardware as well as software level. For instance, new non-volatile memory technologies like phase change memory, flash memory etc., that are more energy efficient than conventional DRAM can be used. This may require modifications to both the hardware and software stack of the system to incorporate them. We refer to this approach towards energy management as *'energy proportional design'*.

The second approach is to leverage technologies like virtualization that enable consolidation of workloads on fewer physical machines to increase the overall utilization of the system. This is beneficial from three angles: (1) It can generate idle systems, which may be either turned off or be used for doing additional work

thereby increasing the overall energy efficiency; (2) The consolidated machines run at higher utilization, which is inherently more energy efficient (see Figure 1.1b); (3) The energy efficiency of the consolidated machines can be enhanced by optimizing the runtime or throughput of the workloads (refer equation 1.1) under the same power budget. We refer to this mechanism for energy efficiency as '*workload consolidation*'.

Role of Workload Characteristics: Due to the flexibility and benefits offered by the virtualization and consolidation approach, it has achieved a lot of traction in recent years in both the open source community (Xen, KVM etc.) as well as industry (VMware, Hyper-V etc.). Policies for power aware virtual machine (VM) consolidation and management using these solutions have been proposed in previous research [86, 74, 103] and are available as commercial products as well (eg. VMware DRS [99]). These policies require understanding of the power consumption and resource utilization of the physical machine, as well as its breakdown among the constituent VMs for optimal decision making. Currently, they treat the overall CPU utilization as the primary workload characteristic of VMs to estimate their respective power consumption and resource utilization, and use it for guiding the VM management policy decisions (VM migration, DVFS etc.). However, this thesis shows that based on more fine grained characteristics of the workloads in these different co-located VMs (like the CPU pipeline, shared cache and memory bandwidth usage), the overall power consumption, resource utilization and performance of the VMs can vary a lot even at similar CPU utilization levels, which can mislead the VM management policies into making decisions that create hotspots of activity, violate performance requirements of diverse workloads and degrade the overall energy efficiency.

1.2 Thesis Contributions

This thesis presents research for energy management using all the techniques described above: active power management, energy proportional design and workload consolidation. The following discussion highlights the primary contributions

and the outline of the rest of the thesis:

- It presents an online learning based approach towards active power management (both DPM and DVFS). We apply an online learning algorithm [34] to select among a set of possible policies and v-f settings instead of developing a new policy. The algorithm (referred to as *controller*) has a set of *experts* (DPM policies/v-f settings) to choose from and selects an expert that has the best chance to perform well based on the controller's characterization of the current workload. The selection takes into account energy savings, performance delay as well as the user specified energy-performance tradeoff (referred to as e/p tradeoff). The algorithm is guaranteed to converge to best performing expert in the set, thus delivering performance atleast as good as the best expert in the set, across different workloads. Evaluation across devices with varying characteristics and workloads shows that the controller is able to achieve energy savings of up to 61%. The controller is described in Chapter 2.

- It shows that the applicability of active power management techniques for server class systems has severely diminished for achieving energy savings, and identifies the reasons for such a trend (non energy proportionality being one of them, as identified in the discussion above). Memory hierarchy is then used an example to illustrate the potential of energy savings through a redesign in both hardware and software stack to achieve much better energy proportionality and system level energy efficiency. The proposed novel memory hierarchy comprises of both conventional DRAM and phase change memory (PRAM), and the system exploits the characteristics of the workloads (in terms of memory reads and writes) to optimize page allocation across both the memories to achieve energy savings of up to 40%. This discussion is included in Chapters 3 and 4.

- It introduces vGreen, a multi-tiered software system to manage virtual machine scheduling and consolidation across a cluster of physical machines with the objective of maximizing the overall energy efficiency and performance

in terms of instruction throughput. The basic premise behind vGreen is to understand and exploit the relationship between the architectural characteristics of the VM workload (eg. instructions per cycle, memory accesses etc.) and its performance and power consumption, which has been not taken into account by the previous state of the art VM management systems. vGreen is implemented and validated on a real life testbed of server systems, and improves the overall performance up to 100% and energy efficiency up to 55% compared to state of the art VM scheduling and power management policies at negligible runtime overhead. The details of vGreen are presented in Chapter 5.

- The vGreen system is extended to include resource management for managing diverse workloads (both latency sensitive and batch applications) in the data center. The enhanced system (referred to as Themis) is responsible for managing the performance and QoS of these diverse workloads while maximizing the overall energy efficiency by facilitating aggressive workload consolidation. The key idea behind Themis is to exploit the inherent heterogeneity in the characteristics of the latency sensitive services and batch applications in the way they use system resources, which allows it to maximize energy-efficient throughput of the latter without sacrificing the service guarantees of the former. Themis implements a resource management policy that outperforms ideal implementations of prior state of the art policies by up to 35% on average in terms of energy efficiency. We present the details of Themis in Chapter 6.

Chapter 1, in part, is a reprint of the material as it appears in IEEE Transactions in Computer Aided Design of Integrated Circuits and Systems, 2009. Dhiman, G. and Rosing, T.S. The dissertation author was the primary investigator and author of this paper.

Chapter 1, in part, is a reprint of the material as it appears in Proceedings of the Workshop on Power Aware Computing and Systems, 2008. Dhiman, G.; Pusukuri, K.K. and Rosing, T. S. The dissertation author was the primary investigator and author of this paper.

Chapter 1, in part, is a reprint of the material as it appears in Proceedings of the 46th ACM/IEEE Design Automation Conference, 2009. Dhiman, G.; Ayoub, R. and Rosing, T.S. . The dissertation author was the primary investigator and author of this paper.

Chapter 1, in part, is a reprint of the material as it appears in ACM Transactions on Design Automation of Electronic Systems, 2010. Dhiman, G.; Marchetti, G. and Rosing, T.S. . The dissertation author was the primary investigator and author of this paper.

Chapter 1, in part, is a reprint of the material under submission at International Conference for High Performance Computing, Networking, Storage and Analysis, 2011. Dhiman, G.; Kontorinis, V.; Ayoub, R.; Sadler, C.; Tullsen, D. and Rosing, T.S. . The dissertation author is the primary investigator and author of this paper.

Chapter 2

Active Power Management

2.1   Introduction

Dynamic Power Management (DPM) and Dynamic Voltage Frequency Scaling (DVFS) are the two most popular techniques for dynamically reducing system power dissipation. DPM achieves this by selective shutdown of system components that are idle, while the key idea behind DVFS techniques is to dynamically scale the supply voltage/frequency level of the device. Reduction in voltage/frequency level is beneficial, since it reduces the overall power consumption [17]. While DPM can be employed for any system component with multiple power states, while DVFS is useful only for components that support multiple speed and voltage levels (like CPU). A number of modern processors such as Intel XScale [36], AMD Opteron [1] etc. are equipped with DVFS capability. In existing literature however, the design of DPM and DVFS policies for general purpose systems has been treated as separate problems. In this thesis, we target both the problems with the objective of achieving system wide energy efficiency.

A number of heuristic and stochastic policies have been proposed in the past with their design varying in terms of how they take the decision to perform shutdown for DPM. While simpler DPM policies like timeout and predictive policies do it heuristically with no performance guarantees, more sophisticated stochastic policies guarantee optimality for stationary workloads. There is no single policy solution which guarantees optimality under varying workload conditions. We pro-

pose a novel setup for DPM, where we maintain a set of DPM policies (suited for different workloads) and design a control algorithm that selects the best suited one for the current workload. In such a setup, the DPM problem reduces to one of accurate *characterization* and *selection*, where the best suited policy is selected based on the characterization of the current workload.

For devices like CPU that support both DPM and DVFS, it is essential to understand the interplay between the two, since the energy savings based on DVFS come at the cost of increased execution time, which implies greater leakage energy consumption and shortened idle periods for applying DPM. This impact, as we show later on, depends on the nature of the executing workload in terms of its CPU and memory intensiveness and the leakage power characteristics. Therefore, the problem of performing DPM aware DVFS can be also viewed as one of accurate *characterization* and *selection*, where the best suited voltage-frequency setting (hereon referred to as *v-f setting*) is selected based on the characterization of CPU leakage and the executing workload.

Instead of proposing a new policy for DPM and DVFS, we apply online learning [34] to select among a set of possible policies and v-f settings. The online learning algorithm (referred to as *controller*) has a set of *experts* (DPM policies/v-f settings) to choose from and selects an expert that has the best chance to perform well based on the controller's characterization of the current workload. The selection takes into account energy savings, performance delay as well as the user specified energy-performance tradeoff (referred to as e/p tradeoff). The algorithm is guaranteed to converge to best performing expert in the set, thus delivering performance atleast as good as the best expert in the set, across different workloads.

We implement the controller for a server and laptop hard disk drive (HDD), and Intel PXA27x CPU. The controller chooses among a set of policies representing state-of-the-art in DPM, and v-f settings available on the Intel PXA27x CPU. In our experiments, the controller achieved as high as 61% and 49% reduction for HDD and CPU respectively at negligible overhead.

2.2 Design

In this section we first describe how both DPM and DVFS can be formulated as a problem of accurate workload characterization and selection. The selection is done among a set of DPM policies (eg. Fixed Timeout, TISMDP etc.) or allowable v-f settings available on the processor or both depending upon the problem we are targeting. Without the loss of generality, we refer to these policies/v-f settings as *experts*. We then elaborate on our algorithm, which we employ to perform this control activity of workload characterization and expert selection.

2.2.1 System Model

DPM problem is fundamentally a decision problem in which the policy has to decide whether or not to perform a shutdown for a given idle period. As described in section 1.1.1, different DPM policies use different mechanisms to decide this. However, as we show in section 2.4, DPM policies outperform each other under different workloads, devices and user e/p tradeoff preferences. This observation motivates the use of multiple DPM policy experts, where the best suited expert is selected as a function of the current workload characteristics and e/p tradeoff.

DVFS problem is intuitively a problem of selection among the given v-f setting experts of the device (eg. CPU). A lower v-f setting can prolong the execution time of a task, hence shortening the upcoming idle period durations, which has a direct impact on the energy savings possible due to DPM. Longer execution times also cause extra leakage energy consumption, which can offset power savings due to DVFS (explained in detail in section 2.3.2). An ideal DVFS policy must understand this impact and hence perform v-f setting selection with the objective of reducing the overall energy consumption. The primary elements which govern the choice of v-f settings are the characteristics of the executing workload in terms of its CPU/memory intensiveness and the device leakage power characteristics. This observation transforms the DVFS problem into one of accurate modeling of the characteristics of the executing task and device leakage, which in turn drives the
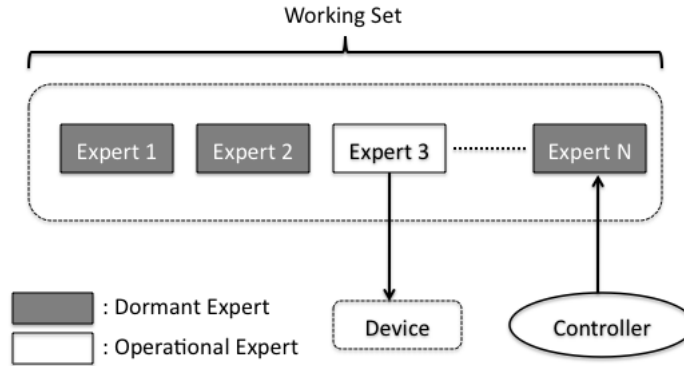
Figure 2.1: Overall System Model.

expert selection.

To summarize, both DPM and DVFS problems can be solved by selection of appropriate expert based on the current workload characterization. For DPM, the characterization is in terms of the distribution of idle period durations, while for DVFS it is in terms of the CPU/memory intensiveness of the executing task. With this background, we now present our system model, which consists of 3 primary entities as shown in Figure 2.1: (1)*Controller*: The core algorithm we employ to perform the workload characterization and expert selection activity for both DPM and DVFS. (2)*Working Set*: It is the set of experts, the controller selects from. An expert can be any DPM policy (for DPM) or any allowable v-f setting (for DVFS). (3)*Device*: Device is the entity whose power is being managed by the controller.

We invoke the controller on an *event*, which we refer to as the *controller event*. For DPM this event is the idle period, i.e. we run the controller whenever an idle period occurs. For DVFS, it is the scheduler tick of the operating system. Scheduler tick is a periodic timer interrupt, which is used by the OS scheduler to make scheduling decisions. Hence, it is the finest granularity at which system updates are performed.

As shown in Figure 2.1, the experts can be in one of the 2 possible states: *dormant* or *operational*. By default all the experts are in the dormant state, and are referred to as the *dormant experts*. When a controller event occurs, the controller on the basis of its model of the current workload selects an expert that has the

highest probability to perform well. This selected expert is referred to as the *operational expert*, which can either be a DPM policy or a v-f setting depending upon the event (idle period or scheduler tick). The amount of time for which the expert stays in the operational state is referred to as the *operative period*, after which it returns to its default dormant state. The operative period for an operational expert in case of DPM is the length of the idle period, while in case of DVFS is the length of the scheduler quantum. In Figure 2.1, this implies that Expert3 has been selected by the controller as the operational expert for the current operative period.

2.2.2 Controller

We adapt Freund et al's on-line allocation algorithm [34] to the problem of DPM/DVFS. A big advantage of the algorithm is that it provides a theoretical guarantee on convergence to the best suited expert in the working set. We present an analysis of the theoretical bound in section 2.2.3.

Table 2.1 contains the pseudo-code for the algorithm. The controller has N experts to choose from; we number these $i = 1, 2 \ldots N$. The experts can be any DPM policy (for DPM) or any valid v-f setting (for DVFS). The algorithm associates and maintains a weight vector $\mathbf{w}^t = < w_1^t, w_2^t \ldots w_N^t >$, where w_i^t is a weight factor corresponding to expert i for operative period t. The value of weight factor, at any point in time, reflects the performance of the expert, with a higher value indicating a better performance. All of the weights of the initial vector $\mathbf{w}^1$ sum to one, as seen in Table 2.1. In our implementation, we assign equal weights to all the experts at initialization.

To perform expert selection, the controller maintains a probability vector $\mathbf{r}^t = < r_1^t, r_2^t \ldots r_N^t >$ where $0 \leq r_i^t \leq 1$, consists of probability factors associated with each expert for operative period t. It is obtained by normalizing the weight vector as shown below:

$$\mathbf{r}^t = \frac{\mathbf{w}^t}{\sum_{i=1}^N w_i^t} \tag{2.1}$$

At any point in time the best performing expert, in accordance with the current workload, has the highest probability factor among all the experts. Thus the

Table 2.1: Algorithm Controller.

Parameters: $\beta \in [0, 1]$

Initialization:

-Initial weight vector $\mathbf{w}^1 \in [0, 1]^N$, such that $\sum_{i=1}^{N} w_i^1 = 1$

-DPM/DVFS specific initialization

For operative periods $t = 1, 2 \ldots$

1: Choose expert with highest probability factor in r^t

2: Operative period starts $\rightarrow$ operational expert takes control of the device

3: Operative period ends $\rightarrow$ DPM/DVFS specific evaluation of experts

4: Set the new weights vector to be: $w_i^{t+1} = w_i^t \cdot \beta^{l_i^t}$

---

controller simply selects the expert with the highest probability factor as the operational expert for the upcoming operative period. If the probability factor of multiple experts is equal, then it randomly selects one of them with a uniform probability (step 1 in Table 2.1).

When the operative period starts, the operational expert takes control of the device (step 2 in Table 2.1). For DPM, the operational DPM expert takes the shutdown decision. For DVFS, the v-f setting corresponding to the operational DVFS expert is applied to the CPU. When, the operative period ends, the controller does an evaluation of all the experts in the working set (step 3 in Table 2.1). The key objective of performing evaluation is to figure out how suitable each expert was for the just concluded operative period. The details on how the controller actually estimates this suitability are DPM/DVFS specific and are provided in the next section. The end result of this evaluation is a loss factor (l_i^t) corresponding to each expert i, which indicates how unsuitable it was for the previous operative period. A higher value indicates higher unsuitability and vice versa.

The final step in the algorithm involves updating the weight factors for each expert on the basis of the loss they have incurred:

$$w_i^{t+1} = w_i^t \cdot \beta^{l_i^t} \tag{2.2}$$

Thus, the weight factors corresponding to experts with higher loss are penalized

more by this simple multiplicative rule. The value of constant β can be set between 0 and 1. The criterion for selecting its appropriate value is explained in [34]. This rule gives higher probability of selecting the better performing experts in the next operative period. Once the weights are updated we are again ready to select the operational expert for next operative period by calculating the new probability vector $\mathbf{r}^t$ using step 1 in Table 2.1.

2.2.3 Performance Bound of Controller

From the previous discussions we know that l_i^t is the loss incurred by each expert for the operative period t. Hence, the average loss incurred by our scheme for a given operative period t in a system with N experts is:

$$\sum_{i=1}^{N} r_i^t l_i^t = \mathbf{r}^t \cdot \mathbf{l}^t \tag{2.3}$$

The goal of the algorithm is to minimize its cumulative loss relative to the loss incurred by the best expert. That is, the controller attempts to minimize the net loss

$$L_G - min_i L_i$$

where, $L_G = \sum_{t=1}^{T} \mathbf{r}^t \cdot \mathbf{l}^t$ is the total loss incurred by controller, and $L_i = \sum_{t=1}^{T} l_i^t$ is individual expert i's cumulative loss over T operative periods. It can be shown [34], that net loss of the algorithm is bounded by $O(\sqrt{T \ln N})$ or that the average net loss per period decreases at the rate $O(\sqrt{\ln N/T})$. Thus, as T increases, the difference decreases to zero. This guarantees that the performance of the controller is close to that of the best performing expert for any workload.

2.3 Implementation Details

The previous section gave an overview of how DPM and DVFS can be modeled as workload characterization and expert selection problems that can be solved using the controller algorithm. In this section, we provide implementation details required for accomplishing this solution. We break this section into two

parts, with the first part discussing details pertaining devices that support only DPM and the second part considering devices that support both DPM and DVFS (specifically CPU). Finally, we show how controller can explicitly be adapted to different leakage regimes.

2.3.1 Devices with only DPM

In this section, we discuss controller implementation with respect to devices that support only DPM such as HDD, memory, network card etc.

Expert Selection As described before, the controller maintains a weight vector $\mathbf{w}^t$ (Section 2.2.2) and its normalized version probability vector $\mathbf{r}^t$ (Equation 2.1) for the experts. At any point in time the best performing expert has the highest probability factor among all the experts and hence, for performing selection, the controller simply selects the expert with the highest probability factor as the operational expert for the next idle period.

The controller evaluates the performance of all the experts at the *end* of the idle period. The evaluation takes into account energy savings, performance delay as well as user specified e/p tradeoff for this update. The energy savings and performance delay caused by the operational expert can be easily calculated since the length of that idle period is known. The dormant experts are evaluated on the basis of how they would have performed had they been the operational experts. We evaluate *loss* with respect to an ideal oracle policy that has zero delay and maximum possible energy savings. Since this loss evaluation takes place at the end of idle period, when we already know its length, we can easily estimate the performance of the ideal oracle policy as well. The value of loss factor (l_i^t) for each expert is influenced by the relative importance of energy savings and performance delay as expressed by factor α ($0 \leq \alpha \leq 1$), which is specified by the user. For this purpose, we break it down into two components: l_{ie}^t and l_{ip}^t, which correspond to energy and performance loss respectively for expert i. The loss factor is then given by equation 2.4:

$$l_i^t = \alpha \cdot l_{ie}^t + (1 - \alpha) \cdot l_{ip}^t \tag{2.4}$$

In our implementation, we determine the energy loss, l_{ie}^t, by comparing the length of the idle period with the sleep time. If it is less than T_{be} (break-even time, defined in Section 1.1.1), then we do not save energy and thus $l_{ie}^t = 1$. For the values of sleep time T_{sleep_i} of an expert i greater than T_{be}, and idle period, T_{idle} we use the following equation:

$$l_{ie}^t = 1 - T_{sleep_i}/T_{idle} \qquad (2.5)$$

Calculation of performance loss, l_{ip}^t, is based on whether the device sleeps or not. If the expert makes the device sleep, $l_{ip}^t = 1$ since we incur performance delay upon wakeup, otherwise it is set to 0.

Once the loss factor is available for each expert, the corresponding weight factor is updated using equation 2.2 (we use β=0.75 in equation 2.2). Once all the weight factors have been updated, the probability factors for all the experts are updated using equation 2.1. Then the operative expert for the next idle period is simply the expert with the highest current probability factor. At any given point in time, the current value of the weight/probability factors is a result of the successive application of equation 2.2 in the previous idle periods. This means, the weight/probability vector actually characterizes the workload or more precisely its idle period distribution in the form of suitability of the different DPM experts for it. By regularly updating it at the end of every idle period, this suitability is updated to keep up with changes in the workload characteristics.

2.3.2 Devices with DPM and DVFS (CPU)

For devices that support both DPM and DVFS (eg. CPU), the controller must determine how to perform a tradeoff between the two. This is important, since DVFS impacts DPM by potentially reducing the idle period durations. We will show in this section that the impact is dominated by two factors, CPU leakage characteristics and CPU/memory intensiveness of the executing task. To understand this, we first present a simple energy model based on the analysis done in [24]. Then using it we show how task and CPU leakage characteristics affect energy savings due to DVFS and DPM.

Energy Model Consider the following equation, which approximates the power consumption in a CMOS circuit:

$$P = D + L, \ D = C_L V^2 f, \ L = I_L V \tag{2.6}$$

where C_L is the load capacitance, V is the supply voltage, I_L is the leakage current, and f is the operational frequency. The first term corresponds to the dynamic power consumption component of the total power consumption (referred to as D), while the second term corresponds to the leakage power consumption (referred to as L).

Let the maximum frequency of the device be f_{max} and the corresponding voltage be V_{max}. The device would consume the maximum power, i.e. P_{max} at this v-f setting. We define P_n, D_n and L_n as the normalized power consumption values, i.e.

$$P_n = \frac{P}{P_{max}} = D_n + L_n = \frac{D}{P_{max}} + \frac{L}{P_{max}} \tag{2.7}$$

We next define λ and ρ as the % contribution of dynamic and leakage power to the total power consumption at the highest v-f setting:

$$\lambda = \frac{D_{max}}{P_{max}}, \ \rho = \frac{L_{max}}{P_{max}} \tag{2.8}$$

If we further define V_n and f_n as the normalized voltage and frequency levels, then D_n and L_n can be rewritten as:

$$D_n = \lambda \frac{D}{D_{max}} = \lambda \frac{C_L V^2 f}{C_L V_{max}^2 f_{max}^2} = \lambda V_n^2 f_n \tag{2.9}$$

$$L_n = \rho \frac{L}{L_{max}} = \rho \frac{I_L V}{I_L V_{max}} = \rho V_n \tag{2.10}$$

Combining equations 2.9 and 2.10, we get:

$$P_n = \lambda V_n^2 f_n + \rho V_n \tag{2.11}$$

We next define T_n as the execution time of a task at v-f setting f_n normalized against the execution time at the maximum v-f setting ($f_n = 1$), T_{max}, i.e.

$$T_n = \frac{T}{T_{max}} \tag{2.12}$$

(a) T_n Estimates

(b) mem ($\mu \approx 0$)

(c) combo ($\mu \approx 0.5$)

(d) burn_loop ($\mu \approx 1$)
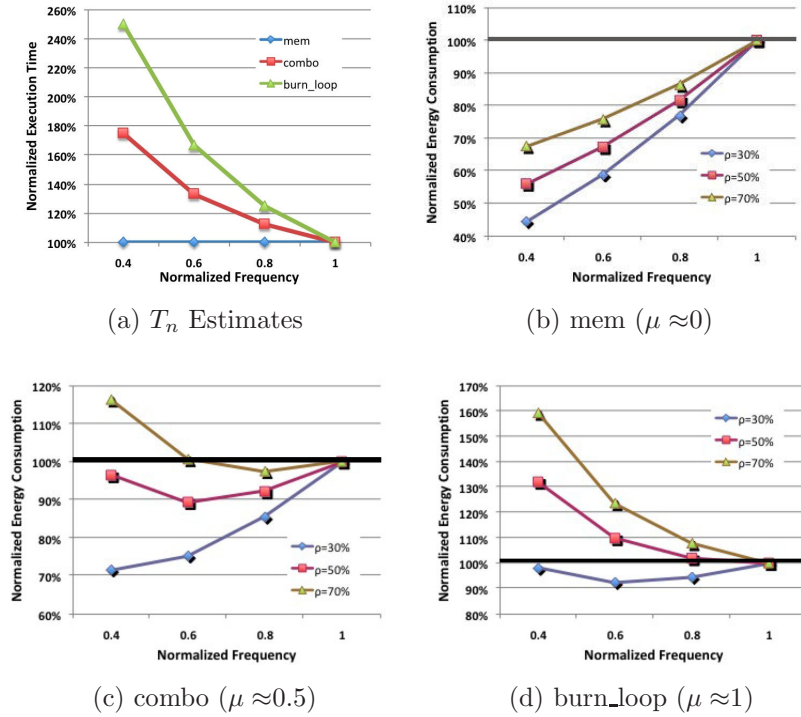
Figure 2.2: Execution Time and Energy Consumption Estimates. For energy estimates the black line indicates the baseline or the energy consumption at $f_n=1$. The region below the baseline indicates energy savings, while the region above indicates higher energy consumption or energy loss.

The normalized energy consumption of an executing task as a function of the normalized voltage, frequency and execution time is then:

$$E_n(V_n, f_n, T_n) = \lambda V_n^2 f_n T_n + \rho V_n T_n \tag{2.13}$$

For any given processor, λ and ρ can be considered to be constants. Given the selected v-f setting (V_n/f_n) and the execution time (T_n) for a task at that setting, we can estimate the CPU energy consumption using equation 2.13.

Task Characterization The execution time of a task can be broken down into two components, 1) T_{cpu}: corresponding to the time during which the execution is CPU bound and hence no stalls occur; 2) T_{stall}: corresponding to the time during which CPU stalls because of memory accesses (cache misses), dependencies etc. If we define T_{max}, $T_{cpu_{max}}$ and $T_{stall_{max}}$ as the durations at the maximum v-f setting

$(f_n = 1)$, and T_{ncpu} and T_{nstall} as normalized versions of T_{cpu} and T_{stall} respectively at f_n, then:

$$T = T_{stall} + T_{cpu}$$

$$T_{max} = T_{stall_{max}} + T_{cpu_{max}}$$

$$T_n = \frac{T}{T_{max}} = T_{nstall} + T_{ncpu} \tag{2.14}$$

During T_{stall}, the CPU is waiting for the cause of stall to be resolved. For instance, when there is a cache miss, CPU is stalled waiting for the memory access to complete. Thus, duration of T_{stall} is independent of the frequency setting of the CPU, since the CPU is not executing instructions. This means, that T_{nstall} is constant across all the v-f settings, and hence also equal to $T_{nstall_{max}}$, the normalized duration at the maximum v-f setting ($f_n = 1$):

$$T_{nstall} = T_{nstall_{max}} = \frac{T_{stall_{max}}}{T_{max}} \tag{2.15}$$

In contrast, during T_{cpu}, the execution is cache intensive, and thus its duration depends on the number of cycles it takes to access the cache and CPU registers. The duration of this cycle is directly proportional to frequency of the CPU, and hence, a reduction in frequency results in a proportional increase in T_{ncpu}:

$$T_{ncpu} = \frac{T_{ncpu_{max}}}{f_n}, \text{ where } T_{ncpu_{max}} = \frac{T_{cpu_{max}}}{T_{max}} \tag{2.16}$$

Combining equations 2.14, 2.15 and 2.16 we get:

$$T_n = T_{nstall_{max}} + \frac{T_{ncpu_{max}}}{f_n} \tag{2.17}$$

If we define $T_{ncpu_{max}} = \mu$, then $T_{nstall_{max}} = (1 - \mu)$ since T_n at the maximum v-f setting ($f_n = 1$) is 1. The factor μ indicates the degree of CPU and cache intensiveness of a task, with a high value indicating high CPU intensiveness, and a low value indicating otherwise. Substituting these values in equation 2.17, we get:

$$T_n = (1 - \mu) + \frac{\mu}{f_n} \tag{2.18}$$

On the basis of this discussion, we define three tasks with different characteristics: (1) *burn_loop*: highly CPU and cache intensive ($\mu \approx 1$), (2) *mem*: highly stall

intensive due to memory accesses ($\mu \approx 0$) and (3) *combo*: combination of the previous two ($\mu \approx 0.5$). Figure 2.2a, shows T_n estimates based on equation 2.18 for these three tasks across four different f_n values (100%, 80%, 60%, 40%). It illustrates that T_n for highly memory/stall intensive task *mem* is fairly insensitive to changes in f_n, while for CPU intensive *burn_loop* it increases in proportion to decrease in f_n. To verify these estimates we implemented three benchmarks with such characteristics and ran them on an Intel PXA27x CPU [54] at 520MHz/f_n=100%, 416MHz/f_n=80%, 312MHz/f_n=60% and 208MHz/f_n=40%. We found the estimated values of T_n to be on an average within 1% of the actual measured values. This shows that our analysis of T_n estimation is accurate and emphasizes the importance of task characteristics for modeling its execution times.

Impact of task and leakage characteristics To understand the impact of task (μ) and CPU leakage (ρ) characteristics on the energy savings at different f_n values, we next estimate the energy consumption (E_n) of these three tasks using equation 2.13. Figures 2.2b-d show E_n for benchmarks *mem*, *combo* and *burn_loop* at different f_n values for platforms with different leakage percentage (ρ) values (30%, 50% and 70%). For a modern day processor like Intel PXA27x, ρ is around 30%, and is expected to rise further in future CPUs according to industry estimates [49]. We confirmed this value of ρ for PXA27x CPU by measuring the current flowing into the processor employing a 1.25Msamples/sec DAQ at different v-f settings. We also measured the energy consumption of these benchmarks on PXA27x CPU ($\rho \approx 30\%$), and found them to be on an average within 1% of the theoretically estimated values for $\rho \approx 30\%$ using equation 2.13. This confirmed the validity of our analysis and estimates. We used V_n values of (80%, 86.6%, 93%, 100%), which correspond to the v-f settings on the PXA27x CPU, for these estimates (refer to Table 2.7).

Figures 2.2b-d show the following. First, increasing the fraction of leakage (ρ) value results in a reduction in energy savings consistently across all the tasks. The reason for this is that DVFS brings just a linear decrease in leakage power, compared to cubic in active power (refer equation 2.6). Hence its effectiveness begins to diminish with increasing ρ. Second, the energy savings decrease due to

the increased execution times of the task. For *mem*, which has no performance penalty across different f_n values, the gain in energy savings is significant with decreasing frequency because of the lower voltage. Task *burn_loop* incurs the highest performance penalty, which manifests in low energy savings at ρ=30% and higher energy consumption than the baseline (f_n=100%) at higher ρ values. Same can be observed for *combo* at ρ=70%, which means it is no longer energy efficient to run *burn_loop* and *combo* at lower v-f settings. It is better to run such tasks at the highest v-f setting (i.e. no DVFS), and then switch to DPM when they become inactive. This indicates that task and leakage characteristics determine the tradeoff between DVFS and DPM, and hence it is important to take both into account. Based on these observations, we now discuss how the controller incorporates estimation of these characteristics into its design.

Expert Selection The controller maintains two weight vectors: one for the DPM experts (policies) and second for the DVFS experts (v-f settings). During the idle periods, when the idle task is scheduled by the OS, the controller selects among the DPM experts as explained in section 2.3.1. During active execution periods, the controller selects among the DVFS experts based on the characteristics of the executing task. Since in a multi-tasking environment, tasks with differing characteristics can be runnable at the same time, the controller maintains a per task weight vector. These weight vectors are initialized when the task is created.

The controller has a form of a meta-policy for DPM, where it performs selection among multiple DPM policies, while for DVFS it is a policy that selects among multiple v-f settings. There are two primary reasons we did not implement a meta-policy for DVFS like DPM. First, different DVFS policies operate under different setups, which might be difficult to put together on a given CPU. For instance, the policy in [19] operates on every scheduler tick, while, the one in [56] operates at 100 million instruction intervals, which might or might not co-incide with a scheduler tick. As a result, there is no common controller event, where it can evaluate these two policies and select an operational policy. Second, different DVFS policies also make assumptions about the systems they operate on, help from applications or compilers etc. ([57, 7] etc.), which might not hold on other

systems.

Workload characterization for DVFS involves accurately measuring the degree of CPU intensiveness or μ (refer equation 2.18) of the executing workload. In order to estimate μ we use the concept of *CPI stack*, which breaks down processor execution time into a baseline CPI plus a number of miss event CPI components like cache and TLB misses etc. [32]. The following equation represents the average CPI in terms of its' CPI stack components:

$$\mathrm{CPI}_{avg} = \mathrm{CPI}_{base} + \mathrm{CPI}_{cache} + \mathrm{CPI}_{tlb} + \mathrm{CPI}_{branch} + \mathrm{CPI}_{stall} \qquad (2.19)$$

CPI stacks give insight into the CPU-intensiveness of the currently executing task. For instance, a high value for $\mathrm{CPI}_{base}/\mathrm{CPI}_{avg}$ ratio indicates that CPU is busy executing instructions for majority of the time, thus indicating a higher μ, and vice versa.

In order to dynamically characterize executing workload, we construct its CPI stack at runtime. For the PXA27x processor we do this by using the *performance monitoring unit* (PMU). The PMU is an independent hardware unit with four 32-bit performance counters that can be used to monitor any four out of 20 unique events available simultaneously. We monitor the number of instructions executed (INSTR), data cache misses (DCACHE), cycles instruction cache could not deliver instruction (ICACHE) and cycles processor stalls due to dependency (STALL). We also get the total number of clock cycles (CCNT) elapsed since the PMU was started in order to calculate the CPI components:

$$\mathrm{CPI}_{avg} = \mathrm{CCNT/INSTR}, \ \mathrm{CPI}_{dcache} = (\mathrm{DCACHE} \times \mathrm{PEN})/\mathrm{INSTR}$$

$$\mathrm{CPI}_{icache} = \mathrm{ICACHE/INSTR}, \ \mathrm{CPI}_{stall} = \mathrm{STALL/INSTR} \qquad (2.20)$$

In this equation, CPI_{cache} has been broken down into CPI_{icache} and CPI_{dcache} and PEN is the average miss penalty for a data cache miss (we used PEN=75 cycles at 520MHz in our experiments). Note that CPI_{tlb} and CPI_{branch} are missing. This is because we can monitor only 4 events at a time, and in our experiments we found the events being monitored more indicative of the task characteristics. Hence, we can estimate CPI_{base} as follows:

$$\mathrm{CPI}_{base} = \mathrm{CPI}_{avg} - \mathrm{CPI}_{icache} - \mathrm{CPI}_{dcache} - \mathrm{CPI}_{stall} \qquad (2.21)$$

Finally we estimate μ as a ratio of CPI_{base} to CPI_{avg} (equation 2.22).

$$\mu = \text{CPI}_{base}/\text{CPI}_{avg} \qquad (2.22)$$

With CPU-intensiveness (μ) of the task estimated, we now describe the loss evaluation stage for the DVFS-experts. To evaluate the loss factor of each expert, we define a data structure called μ-mapper, which maps the suitability of v-f settings to the characteristics of the task as a function of the CPU leakage characteristics. For instance, for CPU with ρ=30%, Figures 2.2b-d indicate that the best suited frequency scales linearly with μ of a task. Based on this observation, Figure 2.3a shows a μ-mapper for a CPU with five experts, where the frequencies increase in equal steps from 0 to Expert5. The μ-mapper divides the domain of μ ($0 \leq \mu \leq 1$) into intervals for each expert, and sequentially maps each interval to the successive experts. This captures the fact that higher frequency experts are better suited for tasks with higher μ. The mean of each interval, μ-mean, is associated with their respective expert, and is used for performing weight update. If CPU has higher leakage component (eg $\rho = 50\%$), then the μ-mapper is changed accordingly. For instance, if we compare the energy results in Figures 2.2b-d for ρ=50% and ρ=30%, we can see that the energy consumption of *combo* for ρ=50% is similar to that of *burn_loop* at ρ=30%. This means that for CPUs with such high leakage, workloads with medium CPU intensiveness need to be run at higher v-f settings, while highly CPU-intensive tasks, such as *burn_loop*, must run only at the highest v-f setting. Thus, to derive a μ-mapper for such a CPU, we will map only 0-0.5 of μ's sample space to the available v-f settings. Figure 2.3b shows the corresponding μ-mapper. This way μ-mapper provides the controller with the critical information on CPU leakage characteristics and allows it to adapt seamlessly across different CPUs without any modifications to the core algorithm.

Given a μ-mapper, the loss factors can be easily evaluated by comparing μ to the μ-mean of each expert. This calculation takes both the energy savings and performance delay into account by breaking this loss factor into 2 components, the energy loss (l_{ie}^t) and the performance loss (l_{ip}^t). If $\mu < \mu$-mean for an expert, then the task is more stall intensive with respect to the given expert and hence can afford to run slower. At the same time it means that this expert would cause no

(a) $\rho=30\%$

Note:
Freq(Expert1)<Freq(Expert2)<Freq(Expert3)<Freq(Expert4)<Freq(Expert5)

(b) $\rho=50\%$
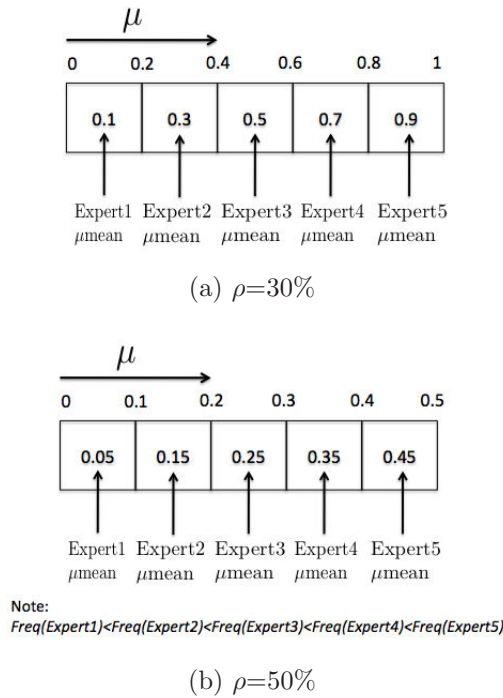
Figure 2.3: Example of μ-mappers for CPUs with different ρ values.

Table 2.2: Loss Evaluation Methodology.

	Energy Loss (l_{ie}^t)	Perf Loss (l_{ip}^t)
$\mu > \mu\text{-mean}$	0	($\mu - \mu\text{-mean}$)
$\mu < \mu\text{-mean}$	($\mu\text{-mean} -\mu$)	0
Total Loss ($l_i^t) = \alpha \cdot l_{ie}^t + (1 - \alpha) \cdot l_{ip}^t$		

performance delay for the current task, since it corresponds to a higher frequency than required. Similarly, there is a performance loss but no energy loss when $\mu > \mu\text{-mean}$. Table 2.2 summarizes how we evaluate the loss factor. The α factor in Table 2.2 is similar to the one used for DPM (equation 2.4); a user defined value that determines the e/p tradeoff. Once the loss factors are evaluated for each expert, the controller updates the weights of all the experts using the equation 2.2. The controller then restarts the PMU so that μ for the upcoming scheduler quantum can be evaluated at its conclusion.

The task weight vector accurately characterizes the task it represents, since it encapsulates all the updates based on previous μ values. Besides, the weight

Table 2.3: Device and Workload Characteristics for HDD.

Device	Trace	Duration (sec)	$\bar{t}_{idle}$	$\sigma_{\bar{t}_{idle}}$	P_{on} (W)	P_{sleep} (W)	T_{be} (sec)
HP	HP-1	32311	30	91	1.6	0.4	4.2
	HP-2	35375	25	86			
	HP-3	29994	36	119			
Laptop	Laptop-1	18017	17	111	0.95	0.13	5.2
	Laptop-2	7528	11	31			

$\bar{t}_{idle}$: Average Idle Period Duration (in sec)

updates are based on both the task as well as the CPU leakage characteristics (μ-mapper). This ensures that the controller understands when is it beneficial to perform aggressive DVFS or not from overall energy efficiency point of view (as described in the previous paragraphs).

2.4   Experiments and Results

We performed our experiments using the following devices: a hard disk drive (HDD) and Intel PXA27x core (CPU) with real life workloads. For HDD we used the controller with just DPM enabled because of lack of DVFS functionality, while for CPU we used controller with both DPM and DVFS enabled. The results indicate that our controller is capable of dynamically adapting while delivering sizable (as high as 60%) energy savings over a range of e/p tradeoff settings.

2.4.1   HDD (DPM)

We evaluated our online learning based DPM algorithm by studying both server and laptop HDDs. We used two set of workload traces: 1) originally collected on an HP server [89] (referred to as HP traces), 2) traces collected on a laptop hard disk (referred to as laptop traces) [90]. The characteristics of workloads selected are described in Table 2.3. This is a broad range of workload characteristics. For example, HP-1 and HP-3 traces have very different distribution of idle time durations in terms of both average value and standard deviation ($\bar{t}_{idle}$ and $\sigma_{\bar{t}_{idle}}$ respectively). We consider the HDD to be idle after 1s of inactivity. This threshold

Table 2.4: Working set characteristics for HDD.

Expert	Characteristics
Fixed Timeout	Timeout $= 3T_{be}$
Adaptive Timeout [30]	Initial Timeout $= T_{be}$ Adjustment $= +1T_{be}/-1T_{be}$
Exponential Predictive [53]	$I_{n+1} = \alpha \cdot i_n + (1 - \alpha) \cdot I_n$ with $\alpha = 0.5$
TISMDP [90]	Optimized for delay constraint of 2.3% on HP-3 Trace

is based on the observation that across all the workloads, many idle periods are smaller than 1s. These idle periods incur performance delay without contributing much to the energy savings. Table 2.3 also shows the device characteristics in terms of P_{on} and P_{sleep}, which refer to the power consumed when the devices are on and in the sleep state respectively. T_{be} refers to the break even time. We run the workload traces described in Table 2.3 and record the performance in terms of energy savings and performance delays for both the individual experts as well as the controller.

For our working set we select fixed timeout, adaptive timeout [30], exponential predictive [53] and TISMDP [90] policies, representing different classes of state of the art DPM policies. While fixed and adaptive timeout policies represent the timeout class, exponential predictive policy represents the predictive class and TISMDP represents the stochastic class of policies. Table 2.4 describes the precise characteristics of the DPM policy experts employed for the experiments. The fixed timeout employs a timeout equal to three times the break even time or T_{be}. The adaptive timeout policy uses the T_{be} as the initial timeout with an adjustment factor of $+1T_{be}/-1T_{be}$ depending on whether the previous idle period resulted in energy savings or not. Exponential predictive policy is implemented as described in [53] without pre-wakeup. It predicts the length of the upcoming idle period (I_{n+1}) using the actual (i_n) and predicted (I_n) lengths of the previous idle period. TISMDP policy is optimized for a given delay (2.3%) on the HP-3 trace. The main idea we are trying to show is that given a set of experts, the controller

always converges to select the best performing expert at any point in time.

Table 2.5 shows the results achieved in terms of energy savings and performance delay for the individual experts on the HDD traces. The *%energy* indicates the amount of energy saved relative to the case where we do not have any DPM policy while the *%delay* shows the amount of performance delay caused relative to the total timeframe because of power management. The first row of the table gives the results for the oracle policy. It is an ideal offline policy which knows the workload in advance and hence always takes the optimal decision for each idle period, so its performance delay is zero. The results highlighted in black show where we get the best energy savings while the results highlighted in gray show the case where we get the least performance delay. We can notice that the HP traces predictive policy does well in terms of saving energy. For instance, on HP-1, it achieves around 58% energy savings. It performs equally well for the other workloads as well. However, predictive policy is also the worst in terms of causing performance delay, since it is extremely aggressive in turning off the HDD and thus incurs delay while waking up. In contrast, TISMDP causes the least performance delay and consequently fetches the least energy savings. It can be observed in Table 2.4 that TISMDP was optimized for 2.3% delay on HP-3 workload and the results achieved confirm this. However, the figure is not the same for HP-1 and HP-2 workloads which confirms that it is optimal for stationary workloads and does not adapt with changing workloads. Fixed timeout performs reasonably well on both the accounts while adaptive timeout is quite close to predictive in terms of energy savings.

Similarly Table 2.5(b) shows the results with individual experts for the laptop traces. We can observe that for these traces the predictive expert performs the worst in terms of both energy savings as well as performance delay. This is in contrast to HP traces, where it did the best in terms of energy savings. This happens due to the smaller idle periods of laptop traces (Table 2.3) and lack of co-relation between successive idle period durations, which causes the predictive expert to cause many wrong shutdowns (where $T_{idle} < T_{be}$). These results highlight that different classes of policies, depending upon their characteristics, deliver dif-

Table 2.5: Energy Savings/Performance Delay Results for HDD with Individual Experts.

Expert	HP-1 Trace		HP-2 Trace		HP-3 Trace	
	%delay	%energy	%delay	%energy	%delay	%energy
Oracle	0	68.68	0	66.58	0	71.38
Timeout	3.95	44.4	4.08	41.28	3.23	50.44
Ad Timeout	6.61	57.2	7.35	54.12	5.27	60.63
TISMDP	2.9	38.7	2.7	36.5	2.29	44.6
Predictive	7.32	58.2	8.32	54.9	5.87	60.7

(gray shade indicates min perf delay and black indicates max energy savings)

(a) HP

Expert	Laptop-1 Trace		Laptop-2 Trace	
	%delay	%energy	%delay	%energy
Oracle	0	76	0	65
Timeout	1	59	1.5	45
Ad Timeout	2	65	3.3	53
TISMDP	0.7	47	0.9	32
Predictive	4.1	42	6.7	30.2

(b) Laptop

ferent levels of performance across different workloads. However, depending upon the application requirements or user preferences one might want the overall performance to be more delay sensitive or more energy sensitive. The problem with just having a single DPM policy is that it does not offer the flexibility to control this behavior. The Controller offers exactly this flexibility.

Table 2.6 shows results achieved on the same traces using the controller with different e/p tradeoff (α) settings. As explained in Section 2.3.1, α value indicates the desired e/p tradeoff setting. A high value indicates a higher preference to energy savings, a low value indicates higher preference to performance while a medium value indicates a reasonable ratio of both. In our experiments we tested with values of α ranging from around 0.3 (low) to 0.7 (high). We used values of α around 0.5 for the medium value. As we increase the value of α, we get higher energy savings and for lower values of α, we get low performance delay. For instance, on HP-2 workload we get 50.1% energy savings for high α, which is quite close to that achieved by predictive and adaptive timeout policies. In contrast, for

Table 2.6: Energy Savings/Performance Delay Results for HDD with Controller.

Preference	HP-1 Trace		HP-2 Trace		HP-3 Trace	
	%delay	%energy	%delay	%energy	%delay	%energy
Low α	2.95	39.27	2.8	36.6	2.5	45.9
Med α	4.17	47.2	4	43.4	3.33	53.2
High α	5.76	55.2	6.21	50.1	4.53	57.9

(a) HP

Preference	Laptop-1 Trace		Laptop-2 Trace	
	%delay	%energy	%delay	%energy
Low α	0.7	49	1	33.5
Med α	0.8	54	1.3	41
High α	1	61	1.7	48

(b) Laptop
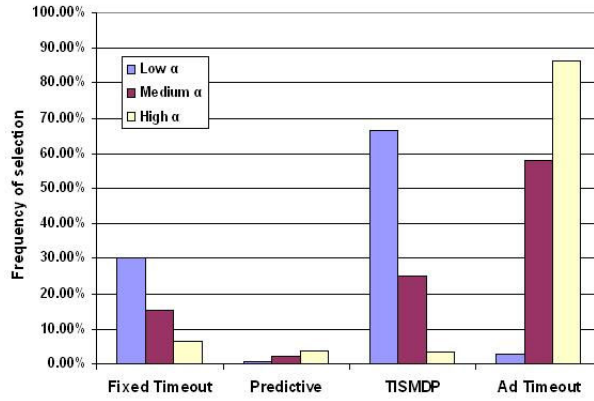
Figure 2.4: Frequency of selection of experts for HP-3 trace.

low α we get performance delay comparable to that of TISMDP. Remember that we limit the values of α between 0.3 and 0.7. For even higher values (close to 1) we achieve energy savings even closer to that of adaptive timeout and predictive policies. Similarly for the laptop traces we observe results converging to that of TISMDP for low α, and to that of adaptive timeout for high α.

Figure 2.4 shows how the frequency of selection of experts changes with α on HP-3 trace. For higher value of α, adaptive timeout expert is selected most often since it achieves close to highest energy savings (60.6%) at a lower performance delay than predictive expert. For lower values of α, TISMDP expert is selected
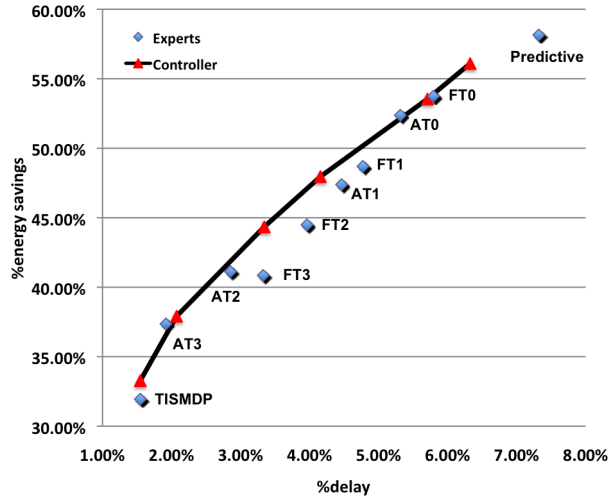
Figure 2.5: Comparison of e/p tradeoff of controller with multiple experts for HP-1 trace. The black line connects the different e/p tradeoff points of the controller.

with higher frequency since it is conservative in turning off the HDD and thus offers lower performance delays. For the medium value of α, we can see that it selects among all the experts to deliver a performance which offers a reasonable e/p tradeoff. Hence, α factor offers us a simple yet powerful control knob to obtain the desired e/p tradeoff.

We next show that our controller can form a Pareto optimal curve over a set of experts. The experts consist of TISMDP, predictive, four adaptive timeout (AT0-3) and four fixed timeout (FT0-3) policies. We run the controller at six different α values (ranging from low of around 0.3 to high of around 0.7) to get different e/p tradeoffs. Figure 2.5 shows a line connecting the e/p tradeoff points offered by the controller, and the e/p tradeoff points offered by the individual experts. The line divides the e/p space into two parts: (1) the part above the line represents e/p tradeoff points that are better than those offered by the controller since they either offer lower performance delay for same energy savings and vice versa, (2) the part below the line represents e/p tradeoff points that are inferior to those of controller. Figure 2.5 illustrates that the e/p tradeoff points of individual experts either lie on this line or below it. Therefore, based on the e/p preference (α), the performance of the controller either converges to that of the best expert or is even superior to any of the experts. Better results than any single expert
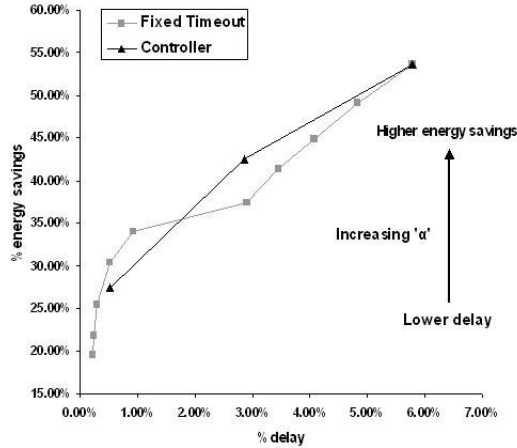
Figure 2.6: Comparison of e/p tradeoff of controller with fixed timeout experts for HP-1 trace. The black and gray lines connects the different e/p tradeoff points of the controller and the fixed timeout experts respectively.

are possible due to the fast convergence property of the controller to the best performing experts over different phases of the workload.

Selection with Fixed Timeout Policies: We next test our controller with a working set of eight simple fixed timeout policies that have timeout values ranging from T_{be} to as large as 50s for HDD. A timeout of T_{be} guarantees that the energy consumption is not be greater than a factor of 2 when compared to an ideal offline policy [59]. Timeout of 50s represents a conservative policy to keep the performance delay low.

Figure 2.6 illustrates the performance of these individual timeout policies against the performance of the controller corresponding to the HP-1 workload. The line in Figure 2.6 shows e/p points for the controller algorithm with five different values for α. Just like Figure 2.5, all the e/p tradeoff points for the fixed timeout experts are either close to the line or below it, showing that our controller can achieve pareto optimality. For high values of α, we get energy savings as high as 54%, which is very close to that achieved by controller using a working set of more sophisticated policies (see Table 2.5). For a low value of α, it gives achieves a very low performance delay of just 0.5%. This clearly shows that the controller can achieve competitive e/p tradeoffs just with a set of simple fixed timeout policies.
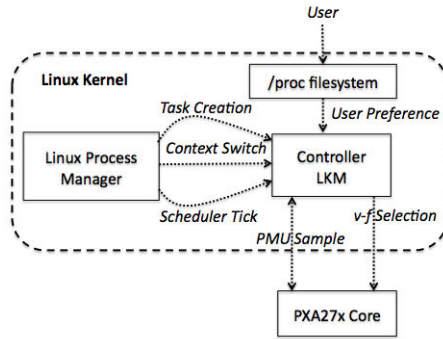
Figure 2.7: System Level Implementation of Controller for CPU.

2.4.2 CPU (DPM+DVFS)

In this section, we present results for the controller with both DPM and DVFS enabled. We use the CPU of the Intel PXA27x platform (running Linux 2.6.9) as the testbed for these experiments. Figure 2.7 shows details of our system level implementation of the controller. We implement the controller as a Linux loadable kernel module (LKM). As soon as the LKM is loaded, it performs both the DPM and DVFS specific initializations. For DPM it initializes the weight and probability vectors corresponding to the DPM policy experts. For DVFS, it scans the available v-f setting experts and calculates the corresponding μ-means based on the μ-mapper shown in Figure 2.3a. This is based on our experiments with PXA27x (refer to section 2.3.2) that indicate $\rho \approx 30\%$.

As shown in Figure 2.7, the LKM is closely knit to the Linux process manager. To isolate the characterization in terms of CPU/memory intensiveness on a per task basis and preserve them across context switches, the Linux task data structure *task_struct* is modified to include the weight vector. This is required for DVFS but not for DPM since in that case we model the inactivity of the system, and when the system gets inactive the only task that runs is the idle thread. The controller always has a pointer to the current task data structure, and this ensures that the update and selection of experts is occurring for the current task only. The LKM also exposes a */proc* interface to the user, which can be used to specify the e/p tradeoff (α). The LKM receives notifications from the process manager on the following 3 events: 1) *Task Creation*: The process manager provides LKM with

Table 2.7: Device Characteristics: CPU.

P_{on}	P_{idle}	$P_{standby}$	P_{tr}	T_{tr}	T_{be}
747mW	129mW	1.7mW	747mW	12ms	12ms

(a)

Freq (MHz)	Voltage (V)
208	1.2
312	1.3
416	1.4
520	1.5

(b)

the pointer to the new *task_struct*, and the LKM initializes the per task DVFS weight vector. 2) *Scheduler Tick*: This acts as a controller event for DVFS. 3) *Context Switch*: On this notification, the LKM checks if the next thread that is getting scheduled is the idle thread. If it is the case then it switches to the lowest v-f setting in the working set and selects a DPM expert since it indicates beginning of an idle period.

The Linux 2.6.9 kernel has a scheduler quantum of 10ms. This means that the scheduler runs every 10ms, irrespective of whether there are any schedulable threads in the system or not. This is clearly energy inefficient since it makes 10ms the upper bound on the maximum achievable idle time for the CPU. This renders most of the low power modes of the PXA27x CPU (standby, sleep etc.) unusable, since they have larger break even times. To solve this problem, dynamic tick support has been added to the latest kernel version (2.6.24)[43]. This allows reprogramming of scheduler tick when there are no schedulable threads in the system, hence achieving longer idle periods. However, there is no publicly available support for the port of this kernel version on the PXA27x platform, with the latest being 2.6.9 [44]. The absence of dynamic tick support made practical experiments of DPM for CPU impossible because of the periodic scheduler ticks. Hence, we performed experiments for DPM based on simulations on traces of real life workloads collected using the LKM. The LKM calculates the idle periods as durations for which the idle thread stays scheduled at a stretch. The approach is very similar to the one used in [12].

Hence, in our current setup, we run the workloads with and without DVFS enabled, and calculate CPU energy savings by current measurements using a 1.25M

samples/sec DAQ. The energy savings and performance delay values are compared to a system, which: 1) runs at the highest v-f setting (1.5V/520MHz in our case) when active, 2) switches to idle mode and the lowest v-f setting (1.2V/208MHz) when idle. While running the workloads the LKM also generates a trace of the idle period distribution. This allows us to estimate the energy savings due of DPM offline using the characteristics listed in Table 2.7a. The DPM working set comprises of 9 fixed timeout policies with timeouts ranging from T_{be} to $9T_{be}$. We have shown in section 2.4.1 on HDD that working set of fixed timeout policies performed nearly as well as the one with more sophisticated policies. For DVFS, we use working set of v-f setting experts listed in Table 2.7b.

Workloads for CPU can be divided into two categories: *idle dominated* and *computationally intensive.* Idle dominated workloads are the ones for which the idle thread is scheduled most of the time. All user interface related workloads like word processing, web surfing etc. belong to this category. The computationally intensive workloads are the ones for which the idle thread never gets scheduled. This category includes tasks like decompressing, running intensive encryption/decryption algorithms etc. Typical real life applications are a mix of both. For instance, the workload for a user using GUI based archiving application would be mostly idle dominated when he is browsing the various options, and computationally intensive in bursts when he enters password for authentication or actually compresses/decompresses some file. In the next two sections we present the results for these kinds of workloads separately.

Idle Dominated Workloads

For these experiments we used two real life workloads: *editor* and *www.* The *editor* workload involves use of *vi* editor for writing and reviewing data files and use of some basic shell commands (ls etc.). The *www* workload involves general web surfing (search, reading articles etc.) on a wireless interface using the lynx web browser. We analyzed the results for the following 3 configurations of the controller: only DPM, only DVFS, both DPM and DVFS.

For the first case, we disabled DVFS and kept DPM enabled. Table 2.8

Table 2.8: Energy Savings/Performance Delay Results for CPU on Idle Dominated Workloads.

Preference	editor		www	
	%delay	%energy	%delay	%energy
Low α	0.41	17	0.31	19
Med α	0.81	20	0.85	22
High α	1.56	24	1.15	25
Oracle	0	26	0	27

Note: The %energy savings are baselined against a CPU that
is in the idle mode (clocks halted) and the lowest v-f setting during idle periods

shows the results achieved for with the two workloads for different values of α. The *%energy* numbers indicate the energy savings baselined against the case where the CPU is placed in idle mode (which halts its clock supply) and the lowest v-f setting as soon as it gets idle. The *%delay* is the overhead incurred because of power management. We again used values around 0.3, 0.5 and 0.7 for low, medium and high values of α. The last row shows the results for the oracle policy, which shows the maximum achievable energy savings for these workloads using DPM. We can see in Table 2.8 that for both the workloads, with increasing value of α, the energy savings increase. For high α, the energy savings are around 25% for both *editor* and *www*, which is very close to the energy savings achieved by the oracle policy.

Table 2.9 shows the frequency of selection of experts for the different α settings for both the *editor* and *www* workloads. We can observe that for low α, $9T_{be}$ is the most selected expert since it offers the least possible delay across all the experts. As we move towards higher values of α, the frequency of selection of smaller timeout experts increases since they offer higher energy savings. Infact for *www* workload, T_{be} expert gets selected for all idle periods. The table also shows that the nature of workload affects the expert selection even for the same value of α. For instance for medium value, T_{be} expert gets selected for 37% with the *editor* workload, but for 65% with the *www* workload.

We next experiment the controller with DVFS enabled and DPM disabled for different values of α. The energy savings are negligible for both *editor* and *www* across all the values of α since they spend most of their time in the idle thread.

Table 2.9: Frequency of selection of experts (%).

Expert	editor			www		
	Low α	Med α	High α	Low α	Med α	High α
T_{be}	1.4	37	84	10.9	65	100
$2.T_{be}$	3.7	29.7	14.3	0.8	13.7	0
$3.T_{be}$	1.4	7	1.8	2	1.6	0
$4.T_{be}$	0.6	2.4	0	0.4	0.4	0
$5.T_{be}$	4.7	1	0	0	0	0
$6.T_{be}$	3.5	0.6	0	0	0	0
$7.T_{be}$	25.5	5.5	0	2.4	2.4	0
$8.T_{be}$	0	0	0	17.3	12.5	0
$9.T_{be}$	59.1	16.7	0	66.1	4.4	0

As described above, our baseline CPU switches to the idle mode and the lowest v-f setting available in the working set (1.2V/208MHz) once idle. Thus, there are no additional savings possible because of DVFS during idle periods. The energy savings for the active periods due to DVFS are negligible, since they are very small compared to the idle periods. The performance delay is almost zero across all α values as the active periods are not long enough to observe significant performance loss.

In the last set of experiments, we enable both DPM and DVFS. In this case we observe that the energy savings and performance delay results converge to those in Table 2.8, i.e. the case with just DPM enabled. The reason for this is that we now switch to DPM as soon as the idle thread is scheduled, where these workloads spend most of their time. This shows that DPM is very effective for idle dominated workloads.

Computationally Intensive Workloads

We experimented with a number of computationally intensive workloads in both single and multitasking environments. For such workloads, the idle thread is not scheduled, and thus, the energy savings are exclusively due to DVFS. The chosen workloads include common UNIX utility gzip for decompression (*dgzip*) and 3 benchmarks taken from an open source benchmark suite mibench [39]: *bf* (blowfish) - a symmetric block cipher; *djpeg* - decoding a jpeg image file; *qsort* - sorting a large array of strings in ascending order. All these workloads have a

Table 2.10: Energy Savings/Performance Delay Results for CPU on Computationally Intensive Workloads.

Benchmarks	Low α		Med α		High α	
	%delay	%energy	%delay	%energy	%delay	%energy
qsort	6	17	17	32	25	41
djpeg	7	21	15	37	27	45
dgzip	15	30	21	42	28	49
bf	6	11	16	28	25	40
burn_loop	0	0	10	2.5	25	5

(a) Single Tasks

Benchmarks	Low α		Med α		High α	
	%delay	%energy	%delay	%energy	%delay	%energy
qsort+djpeg	6	17	15	33	25	41
dgzip+djpeg	13	24	19	40	27	49
qsort+dgzip	7	20	18	35	26	42
dgzip+bf	11	17	20	31	26	45

(b) Multi-Tasking

mix of CPU and memory intensive phases. We have also added results for the synthetic workload *burn_loop* (section 2.3.2) to see how the controller performs for highly CPU intensive workloads, which benefit the least from DVFS. The results achieved under both single and multi task environments for these benchmarks are illustrated in Table 2.10. We discuss them separately.

Single Task Environment Table 2.10(a) displays the results we achieved for each individual benchmark in a single task environment. From the results we can
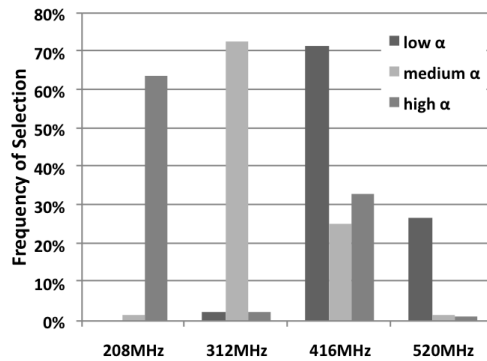
Figure 2.8: Frequency of selection of experts for qsort benchmark.
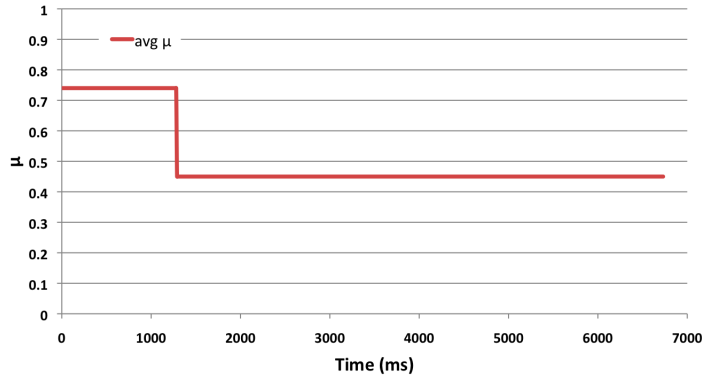
Figure 2.9: Plot of average μ of qsort benchmark across its execution timeline.

observe that as we increase the value of α, we get higher energy savings. For lower values of α, we get lower performance delay. For instance, with *qsort*, the delay is just 6% for a low value of α, while the energy savings are as high as 41% for a high value.

Figure 2.8 shows the frequency of selection of different experts for *qsort* according to the selected value of α. For higher value of α, the 208MHz expert is selected for 65% of the time, while rest of the time 416MHz expert is chosen. This suggests that *qsort* has both CPU and memory intensive phases. Figure 2.9 plots the average μ of *qsort* along its execution timeline and illustrates these phases. We can observe that around the first 20% of its execution, *qsort* is consistently very CPU intensive (high μ), for which the controller selects 416MHz expert. Beyond that it varies, but the average is on the lower side (around 0.45), for which the controller mostly selects the 208MHz expert. Thus, the controller can quickly and accurately identify CPU/memory intensive phases in the workload and adapt at runtime.

We perform offline analysis to evaluate the maximum achievable energy savings for these benchmarks for the given working set by running all the individual benchmarks statically at 208MHz/1.2V. This is inline with our observation in section 2.3.2 (Figure 2.2b-d), that for $\rho \approx 30\%$, the energy savings and performance delay increase with decreasing v-f settings for all kinds of tasks. Table 2.11 displays the *%energy* and *%delay* for the benchmarks at this setting. Comparing these results with Table 2.10(a), we can see that for high α, the energy savings for all

Table 2.11: Energy Savings/Performance Delay at 208MHz/1.2V.

Benchmark	%delay	%energy
qsort	56	48
dgzip	34	54
djpeg	33	54
bf	40	51
burn_loop	150	10

the benchmarks is on an average within 8% of the maximum possible at much lower overhead. For instance, for *qsort*, with the controller at high α, we get 41% energy savings at delay of 25% compared to 48% savings and 56% delay at 208MHz/1.2V. Thus, the controller almost gets the same energy savings at much lower performance delay because it runs the highly CPU intensive phase of *qsort* (Figure 2.9) at 416MHz (as discussed in the preceding paragraph). For *burn_loop*, the controller is able to identify the high CPU-intensiveness, and hence runs at 416MHz/1.4V even for high α to achieve within 5% of maximum energy savings at significantly lower performance overhead.

Multi Task Environment We next experimented with the benchmarks in a multi-tasking environment to verify our per task characterization by spawning 2 threads running different benchmarks simultaneously. Table 2.10(b) presents the results we achieved for multitasking for different values of α. For *djpeg+dgzip* the results are roughly an average of the individual results in Table 2.10(a). This happens because the duration of execution of both tasks is equal. An average value of the delay and savings indicate that both the tasks run with the same expert selection over their execution time-frame as in the single task case across all the α settings. This shows that per task characterization of the controller is accurately preserved across context switches. For *qsort+djpeg*, the results for all the values of α correspond very closely to the results of *qsort* in Table 2.10(a). Since the total time of execution of *qsort* benchmark is roughly 4 times the duration of *djpeg* benchmark, the results converge to that of individual *qsort* benchmark results. However, the *djpeg* benchmark runs exactly twice longer with *qsort* than alone. This shows that accurate preservation of per task characteristics enables the controller to select the same set of experts for *djpeg* as it does when *djpeg* runs

alone in the system, hence keeping its effective run-time the same. We observed similar results for the *qsort+dgzip* combination.

Task Characteristics and Leakage Awareness   Since the controller incorporates task characteristics and leakage awareness, it knows when DVFS is beneficial for a task from overall system energy efficiency and performance point of view. This awareness becomes more critical for future generation processors with higher leakage, where running at lower frequency could result in higher energy consumption (see *burn_loop* results in Figure 2.2d). To verify how controller would adapt to such platforms, we derived a new μ-mapper for our v-f setting experts based on Figure 2.3b to simulate a platform with ρ=50%. We then ran the benchmarks under different α settings. For *burn_loop*, we observed that the controller now selected the 1.5V/520MHz expert consistently for the high α settings, since the new μ-mapper now incorporates the knowledge of high leakage power consumption of the CPU. This is in contrast to the results with our old μ-mapper ($\rho \approx$30%), where it selected 1.4V/416MHz expert for high α. We observed similar results for other benchmarks as well. By incorporating CPU leakage characteristics, controller is also able to balance DVFS and DPM for better overall energy efficiency. This makes the controller scalable and adaptable across CPUs with varying leakage characteristics.

2.4.3   Overhead

For HDD, the controller causes overhead in terms of both energy and time to perform the evaluation of experts. In our experiments, we measured the average controller overhead at 0.0001% of the total timeframe for HDD, which is negligible relative to the overall timeframe. For CPU the controller adds overhead to the system, since it processes the 3 events delivered to it by the Linux process manager as discussed in Figure 2.7. We used *lmbench* [45] for measuring the overhead caused by the LKM. The *lat_proc* and *lat_ctx* tests measure the overhead added to process creation and context switch times. For *lat_proc* the overhead was 0%, while for *lat_ctx* it was around 3%. The overhead is negligible because the event processing

functions in the controller are extremely fast and lightweight. The controller itself is implemented in fixed point arithmetic and is very lightweight. The use of weights obviates the need for storing μ estimates thereby avoiding a potential overhead. For instance in [19] a regression based approach is used for workload characterization, which maintains a queue of 25 most recent estimates of CPI_{avg} and MPI_{avg} samples. In tests with lmbench, they increase the context switch time by a factor of 2, while our overhead is negligible.

In terms of memory overhead, the controller adds an array of unsigned long long variables, whose size is equal to the number of experts in the working set, to the *task_struct* of Linux. The LKM uses unsigned long long variables to simulate decimal values for weight/probability factors since we do not use floating point inside the kernel. Thus, overall the overhead of controller on a running system is negligible, which makes its deployment in real systems practical.

2.5 Conclusion

In this chapter we presented a novel online learning algorithm that solves both dynamic power management (DPM) and dynamic voltage frequency scaling (DVFS) problems. We presented a formulation of both DPM and DVFS as one of workload characterization and expert selection, and used the algorithm to solve it. The advantage of using an online learning algorithm is that it provides a theoretical guarantee on convergence to the best performing expert. The general formulation of the controller makes it applicable to any system component with support for power management. We performed experiments on two different HDDs and an Intel PXA27x CPU under varying real life workloads in both single task and multi-task scenarios. Our results indicate that our algorithm adapts really well to changing workload characteristics and achieves an overall performance comparable to the best performing expert at any point in time. Moreover, the algorithm incorporates leakage awareness, which allows it to adapt seamlessly to changing CPU leakage characteristics and also understand the tradeoff between DPM and DVFS. It is extremely lightweight and has negligible overhead in terms of performance and

energy.

Chapter 2, in part, is a reprint of the material as it appears in Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, 2006. Dhiman, G. and Rosing, T.S. The dissertation author was the primary investigator and author of this paper.

Chapter 2, in part, is a reprint of the material as it appears in Proceedings of the 12th ACM/IEEE International Symposium on Low Power Electronics and Design, 2007. Dhiman, G. and Rosing, T.S. The dissertation author was the primary investigator and author of this paper.

Chapter 2, in part, is a reprint of the material as it appears in IEEE Transactions in Computer Aided Design of Integrated Circuits and Systems, 2009. Dhiman, G. and Rosing, T.S. The dissertation author was the primary investigator and author of this paper.

Chapter 3

Analysis of Energy Efficiency in Server Systems

3.1 Introduction

Active power management is an extremely attractive proposition for mobile systems to achieve energy savings. As discussed in chapter 1, the primary reason for that is the usage pattern of mobile systems, which is bimodal in nature – high levels of activity intersperse long periods of inactivity [90, 42], which allows use of aggressive system level active power management.

In this chapter we evaluate the effectiveness of DVFS and DPM in terms of possible energy performance tradeoffs for server class systems. We show experimentally and through analysis that the potential for energy savings with DVFS on such systems has significantly diminished in newer CPU technologies due to reasons like faster memory interface system, more efficient support for low power modes in CPUs and higher relative power consumption of components other than CPU (e.g. memory). In fact, simple DPM policies provide better system energy savings/performance tradeoffs across a wide range of workloads than DVFS. This is in sharp contrast to what we observed for a mobile system CPU in the previous chapter, which benefited a great degree from DVFS. We further show that due to a smaller contribution of CPU in power consumption of servers overall benefits of

aggressive CPU power management in terms of DPM and DVFS have diminished.

We then identify possible ways to achieve better energy efficiency for such systems, namely (1) developing more energy proportional technologies and architectures. We show this by taking an example of memory hierarchy; (2) maximizing the performance per watt of server clusters through energy aware workload consolidation. These ideas form the baseline for the next three chapters.
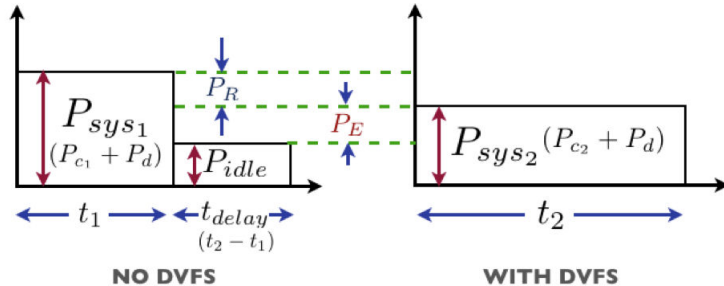
Figure 3.1: Comparison of Power Consumption and Execution Times of a workload with and without DVFS.

3.2   DVFS for System Level Energy Savings

Figure 3.1 illustrates system power consumption with and without DVFS for a workload. Without DVFS the CPU executes the workload at the highest frequency for time t_1 and the system consumes power P_{sys_1}. When not executing anything useful, the system/CPU is idle and consumes power P_{idle}. With DVFS the power consumption reduces to P_{sys_2}, while the execution time increases by t_{delay} from t_1 to $t_1 + t_{delay}$ (shown as t_2) because of CPU operation at a lower v-f setting. The decrease in power consumption depends on the degree of reduction in voltage and frequency, while the increase in execution time of the workload (t_{delay}) is a function of how it utilizes the CPU resources [56, 29].

We can further break down the system power consumption into that of CPU (P_c) and the other devices in the system (P_d). If we let the power consumption of CPU at v-f settings 1 and 2 by P_{c_1} and P_{c_2} respectively, and $P_{c_{idle}}$, when it is idle

(as shown in Figure 3.1), we can represent the energy savings due to DVFS as:

$$
\begin{aligned}
E_{DVFS} &= P_{sys_1}t_1 + P_{idle}t_{delay} - P_{sys_2}(t_1 + t_{delay}) \\
&= (P_{sys_1} - P_{sys_2})t_1 - (P_{sys_2} - P_{idle})t_{delay} \\
&= (P_{c_1} - P_{c_2})t_1 - (P_{c_2} + P_d - P_{c_{idle}} - P_{d_{idle}})t_{delay} \\
&= (P_{c_1} - P_{c_2})t_1 - ((P_{c_2} - P_{c_{idle}}) + (P_d - P_{d_{idle}}))t_{delay} \\
&= P_R t_1 - P_E t_{delay} \\
&= E_R - E_E
\end{aligned}
\tag{3.1}
$$

P_R/E_R is the reduction in CPU power/energy consumption because of DVFS. The second term $(P_E t_{delay}/E_E)$ represents the extra energy consumption that DVFS causes relative to the case without DVFS. There are two sources of extra power consumption for a system with DVFS: 1) The difference between CPU power consumption at the lower v-f setting and the idle CPU $(P_{c_2} - P_{c_{idle}})$, 2) The difference between device power consumption when it is active and idle $(P_d - P_{d_{idle}})$. The extra device power consumption depends on how often the executing workload accesses the devices. For instance, for a memory bound workload, the difference would be high, since it would make the memory consume more power for the extra time t_{delay} compared to an idle system. In contrast, for a CPU bound workload, the difference would be negligible. The performance delay (t_{delay}) determines for how long the DVFS based system consumes this additional power, and hence the extra energy consumption (E_E).

Clearly, the DVFS provides energy savings only as long as $E_R > E_E$. When this inequality does not hold, the system incurs performance overhead (t_{delay}) and consumes more energy than when running at the highest CPU frequency. We next show how the performance delay, low overhead of entry into sleep states and the energy impact of other system components affects the efficiency of DVFS.

Performance Delay (t_{delay}) As shown in the previous chapter, memory bound workload incurs lower performance hit at a lower frequency setting, since it causes many CPU stalls due to memory accesses. For an ideal stall intensive workload, the delay in execution is zero, and hence represents the best case for DVFS. In

contrast, the execution time of a CPU intensive workload is entirely determined by the CPU frequency. For an ideal CPU intensive workload, the increase in execution time when switching from frequency f_1 to f_2 can be estimated as $(\frac{f_1}{f_2})$. Such a workload represents the worst case for DVFS, since it incurs the highest possible t_{delay}.

In SPECCPU 2000 suite, *mcf* is a memory bound benchmark, while *sixtrack* is CPU intensive [15]. To understand the correlation between execution time delay and workload characteristics we ran the benchmarks at different frequency settings on a state of the art quad core AMD Opteron based system. Table 3.2 shows the *%delay* incurred by the benchmarks at the four settings supported by the processor. We also plot the *%delay* for an ideal CPU intensive workload, which we label as the "worst" case, and for the ideal stall intensive workload, which we label as the "best" case. *Sixtrack* incurs a delay that is identical to that of the worst case due to its high CPU intensiveness. In contrast, for *mcf* it is relatively lower.
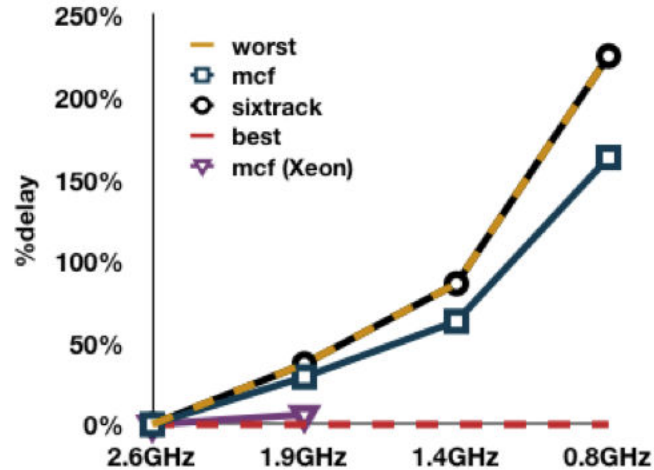
Figure 3.2: Analysis of Performance Delay (%delay) for *mcf* and *sixtrack* workloads at lower frequency settings.

However, this delay is significantly larger than the best case. This delay is much smaller on processors with smaller caches and slower memory controllers. The AMD processor we use has three levels of caches (L3 cache of 6MB), an on-die memory controller that operates at 2.6GHz and sophisticated cache prefetching

mechanisms. Consequently, the memory bound phases of *mcf* are much shorter on this processor compared to running it on a processor with slower memory accesses. For instance, running *mcf* on an older Intel quad core Xeon (with two frequency settings: 2.6/1.9GHz, no L3 cache and an off chip memory controller), the delay of *mcf* at 1.9GHz is just 6% from the best case (see Figure 3.2). This is much lower when compared to 30% delay on the Opteron at the same frequency. Faster memory controllers and larger caches successfully mask the memory latency making the execution more dependent on the CPU frequency, thus limiting the possible energy savings due to DVFS. The trend towards faster and efficient on die memory controllers has also been adopted in the more recent Intel microarchitectures as well [48].

Lower idle CPU power consumption   The support for ACPI C-states or CPU low power states/modes has evolved significantly over the last few years. The C1 state, where the clock supply to the CPU is gated, is so efficient in modern processors, that it is currently used by default in all major operating systems (eg. Linux, OpenSolaris) when CPU is idle. Recently announced Intel and AMD processors have also added support for deeper C states which can significantly reduce the power consumption of CPU during idle times [47]. For instance, C6 power state can reduce the power consumption to zero. The performance overhead of such states is in the order of just μ-seconds, thus allowing their frequent use. The previous generation processors did not have such C-state support, and thus consumed higher power during the idle periods. This increases the difference between the CPU power consumption at lower v-f setting and idle time, or in terms of equation 3.1, increases $(P_{c_2} - P_{c_{idle}})$, and hence E_E (see Figure 3.1).

Power consumption of other system components   An important aspect that is often ignored by research in DVFS is its impact on system level energy savings. This is important to consider for server systems, since the power contribution of other components is also significant. DVFS can make other system components consume more power for a longer duration due to operation at lower frequency. For instance, our Opteron system is equipped with 4GB DDR3 RAM, which ap-

Table 3.1: Power Management Policies.

Policy name	Description
PM-1	switch CPU to ACPI state C1 (remove clock supply) and move to lowest voltage setting
PM-2	switch CPU to ACPI state C6 (remove power)

proximately consumes 4.5W when it is idle/not being accessed. However, when a memory intensive benchmark (like mcf) is running, the memory consumption increases by 5W, i.e. it more than doubles. Thus, running mcf at a lower frequency makes the memory consume extra 5W for t_{delay} (see Figure 3.1), compared to a system without DVFS. This means a higher value of $(P_d - P_{d_{idle}})$, or higher E_E (see Figure 3.1).

3.3 Evaluation Setup and Results

To evaluate the effectiveness of DVFS for system level energy savings we formulate a simple static DVFS policy (s-DVFS), where workloads are executed statically at different v-f settings. This is sub-optimal, since one can potentially get better results by running a workload at different settings based on its phase of execution. However, as we show later, it does give a fair idea of the possible savings in the best case for most of the benchmarks. We also propose three simple power management policies that are based on running the workload at the highest speed and then reducing the power during the idle periods (P_{idle} in Figure 3.1) through different mechanisms. These policies are listed in Table 3.1. Each policy is successively more aggressive than the previous one in terms of reducing P_{idle}. PM-1 is extremely easy to implement as support for C1 states is widely available in current processors. PM-2 relies on efficient C6 state support, which has recently been introduced in Intel and AMD processors.

For our experiments, we instrument a quad core AMD Opteron processor based system running OpenSolaris. The processor supports four v-f settings: 1.25V/2.6GHz, 1.15V/1.9GHz, 1.05V/1.4GHz and 0.9V/0.8GHz. For workloads, we use integer and floating point benchmarks from the SPECCPU 2000 suite. For comparison of s-DVFS with the other PM policies in Table 3.1, we use the model developed in section 3.2 (see equation 3.1). For instance, to compare s-DVFS and

PM-1, we run the benchmarks at all the frequencies and measure the corresponding execution time and system power. In terms of Figure 3.1, we get t_1 and P_{sys_1}, i.e. time and power for the highest frequency (we use that for PM-1) and t_2 and P_{sys_2} for the lower frequencies(we use that for s-DFVS). We measure power at the power outlet of the system using a data acquisition system (DAQ), which collects power samples every 300ms. These measurements allow us to estimate P_R, P_E, t_1 and t_{delay} (see Figure 3.1), and hence E_{DVFS} based on equation 3.1 for each frequency 'f'. In other words E_{DVFS} gives the energy savings of s-DVFS over PM-1. Based on this, we estimate the %energy savings of s-DVFS over PM-1 at given frequency 'f' as:

$$\%E_{savings_{PM-i}} = \frac{E_{DVFS_f} - E_{PM-i}}{E_{PM-i}}$$

where i = 1,2 \hfill (3.2)

E_{PM-i} varies based on the policy we are comparing s-DVFS against, since each has different idle system power consumption or P_{idle} (see Table 3.1). We measure the CPU and memory power consumption separately to estimate P_{idle} for PM-(1-2).

Results: Table 3.2 shows the comparison of energy and performance results achieved for s-DVFS and policies PM-(1-2) across 16 SPEC benchmarks. The "frequency" column indicates the frequency (in GHz) at which the processor is set for s-DVFS policy. The *%delay* indicates the percentage by which the benchmark's execution time increases because of s-DVFS when compared against policies PM-(1-2). The *%energy* savings ($\%E_{savings_{PM-i}}$) column indicates the system level energy savings achieved by the s-DVFS policy compared to the PM-(1-2) based on equation 3.2. Positive savings indicate that s-DVFS at the given frequency is more energy efficient than the corresponding PM policy and vice versa.

We can observe from Table 3.2 that across all the benchmarks the performance delay because of s-DVFS is large. For 9 benchmarks (*bzip2, eon, gcc, crafty, gzip, parser, sixtrack, mesa, ammp*) the performance delay is within 5% of the worst case (refer to Figure 3.2) due to their high CPU intensiveness. For 5 benchmarks (*art, mgrid, twolf, swim, applu*), it is within 15% of the worst case,

Table 3.2: Comparison of s-DVFS and PM 1-2.

Workload	freq	%delay	%E_savings PM-1	%E_savings PM-2	Workload	freq	%delay	%E_savings PM-1	%E_savings PM-2
			PM-1	PM-2				PM-1	PM-2
mcf	1.9	29%	5.2%	0.7%	sixtrack	1.9	37.3%	5.0%	-0.5%
	1.4	63%	8.1%	0.1%		1.4	86.2%	6.0%	-4.3%
	0.8	163%	8.1%	-6.3%		0.8	226.4%	6.8%	-10.7%
bzip2	1.9	37.1%	4.7%	-0.6%	mesa	1.9	36.5%	2.9%	-2.5%
	1.4	85.7%	7.4%	-2.4%		1.4	84.8%	5.8%	-4.2%
	0.8	222.9%	7.8%	-9.0%		0.8	223.8%	7.2%	-9.9%
eon	1.9	33.3%	4.0%	-0.9%	lucas	1.9	29%	4.3%	0.2%
	1.4	81.0%	6.6%	-3.1%		1.4	63.2%	6.7%	-1.0%
	0.8	219.0%	7.1%	-9.9%		0.8	169.4%	6.1%	-8.5%
crafty	1.9	37.6%	4.7%	-0.6%	swim	1.9	31.8%	3.3%	-1.2%
	1.4	85.5%	7.5%	-2.3%		1.4	75.2%	3.9%	-5.0%
	0.8	222.4%	7.9%	-8.9%		0.8	198.4%	4.2%	-11.9%
gcc	1.9	34.7%	4.3%	-0.7%	art	1.9	32.4%	5.9%	1.2%
	1.4	81.3%	7.4%	-2.0%		1.4	76.1%	7.3%	-1.7%
	0.8	214.2%	7.9%	-8.6%		0.8	202.4%	8.0%	-8.0%
gzip	1.9	36.6%	6.4%	1.3%	mgrid	1.9	31.1%	2.9%	-1.6%
	1.4	85.2%	8.4%	-1.2%		1.4	79.2%	4.1%	-5.3%
	0.8	224.7%	7.9%	-8.9%		0.8	208.5%	4.3%	-12.3%
parser	1.9	35.8%	4.4%	-0.9%	ammp	1.9	35.6%	5.0%	-0.2%
	1.4	82.1%	7.0%	-2.7%		1.4	83.3%	6.6%	-3.3%
	0.8	214.8%	8.0%	-8.6%		0.8	218.8%	2.0%	-16.0%
twolf	1.9	35.1%	4.5%	-0.8%	applu	1.9	32.5%	2.7%	-2.1%
	1.4	80.5%	6.8%	-2.8%		1.4	73.9%	4.7%	-4.4%
	0.8	211.4%	7.1%	-9.6%		0.8	193.9%	5.7%	-10.3%

(a) SPEC 2000 INT (b) SPEC 2000 FP

which means they comprise of some phases of execution, which are memory bound. Only for *mcf* and *lucas* it is more than 15%. Thus, in terms of performance all the benchmarks except *mcf* and *lucas* take a severe hit because of DVFS. The primary reason for the high delay is the sophisticated memory subsystem of the CPU we use (refer to section 3.2).

The *%energy* savings results indicate that s-DVFS is also not very efficient from the perspective of energy savings. Compared to PM-1, it achieves on an average a maximum of just 7% energy savings across all the benchmarks, which comes at the cost of around 208% increase in execution time. We observe two interesting trends in these results: (1) *Energy savings has little correlation to benchmark characteristics*: This happens due to *%delay* being uniformly high for most of the

benchmarks. Intuitively, *mcf* and *lucas* should incur higher savings due their lower t_{delay}. However, as identified in section 3.2, the memory bound workloads cause the memory to consume extra energy when compared to CPU bound workloads. This offsets their higher CPU energy savings. (2) *There is not much gained by running benchmarks at lower v-f settings*: Across all the benchmarks, the average gain in energy savings for s-DVFS by switching from 1.9GHz to 0.8GHz is just 2%. For the same switch the increase in performance delay is around 180%. This indicates that the higher power savings at lower v-f setting do not translate into higher energy savings due to higher performance delay at that setting.

In comparison to PM 2, the results of s-DVFS are worse, as for majority of the benchmarks, s-DVFS is actually energy inefficient. The reason for this follows from our analysis on idle CPU power consumption in section 3.2, which is now reduced to zero. Consequently, based on equation 3.1, E_E becomes greater than E_R.

3.4   Conclusion
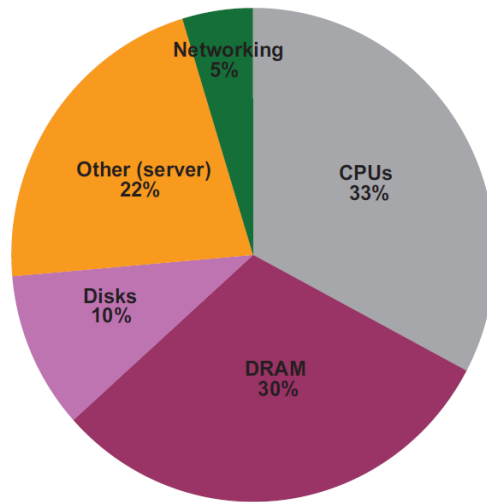
Figure 3.3: Power Consumption breakdown for a typical modern server [42].

The results clearly indicate that benefits of DVFS from system level energy savings viewpoint has diminished significantly. Simple DPM policies achieve better system level energy performance tradeoffs in majority of the cases. This

suggests that active power management in server class systems is much simpler – just use DPM across all system components. However, the scope of achieving energy savings through DPM in such systems is also limited due to the following reasons: (1) The usage model for servers has very different characteristics from that of mobile systems. As shown in Figure 1.1a, servers are rarely completely idle and seldom operate near their maximum utilization. Instead, servers operate most of the time at between 10 and 50% of their maximum utilization levels. This means servers cannot take advantage of system level DPM policies. (2) In terms of power consumption, close to 65% power consumed by server systems is contributed by non energy proportional components like fans, power supplies, memory etc. as shown in Figure 3.3. This property makes these systems highly energy inefficient at lower utilization levels, since their power consumption does not come down in proportion to their utilization due to poor support for DPM. For instance, memory supports self-refresh mode where the power consumption goes down by 80%, but it cannot be accessed at all in that mode and the break even time (defined in section 1.1.1) is large. This makes its usage for server systems difficult since memory and system are rarely completely idle, as discussed above.

Thus, for the server class systems, there are two ways of achieving energy efficiency: (1) Design systems to be more energy proportional. (2) Push the existing systems towards more energy efficient zone of operation by increasing their utilization and maximizing the overall performance per watt.

In the next chapter, we focus on the first approach by taking memory hierarchy as an example, since in modern servers memory is as big a consumer of power as the CPUs (see Figure 3.3). The second approach is the focus of the following two chapters.

Chapter 3, in part, is a reprint of the material as it appears in Proceedings of the Workshop on Power Aware Computing and Systems, 2008. Dhiman, G.; Pusukuri, K.K. and Rosing, T. S. The dissertation author was the primary investigator and author of this paper.

Chapter 4

Energy Efficient Memory Hierarchy

4.1 Introduction

In this chapter, we introduce a new heterogeneous organization for main memory that is composed of conventional DRAM and phase change memory (PRAM). This configuration helps reduce the energy consumption of memory subsystem with minimal impact on workload performance by exploiting benefits of both the technologies. The properties of PRAM that we leverage are its lower read access and standby power compared to DRAM while having a comparable throughput. However, the primary challenges in using PRAM include its lower write endurance (typical mean time to failure in the range of $10^9 - 10^{12}$ cycles), and the higher power cost of write accesses compared to DRAM. These properties motivate the use of a heterogeneous memory architecture consisting of both DRAM and PRAM, which we refer to as PDRAM, enabling exploitation of positive aspects of the respective memories.

We propose a hybrid hardware/software solution to manage the PDRAM memory organization. In order to maintain reliability for PRAM (because of write endurance problem), we introduce cost efficient book keeping hardware technique that stores the frequency of writes to PRAM at a page level granularity. We com-

plement hardware solution with an efficient operating system (OS) level page manager that utilizes the write frequency information provided by the hardware to perform uniform wear leveling across all the PRAM pages. Wear leveling refers to the process of prolonging the lifetime of erasable storage devices with endurance problems (like Flash, PRAM etc.) by ensuring uniform usage/utilization of all the storage blocks/pages of the device. The page manager intelligently allocates/migrates pages across DRAM/PRAM in order to minimize the impact of wear leveling on performance. The benefits of using such a hybrid approach is that the hardware can maintain and track page level accesses at a very low cost, while the software (OS) can leverage the high level observability and policies for management of free pages across DRAM/PRAM.

Existing memory power management research has primarily focused on DRAM based systems. In [62], the authors propose power aware page allocation algorithms for DRAM power management. They assume support in memory controller for fine grained bank level power control and show that their allocation algorithms give greater opportunities for placing memory in low power modes. In [26, 52], the authors propose an OS level approach, where the OS maintains tables that map processes onto the memory banks they have their memory allocated in. This allows the OS to dynamically move unutilized DRAM banks into low power modes.

The possible use of alternative memory technology for improving energy efficiency has also been explored before. The authors in [60] propose NAND flash based page cache, which reduces the amount of DRAM required for system memory. This results in energy efficiency due to lower power consumption and higher density of NAND flash compared to DRAM. However, this approach is beneficial for more disk intensive applications, since flash is used only for page cache. In addition, flash also has endurance problems in terms of number of write cycles, which is tackled using wear leveling in the flash translation layer [65].

PRAM is an attractive alternative to flash, since: (1) Its endurance is higher by several orders of magnitude, (2) It is a RAM and hence does not require an overhead of an erase before a write. In [63], the authors evaluate the challenges
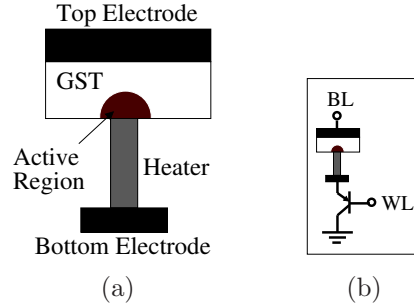
Figure 4.1: Illustration of PRAM cell (a) and transistor (b).

involved in architecting PRAM as a DRAM alternative. The authors in [67] propose a hybrid cache, that is composed of PRAM and SRAM for power savings. To solve the endurance problem of PRAM, they set a threshold on number of writes to the PRAM cache lines, beyond which they do not use those lines. However, the concern is that the typical level of conflicts and activity in cache could create reliability problems very soon on such a configuration. Extensive research has been done in modeling and understanding the basic characteristics of this technology [94, 61, 105, 82, 51].

4.2   Design

4.2.1   PRAM/DRAM Background

The DRAM memory is organized as a grid of rows and columns, where each bit is stored in the form of charge in a small capacitor. As the charge gets exhausted due to leakage and frequent accesses, DRAM requires a consistent refresh operation to sustain its data. This results in a constant power consumption referred to as the *refresh* power. DRAM further consumes power for setting up its row and column for a physical address accessed, and also for closing a row if some other row needs to be accessed. This is referred to as the *activation/precharge* power. Additionally, it consumes power for the actual read/write accesses, and consistent standby power due to leakage and clock supply.

Unlike DRAM, PRAM is designed to retain its data even when the power is turned off. A PRAM cell stores information permanently in the form of the cell

material state, which can be amorphous (low electrical conductivity) or crystalline (high electrical conductivity). A PRAM cell typically consists of chalcogenide alloy material (eg. $Ge_2Sb_2Te_5$ (GST)) and a small heater as shown in Figure 4.1a. The cell can be addressed using a selection transistor (MOS or BJT) that is connected to the word-lines (WL) and bit-lines (BL) as illustrated in Figure 4.1b. To write to a PRAM cell, the GST state needs to be altered by injecting a large but fast current pulse (few 100ns) to heat up the GST active region. Consequently, PRAM write power is high compared to DRAM. For reading a PRAM cell, the power consumption is much lower, since no heating is involved. It is also lower than DRAM cell read power based on the measurements shown in [82]. PRAM consumes no refresh power, as it retains its information permanently and consumes much lower standby power due to its negligible leakage [82]. However, PRAM has limited write endurance ($10^9 - 10^{12}$ cycles), which poses a reliability problem. Regarding access times, the access latency of random reads/writes on PRAM is slower compared to DRAM, although their read throughput is comparable. Thus, considering all of these factors, PRAM is a promising candidate for energy savings because of its low read and standby power compared to DRAM.

4.2.2   Architecture

Both PRAM and DRAM technologies have their respective advantages and disadvantages. This motivates us to propose a hybrid memory architecture, which consists of both DRAM and PRAM (PDRAM) for achieving higher energy efficiency. While PRAM provides low read and standby power, DRAM provides higher write endurance and lower write power.

The primary design challenge in managing a PDRAM system is to manage efficient wear leveling of PRAM pages to ensure its longer lifetime. For this purpose, we provide a hybrid hardware-software based solution. The hardware portion is based in the memory controller and manages the access information to different PRAM pages. The software portion is part of the operating system (OS) memory manager (referred to as the page manager), which performs wear leveling by page swapping/migration. The components of the solution are described in

detail below.

Memory controller   Figure 4.2 illustrates the various components and interactions of the PDRAM memory controller. The memory controller is aware of the partitioning of system memory between DRAM and PRAM. Based on the address being accessed, it is able to route requests to the required memory. To help wear leveling the PRAM, it maintains a map (access map in Figure 4.2) of the number of write accesses to it. This information is kept at a page level granularity, which is a function of the processor being used. For instance, the page size is 4KB for x86, 8KB for Alpha etc. We use page level granularity, since it is the unit of memory management, i.e. allocation and deallocation in the OS. If the number of writes to any PRAM page exceed a given threshold, then the controller generates a 'page swap' interrupt to the processor, and provides the page address. The OS then assumes the responsibility of handling this interrupt and performing page swapping as described in discussion below. The controller stores the map in the PRAM, for which it reserves space during bootup. The access map is maintained for the lifetime of the system, and after the first page swap interrupt, future interrupts are generated whenever the write access count becomes a multiple of the threshold. To maintain the map across reboots, it is stored on disk before the shutdown, and copied back into PRAM during the startup. To protect this data against crashes, it is synced with the disk periodically. If the write count for a page reaches the endurance limit (10^9 in the worst case), the controller generates a 'bad-page' interrupt for that page. This interrupt is also handled by the page manager as described in the discussion below.

The enhancement of the controller incurs energy and memory overhead: 1) time and energy for updating PRAM access map; this involves extra accesses to PRAM, which causes extra power consumption. To avoid this overhead we introduce a small SRAM based cache in the controller (see Figure 4.2), which caches the updates to the map, hence reducing the consequent number of PRAM accesses. 2) memory overhead for storing the access map; the amount of memory required is proportional to the size of the PRAM used, and the size of the entry stored for each physical page in the access map. For instance, if the PRAM used is
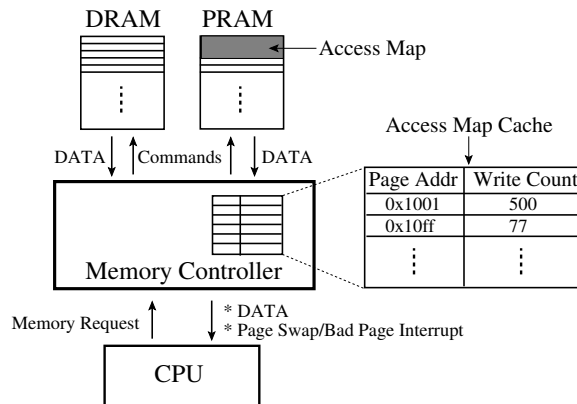
Figure 4.2: PDRAM Memory Controller.

4GB in size and the page size is 8K, the memory required for the access map would be around 4MB (each entry = 8 bytes). This is fairly small for modern systems, which have multiple GBs of memory.

Page Manager   The page manager is the OS level entity responsible for managing memory pages across PRAM/DRAM. It consists of two key subsystems, described below in detail, which help it perform its key tasks: Memory Allocator (page allocation/deallocation) and Page Swapper (uniform wear leveling of PRAM).

Memory Allocator   The goal of the OS memory allocator is to serve the memory allocation requests from the OS and the user processes. A list of free pages is maintained by typical allocators from which the page request is fulfilled. Traditional memory allocators instantly mark the pages released by applications as free, and may immediately allocate them on subsequent requests. Such an approach can create hot spots of pages in memory, where the activity (reads/writes) is significantly higher compared to rest of the pages. This is fine for memories (like DRAM) that do not face endurance problems, but for memories like PRAM, it is a problem, as it may render some pages unusable very soon. To get around it, we make the PRAM memory allocator aware of these issues. The PRAM allocator maintains three lists of free pages: *free*, *used-free* and *threshold-free* list. At the startup time, all PRAM pages are in the *free* list, and the allocation requests from

the applications are served from it. When the pages are freed, they go to the *used-free* list rather than the free list. If an allocated page is written to a lot by an application, and if the number of writes crosses a *'threshold'*, then as described before, the memory controller generates a page swap interrupt for that page. At this point, the page swapper (described below) handles the interrupt, and releases this page, which goes to the *threshold-free* list.

When the free-list becomes exhausted, or it lacks sufficient pages to service a request, it is merged with the used-free list, which is then used as the source of page allocation. The user-free list will get exhausted only when all the free PRAM pages have been written to at least a *'threshold'* number of times. When this happens, the *free* and *threshold-free* list pointers are swapped to move all the free pages from the *threshold-free* list to the original *free* list. Such an approach tries to achieve wear leveling across all the PRAM pages by ensuring that all the free pages have been written to at least *'threshold'* times before getting reused. We assume that there is a separate allocator for DRAM memory, which does not need any such changes.

Page Swapper: The page swapper is responsible for managing the page swap and bad-page interrupts generated by the memory controller. It handles the page-swap interrupt by doing the following: 1) Allocates a new page from the memory allocator. The new page could be either from the PRAM or DRAM. 2) Finds the page table entry/entries (PTE/PTEs) of the physical page for which the interrupt is generated. This can be accomplished in modern systems such as Linux using the reverse mapping (RMAP), which maintains a linked list containing pointers to the page table entries (PTEs) of every process currently mapping a given physical page. 3) Copies the contents of the old page to the new one using their virtual addresses. The advantage of using virtual address for the copy as opposed to something like DMA is that it results in coherent copying of data, which ensures that the new page gets the latest data. 4) Updates the PTE/PTEs derived from the RMAP with the new physical page address. 5) Replaces the TLB entries corresponding to the old PTE/PTEs. 6) Releases the old physical page. If it is a PRAM page, it goes to the threshold-free list as described above.

The key decision for the swapper is in selection of the new page for replacing the old PRAM page, for which interrupt is generated. We implement two policies for this decision:

Uniform memory policy: This policy allocates the new page from the PRAM allocator. We introduce this as our baseline policy, which can be used in a PRAM only memory configuration as well, since it allocates only PRAM pages. This policy exploits the wear leveling mechanism of page swapping to extend PRAM endurance, but does not benefit from the memory heterogeneity of a PDRAM system.

Hybrid memory policy: This policy allocates new page from the DRAM allocator. The motivation to do so is based on the fact that there is a high probability of the page, for which page swap interrupt got generated, being very write intensive. As we show later, this has a two fold advantage: (a) It reduces number of page swap interrupts, which is good for performance; (b) It reduces number of writes in PRAM, which is good from perspective of both reliability and power, since PRAM writes consume higher power than DRAM writes. This policy exploits the heterogeneity of the PDRAM system, hence the name hybrid memory policy.

For the bad-page interrupt, the page swapper does exactly the same things as it does for the page-swap interrupt, except that it moves the page off its free lists and moves it into a bad-page list. The pages in the bad-page list are discarded and not used for future allocations. The list is stored reliably in a known location on PRAM.

4.2.3   Endurance Analysis

In this discussion, we analyze PRAM reliability with and without our wear leveling policies. Lets assume a system with pages of size 4K (eg. x86 systems) and 4GB of PRAM, implying there are N_p=1M pages available for allocation. We assume an application, which consistently writes to two different addresses in PRAM, that map to different PRAM rows in the same memory bank. This ensures that each row is consistently written back to the PRAM cells with alternative writes, because at a time only one row in a bank can be open. Typical latencies

for such writes in PRAM is around 150ns (T_w). We assume that this application writes to these two addresses every 150ns or T_w in an alternative fashion to generate the worst case from endurance perspective. Note that this is unrealistic, since there are caches and write buffers between the CPU and memory, which will throttle the rate of writes. However, the analysis will allow us to estimate PRAM reliability under extreme cases. Lets refer to the write endurance of PRAM (10^9 write cycles in worst case) as N_w. If we do not take any wear leveling into account, then such an application can cause row failure in the page containing these addresses in $2N_w$ writes or $T_w \times 2N_w = 300s$. This is a very low time scale, and hence not acceptable for a real system deployment.

Now, we analyze the case with our uniform memory wear leveling policy. Let the threshold of writes, at which the memory controller generates a page swap interrupt, to be N_t (where $N_t \ll N_w$), and the % of free PRAM pages in the system be $\alpha\%$. Lets assume, that the PRAM rows containing the two addresses being written to by the application map to the same physical page. Now as soon as the application writes N_t times, the page swapper will swap the physical page mapping these two addresses to a new PRAM page, which will correspond to different rows in the PRAM. The old physical page will then be moved into the *threshold-free* list. From the previous discussion, we know that the application will not be able to write to the freed physical page (and its corresponding PRAM rows) again until both the *free* and *used-free* lists become empty and the *threshold-free* and *free* list pointers are swapped. For this to happen, the application will have to write at least N_t times to every free PRAM page ($\alpha N_p N_t$ writes) before it can access the old physical page again. In other words, to write $2N_t$ times to the old physical page, the application will do $\alpha N_p N_t + 2N_t$ writes, i.e.:

$$2N_t \rightarrow \alpha N_p N_t + 2N_t$$
$$3N_t \rightarrow 2\alpha N_p N_t + 3N_t$$
$$\beta N_t \rightarrow (\beta - 1)\alpha N_p N_t + \beta N_t \qquad (4.1)$$

This means that for doing N_w writes ($\beta N_t = N_w$ in equation 4.1):

$$N_w \rightarrow (\frac{N_w}{N_t} - 1)\alpha N_p N_t + (\frac{N_w}{N_t})N_t$$

$$N_w \rightarrow \alpha N_w N_p + N_w \approx \alpha N_w N_p \quad (N_w \gg N_t \; ; \; \alpha N_w N_p \gg N_w) \qquad (4.2)$$

Based on this analysis, our application will have to do approximately $\alpha N_w N_p = \alpha 10^{15}$ writes in order to perform N_w writes to a given physical page. This means, to write N_w times to a PRAM row, it will have to perform $2N_w$ writes. Using T_w as 150ns (see the paragraph above) and $\alpha = 50\%$, this translates to around 2.4 years. Thus, with our wear leveling scheme the bounds go from order of seconds to years. It must be noted that this analysis has been done assuming an application which bypasses cache and write buffers to perform just writes. If we assume a conservative assumption of 50% cache hit for writes, the PRAM lifetime would increase to about 4.8 years. As the quality of PRAM is expected to increase to 10^{12} cycles and beyond, the bounds will be much higher. For the hybrid memory policy, the expected bounds would be even higher since the write intensive pages are moved to DRAM, where endurance is not an issue. In our experiments, we use N_t=1000 as a good trade-off between endurance and swapping cost.

4.3   Evaluation

4.3.1   Methodology

For our experimental evaluation we use the M5 architecture simulator [14]. M5 has a detailed DRAM based memory model, which we significantly enhance to model timing and power of a state of the art modern DDR3 SDRAM based on the data sheet of a Micron x8 1Gb DDR3 SDRAM running at 667MHz [50]. We implement a similar model for PRAM based on timing and power characteristics described in [94, 105, 82, 63]. We assume PRAM cells to be arranged in a grid of rows and columns just like DRAM. Such configuration of PRAM has been practically implemented and demonstrated by Intel [82].

The power parameters used for DRAM and PRAM are listed in Table 4.1. The DRAM parameters are based on 78nm technology [50]. The read-write power

Table 4.1: DRAM and PRAM Characteristics (1Gb memory chip).

Parameter	DRAM	PRAM
Power Characteristics		
Row read power	210 mW	78 mW
Row write power	195 mW	773 mW
Act Power	75 mW	25 mW
Standby Power	90 mW	45 mW
Refresh Power	4 mW	0 mW
Timing Characteristics		
Initial row read latency	15 ns	28 ns
Row write latency	22 ns	150 ns
Same row read/write latency	15 ns	15 ns

values for PRAM are obtained from the results in [63, 10, 11, 82]. For a fair comparison, the values are down scaled for 78nm technology based on the rules described in [81]. We can observe that read power of PRAM is around three times lower compared to DRAM, while write power is around four times more. This suggests that PRAM is not very attractive for write intensive applications both from the perspective of reliability as well as energy efficiency. The third parameter in Table 4.1 (Act) refers to the activation/precharge power, which is consumed in opening and closing a row in the memory array. It is higher for DRAM, since it has to refresh the row data before closing a row, which can be avoided in PRAM due to its non-volatility. The standby power, which the memory consumes when it is idle, is also lower for PRAM due to its negligible leakage power consumption. Finally, DRAM consumes refresh power for supplying sustained refresh cycles for it to retain its data. This is not required in PRAM, since it is non-volatile.

For timing, we use the Micron data sheet to get the detailed parameters for DRAM. For PRAM, we use the results in [10] to obtain the read-write latency values for 180nm technology. We scale down the read latency for 78nm, but maintain the same value for write as a conservative assumption, since the write latency is a function of the material property. Table 4.1 shows these values. We can observe, that for an initial read, PRAM requires almost twice the amount of time

Table 4.2: Benchmark Characteristics.

Benchmark	*rpi* (%)	*wpi* (%)	# *of pages*
applu	1.94	0.93	24435
bzip2	0.12	0.08	24600
facerec	0.6	0.5	2240
gcc	0.15	0.06	2781
sixtrack	0.01	0.008	7601

as compared to DRAM. This happens due to the higher row activation time of PRAM. Similarly, writing back or closing an open row in PRAM is around seven times more expensive than for DRAM. However, reads/writes on an open row have latency values similar to DRAM.

Besides this, we further extend M5 for incorporating the memory controller and page manager as described in section 4.2. We implement the access map cache in the memory controller (see section 4.2) as a 32 entry (each entry = 8 bytes) fully associative cache. For the page manager, we implement both the uniform and hybrid memory policies as described in section 4.2.

For our experiments, we assume a baseline system with 4GB of DDR3 SDRAM, which we refer to as DRAM. The 4GB memory consists of four 1GB ranks, where each rank is made up of eight x8 1Gb chips with characteristics described in Table 4.1. We evaluate it against two experimental systems: 1) **Hybrid system**: It comprises of 1GB DDR3 SDRAM and 3GB of PRAM, and employs *hybrid memory policy* for managing page swap requests. 2) **Uniform system**: It comprises of 4GB of PRAM, and employs *uniform memory policy* for managing page swap requests. The motivation of the comparison is to show how heterogeneity in memory organization can result in better overall performance and energy efficiency.

For workloads, we use benchmarks from the SPECCPU2000 suite, which we execute on M5 using a detailed out-of-order execution ALPHA processor running at 2.66GHz. The simulated processor has two levels of caches: 64KB of data and instruction L1 caches, and 4MB of L2 cache. We use benchmarks described in Table 4.2, and simulate the first five billion instructions. Table 4.2 illustrates
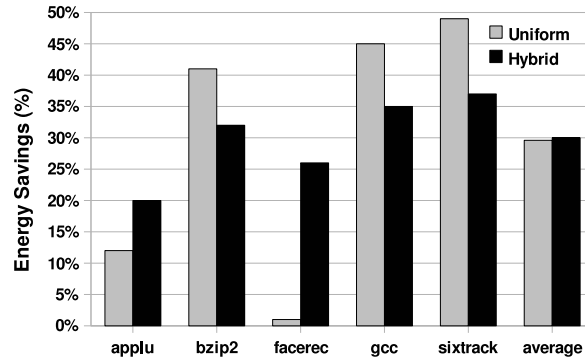
(a) Energy Savings
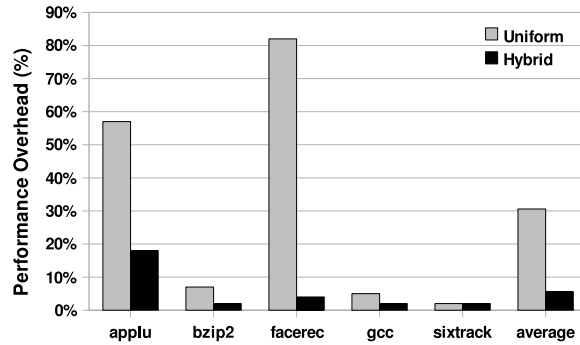
(b) Performance Overhead

Figure 4.3: Energy Savings and Performance Overhead Results for Uniform and Hybrid Policies.

the memory access characteristics of these benchmarks in terms of *rpi* (reads per instruction), *wpi* (writes per instruction), and *number of pages* (total number of pages allocated). We can see that they have varying memory access characteristics. For instance, *sixtrack* has very low *rpi* (0.01%) and *wpi* (0.008%), while for *applu* it is an order of magnitude higher (1.94% and 0.93%); *facerec* has high *rpi* (0.6%) and *wpi* (0.5%), while *gcc* and *bzip2* have medium *rpi* and low *wpi*. Similarly, the working set of these benchmarks in terms of the number of pages allocated also varies from just 2240 (*facerec*) to as high as 24600 (*bzip2*).

4.3.2 Results

Energy Savings and Performance Overhead   Figure 4.3 shows the results of the hybrid and uniform memory systems baselined against the DRAM system for

all the benchmarks. Figure 4.3b shows the overhead incurred in terms of execution time by these systems, while Figure 4.3a shows the reduction in memory energy consumption. Please note that the *%overhead* and *%energy* numbers in these figures include the energy and time overhead due to page migrations and accesses to the access map in the memory controller and the PRAM. We describe the details of the overhead in the following sections.

We can see in Figure 4.3, that on average, the hybrid system achieves around 30% energy savings for just 6% performance overhead across all the benchmarks. In contrast, the uniform system gets 30% energy savings at the cost of 31% overhead. For sixtrack, which has low *rpi* and *wpi*, the impact on performance is negligible since there are not enough accesses to expose the slower access times of PRAM. Both systems achieve high energy savings due to the lower standby power consumption of PRAM compared to DRAM (see Table 4.1). The energy savings for the uniform system is higher (around 49%) than hybrid system (37%) since the uniform system comprises exclusively of PRAM, while the hybrid system contains 1GB of DRAM as described in section 4.1. For *gcc* and *bzip2*, the performance overhead is higher for the uniform system (around 6%). This happens due to the relatively higher *rpi* and *wpi* of these benchmarks compared to *sixtrack* (see Table 4.2), which exposes the slower access times of the PRAM. The overhead is lower for the hybrid system (2%), since it migrates the write intensive pages to DRAM. The energy savings is higher again for the uniform system due to the lower standby power of PRAM.

In contrast, for *applu* and *facerec*, the performance overhead of the uniform system is significantly high (57% and 82% respectively). This happens due to the higher *wpi* and *rpi* of both these benchmarks (see Table 4.2). The overhead for *facerec* is higher than *applu* (despite its lower *rpi* and *wpi*) due to its higher IPC (60% more than *applu*). This implies, *facerec* is more sensitive to higher memory access latencies, which results in the poor performance of the uniform system due to slower access times of PRAM. The high overhead nullifies the lower power consumption of PRAM and results in negligible energy savings. In contrast, the performance overhead of the hybrid system is very low (4%). This happens
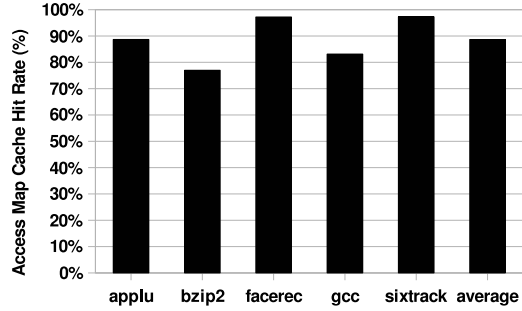
Figure 4.4: Access Map Cache Hit Rate (%) (the values were similar from both Uniform and Hybrid Policies).

because facerec has a very high locality of read and writes to its pages, and the hybrid system migrates them to DRAM. This results in much higher energy savings as well (26%). For *applu*, the energy savings for the uniform system is low (around 10%) due to its high performance overhead. For the hybrid system as well, the overhead is high (around 18%) because of low locality of reads and writes. However, the energy savings is still higher (20%) compared to uniform system.

Thus, the results indicate, that in terms of comparison between the systems, the hybrid system is clearly more beneficial. It is able to exploit the read-friendliness of PRAM as well as the write friendliness of DRAM, and hence achieves better overall performance and energy efficiency.

System Overhead   In this discussion, we analyze the sources of system overhead and their impact in terms of performance and energy in detail. We focus on the overhead due to accesses to the access map and its cache, and page swapping.

Access Map   As described in section 4.2, the access map is used to store write access information to PRAM pages. The updates to the access map and its memory controller cache thus add energy overhead in the system in terms of extra accesses to PRAM and the power consumption of the cache itself.

For understanding the extra accesses to PRAM, Figure 4.4 shows the hit rate of the access map cache. The hit rate was almost the same for both the hybrid and uniform systems. We can see that the hit rate is fairly high across all the benchmarks (average around 90%). This indicates that the overhead is very

Table 4.3: Page Swap Interrupts.

Benchmark	Uniform	Hybrid	% Reduction
applu	29000	18000	38
bzip2	2400	635	74
facerec	24600	1500	94
gcc	1350	320	77
sixtrack	0	0	0

low, since extra PRAM accesses are done only for 10% of writes. From Table 4.2, we know that *wpi* of most of the benchmarks is fairly low, so in the context of overall time frame, the impact is negligible.

For the access map cache, we estimate the power consumption to be around 78mW per access using CACTI 4.1 [46]. Since the cache is accessed only for writes, in the overall time-frame, the extra energy consumption due to it is also very low. It must be noted that the extra energy consumption due to accesses to the access map and its cache is included in the results in Figure 4.3.

Page Swapping   The second source of overhead is page swapping interrupts and the consequent page swaps. Table 4.3 shows the statistics related to page swap interrupts for the uniform and hybrid system. We can observe that for most of the benchmarks, the number of interrupts drop significantly for the hybrid system. For instance, for *facerec*, it drops down by as much as 94%. This happens due to its high *wpi* and significant locality in its reads and writes to a small set of addresses. In the uniform system, when the pages mapping these addresses reach the *threshold*, they get mapped to a new PRAM page. However, sustained writes to these addresses generate further page swap interrupts. In contrast, with the hybrid system, once the pages mapping these addresses reach the threshold, they are mapped to DRAM pages, where they no longer generate any further page swap interrupts. For *applu*, the reduction is smaller (37%) due to its bigger working set, and relatively lower lack of locality of reads-writes.

In terms of time overhead of a page swap, it is fairly low since it is a quick software operation of allocating and copying the page, and modifying the page table entries. We assume it to be 5μs, which is based on the estimate of timer

interrupt overhead in modern systems that get generated as frequently as 1ms. Hence, overall the page swapping overhead is also minimal in terms of performance and energy. Note, the page swap overhead is also included in the results in Figure 4.3.

4.4   Conclusion

In the last two chapters we have evaluated the challenges in achieving energy efficiency for server class systems. We first showed that the effectiveness of DVFS for system level energy savings has largely diminished in modern server class systems, with even simple DPM policies outperforming it. We showed that due to lack of energy proportionality of server class systems, the contribution of DPM for energy savings is also much smaller at the system level. Based on these observations we identified two ways for achieving energy efficiency for such system – designing them to be more energy proportional and increasing their utilization so that they operate in their energy efficient zone through workload characterization. As an example of the first solution we proposed PDRAM, a novel, energy efficient hybrid main memory system based on PRAM and DRAM in this chapter. We highlighted the challenges involved in managing such a system, and provided a hardware/software based solution for it, which exploits the workload characteristics in terms of their read-write intensiveness to intelligently allocate them across DRAM and PRAM. We evaluated the system using benchmarks with varying memory access characteristics and demonstrated that the system can achieve up to 37% energy savings at negligible overhead. Furthermore, we showed that it provides better overall energy and performance efficiency compared to homogeneous PRAM based memory systems as well. The next two chapters focus on the second approach towards energy efficiency for server systems – workload consolidation.

Chapter 4, in part, is a reprint of the material as it appears in Proceedings of the 46th ACM/IEEE Design Automation Conference, 2009. Dhiman, G.; Ayoub, R. and Rosing, T.S. The dissertation author was the primary investigator and author of this paper.

Chapter 5

Energy Efficiency using Workload Consolidation

5.1 Introduction

This chapter looks at the problem of energy efficiency from the perspective of data center and enterprise environments, where the energy consumption of the compute equipment and the associated cooling infrastructure is a major component of the operational costs. As observed in the previous chapters, if workloads in such installations could be consolidated on fewer machines, it can dramatically increase the overall energy efficiency by increasing the overall performance per watt. Consequently, modern data centers and cloud computing providers (like Amazon EC2 [3]) use virtualization (eg. Xen [8] and VMware [40]) to not only get better fault isolation and improved system manageability, but also reduced infrastructure cost through resource consolidation and live migration [23]. Consolidating multiple servers running in different virtual machines (VMs) on a single physical machine (PM) increases the overall utilization and efficiency of the equipment across the whole deployment. Thus, the creation, management and scheduling of VMs across a cluster of PMs in a power aware fashion is key to reducing the overall operational costs. Policies for power aware VM management have been proposed in previous research [86] and are available as commercial products as well (eg. VMware DRS

[99]). These policies require understanding of the power consumption and resource utilization of the PM, as well as its breakdown among the constituent VMs for optimal decision making. Currently they treat the overall CPU utilization of the PM and its VMs as an indicator of their respective power consumption and resource utilization, and use it for guiding the VM management policy decisions (VM migration, dynamic voltage frequency scaling/DVFS, resource allocation etc.). However, in our work we show that based on the characteristics of these different co-located VMs, the overall power consumption and performance of the VMs can vary a lot even at similar CPU utilization levels. This can mislead the VM management policies into making decision that can create hotspots of activity, violate performance requirements and degrade overall energy efficiency. Consequently, it is very important to understand the data center workloads and their characteristics while designing VM management policies.

5.2   Data Center Workloads

This section discusses the classification and key characteristics of the workloads that typically run in modern data centers – batch and service workloads [42].

5.2.1   Services

These workloads refer to the software servers that provide services to clients. The service could be either simple single tier web based, or comprised of multiple tiers spanning web, application, and database servers. Regardless of the scale and architecture of the service, the key property is that they are request driven.   The primary goal for these workloads is to serve the user request within a given time bound to maintain a QoS (Quality of Service) level. In this thesis, we consider the following two applications as representative services:

(a) *RUBiS* [4] is a multi-tier online service that implements the core functions of an auction site including selling, browsing, and bidding. We configure it as a two-tier service model, containing a front-end Apache PHP web server and a

back-end MySQL database. RUBiS provides workloads of different mixes for the client sessions that are emulated on a separate machine. In this chapter, we use the 'browsing mix', which emulates a web-intensive user browsing experience.

(b) *Olio* [5] is a distributed web application that provides an events site similar to popular social networking sites, featuring photos, calendars, and shared comment feeds. Common user operations include homepage accesses, viewing person/event details or adding new events. The client sessions are emulated by the Faban workload generator [71], and based on the type of user request, the Apache PHP web server may carry out several HTTP requests, MySQL database queries, and accesses to a file store for user files, such as photos. Olio can be configured either with or without a memcached server. In our experiments we disable memcacheing to generate a more database intensive workload that provides greater contrast with RUBiS.

5.2.2   Batch

These workloads refer to the (typically) resource intensive jobs that are representative of the analytics, number crunching, and scientific computing class of workloads. The primary goal of these jobs is to maximize the overall instruction throughput, but with no specific response time requirements. We use representative workloads from SPEC-CPU 2K (*eon* – computer visualization, *facerec* – image processing, *equake* - seismic wave propogation simulation etc.) and PARSEC (*streamcluster* – data mining, *swaptions* – financial analysis, *bodytrack* – scientific simulation etc.) benchmark suites, that are heavily throughput intensive, as our batch workloads. The chosen set covers a wide array of application domains as well as both single (SPEC-2K) and multi threaded (PARSEC) execution.

5.2.3   Workload Management

A typical data center, based on the corporate and enterprise requirements or the time of the day could be running only batch jobs, only services or a mix of both types of jobs. The goal of this thesis is to study how to maximize energy

efficiency in all these cases. As we show in the review of the related work, the problem of QoS aware energy efficient management in the presence of services workloads scenario has been studied before [79]. So we focus on the other two cases, i.e. batch only and the mix of batch and services in this thesis. We further show that how a solution optimized for one scenario becomes sub-optimal in the other.

In this chapter, we introduce vGreen, a multi-tiered software system to manage batch VM scheduling with the objective of managing the overall energy efficiency and performance. The basic premise behind vGreen is to understand and exploit the relationship between the architectural characteristics of a VM (eg. instructions per cycle, memory accesses etc.) and its performance and power consumption. vGreen is based on a client server model, where a central server (referred to as *'vgserv'*) performs the management (scheduling, DVFS etc.) of VMs across the PMs (referred to as *'vgnodes'*). The *vgnodes* perform online characterization of the VMs running on them and regularly update the *vgserv* with this information. These updates allow *vgserv* to understand the performance and power profile of the different VMs and aids it to intelligently place them across the *vgnodes* to improve overall performance and energy efficiency.

In the next chapter, we focus on the mixed workload scenario, and introduce the Themis system to manage the heterogeneous mix of workloads. We further show how vGreen and the other existing state of the art is insufficient to maximize the overall energy efficiency in the mixed workload case scenario.

5.3   Related Work

VM Management: A number of systems for management of VMs across a cluster of PMs have been proposed in the past. Eucalyptus [76], OpenNebula [78] and Usher [68] are open source systems, which include support for managing VM creation and allocation across a PM cluster. However, these solutions do not have VM scheduling policies to dynamically consolidate or redistribute VMs. VM scheduling policies for this purpose have also been investigated in the past. In

[103], the authors propose a VM scheduling system, which dynamically schedules the VMs across the PMs based on their CPU, memory and network utilization to avoid hotspots of activity on PMs for better overall performance. The Distributed resource scheduler (DRS) from VMware [99] uses VM scheduling to perform automated load balancing in response to CPU and memory pressure. In [16], the authors propose VM scheduling algorithms for dynamic consolidation and redistribution of VMs for managing QoS requirements of different services VMs in the cluster. They develop dynamic models to capture resource utilization (like CPU utilization) profile of the VMs over different periods of time, which allows them to make precise scheduling decisions that avoid resource bottlenecks. The authors in [41] propose Entropy, which uses constraint programming to determine a globally optimal solution for VM scheduling in contrast to the first fit decreasing heuristic used by [103, 16], which can result in globally sub-optimal placement of VMs. However, these approaches have limited awareness of the performance interference effects between the different VMs and assume that the CPU utilization aggregates upon VM consolidation, which as we show in our work is not always true. Besides, none of these systems are structured to account for QoS requirements of service workloads.

Software level QoS support: Management of QoS for latency sensitive applications in a heterogeneous workload mix on standalone systems has been studied before under the Stanford SMART scheduler [75] and QLinux [93] projects. The primary approach of both the systems is to ensure timely access to CPU for the latency sensitive applications while maintaining proportional sharing of CPU resources for the batch applications. Similar solutions have been proposed for virtualized environments as well [77, 64]. The solution adopted by Themis (adding a QoS state) to guarantee timely CPU access for the service workloads is similar in spirit. However, we further show that the software level support for QoS through timely CPU access is not sufficient to guarantee QoS in presence of interference effects in modern multi-core based systems. Recent work has proposed dynamic VM resource management algorithms for satisfying QoS requirements of workloads in presence of interference effects. In [79], the authors employ CPU capping to

ensure meeting QoS requirements of different consolidated VMs, which as we show in this paper can even be energy inefficient. Besides, their focus is very specific to one class of workloads – just services [79].

Interference Effects: The interference effects due to shared resource usage by co-scheduled workloads on modern multi-core based platforms has been studied before at both the OS and hypervisor levels. The work in [27] shows how the pathological sharing of resources like last level cache and memory bandwidth can severely deteriorate the overall performance as well as energy efficiency. They propose OS scheduling based solutions to resolve the shared resource contention dynamically. However, they explore the problem exclusively for batch jobs. The vGreen system takes the same problem of shared resource usage to the cluster level using virtualization, and develops novel workload characterization schemes to implement energy efficient VM scheduling algorithms by exploiting them. In the presence of both the batch and service workloads, as we show in the next chapter, the interference effects and its impact on the QoS of services can be rather unpredictable and difficult to model. Thus, instead of explicitly modeling the interference, the Themis system solves this problem by inferring it through a performance model based on QoS and CPU utilization feedback.

Power Management: Power management in data center like environments has been an active area of research. In [18], data center power consumption is managed by turning servers off depending on demand. Reducing operational costs by performing temperature aware workload placement has also been explored [72]. In [35], DVFS is performed based on the memory intensiveness of workloads on the server clusters to reduce energy costs. Similarly, [87, 33] use DVFS to reduce average power consumption in blade servers with the objective of performing power budgeting. However, we have already shown in the previous chapter, that in modern server systems, the effectiveness of DVFS for energy management has diminished significantly due to its impact on the performance of the workloads. In this chapter, we further corroborate this observation and show how intelligent VM co-location outperforms state of the art DVFS policies [29, 56] in terms of energy savings.

The problem of power management in virtualized environments has also been investigated. In [74], the authors propose VirtualPower, which uses the power management decisions of the guest OS on virtual power states as hints to run local and global policies across the PMs. It relies on efficient power management policies in the guest OS, and does no VM characterization at the hypervisor level. This makes it difficult to port some of the state of the art power management policies like [29, 56] in guest OS because of lack of exclusive access to privileged resources such as CPU performance counters. This problem has led to adoption of power management frameworks like *cpufreq* and *cpuidle* in recent virtualization solutions (like Xen [8]). In [2], the authors develop a power and performance model of a transaction based application running within the VMs, and use it to drive cluster level energy management through DVFS. However, they assume the application characteristics to be known. In [86], a co-ordinated multi-level solution for power management in data centers is proposed. Their solution is based on a model that uses power estimation (using CPU utilization) and overall utilization levels to drive VM placement and power management. However, the model and results are based on offline trace driven analysis and simulations. In [66], the authors present GreenCloud, an infrastructure to dynamically consolidate VMs based on CPU utilization to produce idle machines, which could be turned off to generate energy savings. However, none of these solutions [74, 2, 86, 66] take the architectural characteristics of the VM into account, which, as we show in section 5.4, directly determine the VM performance and power profile. In [97], the authors use VM characteristics like cache footprint and working set to drive power aware placement of VMs. But their study assumes an HPC application environment, where the VM characteristics are known in advance. Besides, their evaluation is based on simulations. In contrast, vGreen and Themis systems assume a general purpose workload setup with no apriori knowledge on their characteristics.

5.4 Motivation for Workload Characterization

In this chapter we assume Xen as the underlying virtualization hypervisor. It is a standard open-source virtualization solution, which also forms the baseline

technology for commercial products like XenSource, Oracle VM etc. However, the ideas presented in this chapter are independent of Xen, and can be applied to other virtualization solutions like kernel based virtual machines (KVM) etc. as well. In Xen, a VM is an instance of an OS, which is configured with virtual CPUs (vCPUs) and a memory size. The number of vCPUs and memory size is configured at the time of VM creation. Xen virtualizes the real hardware to the VM making the OS running within it believe that it is running on a real machine. A PM can have multiple VMs active on it at any point in time, and Xen multiplexes them across the real physical CPUs (PCPUs) and memory. The entity that Xen schedules over the PCPU is the vCPU, making it the fundamental unit of execution. Thus, a vCPU is analogous to a thread, and a VM is analogous to a process in a system running a single OS like Linux. In addition, Xen provides a control VM, referred to as Domain-0 (or Dom-0), which is what the machine running Xen boots into. It acts as an administrative interface for the user, and provides access to privileged operations like creating, destroying or migrating VMs.

5.4.1   Performance and Power Profile of VMs

The nature of workload executed in each VM determines the power profile and performance of the VM, and hence its energy consumption. As discussed before, VMs with different or same characteristics could be co-located on the same PM. In this section we show, that co-location of VMs with heterogeneous characteristics on PMs is beneficial for overall performance and energy efficiency across the PM cluster.

For understanding this, we performed some experiments and analysis on two benchmarks from SPEC-CPU 2000 suite, namely *eon* and *mcf*. These two benchmarks have contrasting characteristics in terms of their CPU and memory utilization. While *mcf* has high memory per cycle (MPC) accesses and low instructions committed per cycle (IPC), *eon* has low MPC and high IPC. We use a testbed of two dual Intel quad core Xeon (hyperthreading equipped) based PMs (sixteen CPUs each) running Xen. On each of these PMs, we create two VMs with eight virtual CPUs (vCPUs) each (total of four VMs). Inside each VM we
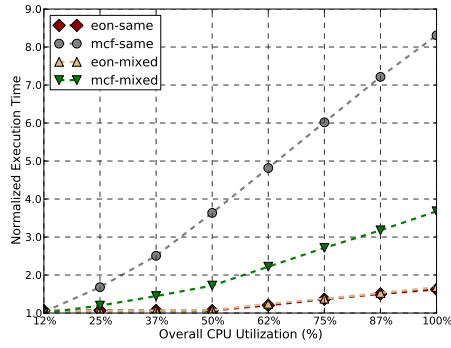
(a) Normalized Execution Time
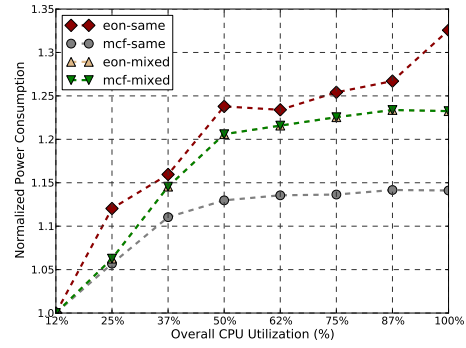
(b) Normalized Power Consumption
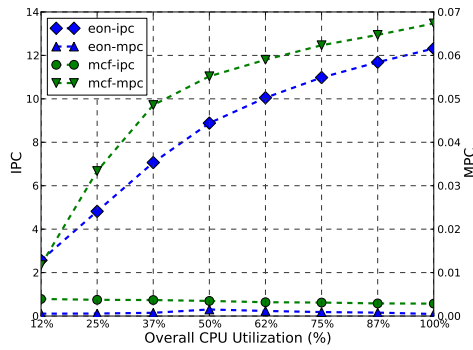
(c) Comparison of Aggregate IPC and MPC

(d) Comparison of 'mixed' vs 'same' VM placement schedules
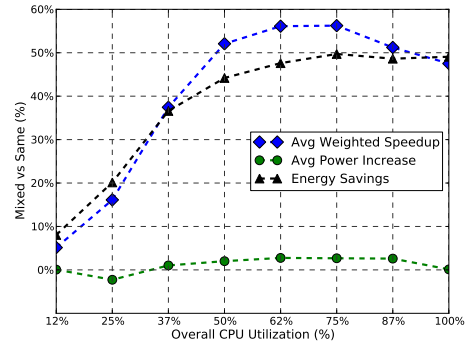
Figure 5.1: Comparison of various metrics of *eon* and *mcf* across 'mixed' and 'same' schedules.

execute either *eon* or *mcf* as the workload. We use multiple instances/threads of the benchmarks to generate higher utilization levels. For our PM (sixteen CPUs), this implies four instances for 25% utilization, eight instances for 50% and sixteen instances for 100% utilization. Each PM is equipped with power sensors, which are interfaced to the Dom-0 OS in a standardized fashion using Intelligent Platform Management Interface (IPMI) [55]. We periodically (every 2s) query the IPMI interface to log the power consumption of the whole PM for all our experiments.

In our first set of experiments, we run homogeneous VMs on each PM, i.e. the two VMs with *mcf* on one PM, and two with *eon* on the other. We refer to this VM placement schedule as 'same' indicating homogeneity. During the execution, we record the execution time of all the benchmark instances. Figure 5.1a

shows the normalized execution time results for different number of instances of the benchmarks, where the execution times are normalized against the execution time with two instances (one instance per VM). We can observe that for *mcf* in the 'same' schedule (shown as 'mcf-same'), as the CPU utilization increases, the execution time almost increases linearly. For 100% utilization *mcf*, the execution time is almost 8.5x compared to the baseline execution time. The primary reason for such an observation is the high MPC of *mcf*. The high MPC results in higher cache conflict rate and pressure on the memory bandwidth when multiple threads execute, which decreases the effective IPC per thread and hence increases its execution time. This is illustrated by the plot of aggregate IPC and MPC of all *mcf* threads in Figure 5.1c. We can see how the MPC increases by around 7x, as CPU utilization goes from 12% to 100%. However, the aggregate IPC almost remains constant, which implies IPC per thread goes down significantly, resulting in increased execution time observed in Figure 5.1a. In contrast, for *eon* ('eon-same'), the execution time is fairly independent of the CPU utilization due to its much lower MPC. We can observe that the execution time shows an increase beyond 50% utilization. This happens since our machine has eight cores and sixteen CPUs (due to hyperthreading), with two CPUs per core. When we reach 50% utilization that corresponds to eight threads of the benchmark, and beyond that the threads start sharing the pipeline, which reduces the individual IPC of threads sharing the pipeline. This phenomena is illustrated in Figure 5.1c, where the IPC slope of *eon* drops off a little beyond 50% CPU utilization. However, this increase in execution time is trivial compared to that of *mcf* as seen in Figures 5.1a and 5.1c. In summary, this analysis indicates that the performance of a VM has a strong negative co-relation to utilization rate of the memory subsystem.

Similarly, Figure 5.1b shows the system level power consumption of the PMs normalized against the power consumption with just two threads. We can observe that for *eon* ('eon-same'), the power consumption increases almost linearly to the increase in utilization. This happens, since it has high IPC, which implies higher CPU resource utilization and power consumption. We can observe that the slope of increase in power changes at 50% utilization. This is again due to pipeline sharing

between threads beyond 50% utilization, which lowers the contribution of new threads to power consumption (see Figure 5.1c). In contrast, for *mcf*, the power consumption increases initially but then it saturates. This primarily happens due to the lower IPC of threads at higher utilization levels as discussed above. As a consequence of this, the difference in power consumption between the two PMs is almost 20% (∼45 Watts in our measurements). This analysis indicates that the power consumption of a VM has direct co-relation to IPC of the workload running inside it.

These results indicate that co-scheduling VMs with similar characteristics is not beneficial from energy efficiency point of view at the cluster level. The PM running *mcf* contributes to higher system energy consumption, since it runs for a significantly longer period of time. To understand the benefits of co-scheduling heterogeneous workloads in this context, we swapped two VMs on the PMs, hence running VMs with *mcf* and *eon* on both the PMs. We refer to this VM placement schedule as 'mixed', indicating the heterogeneity. Figure 5.1 shows the results (indicated as 'mixed') achieved for this configuration in terms of normalized execution time and power consumption. We can observe that *eon* execution time almost stays the same, while *mcf* execution time goes down significantly at higher utilization rates (around 450% reduction at 100% utilization). This happens because we now get rid of the hot-spot of intense activity in the memory subsystem on one PM (running just the *mcf* VMs in the 'same' schedule), and share the overall system resources in a much more efficient fashion. The average power consumption of the two PMs becomes similar, and roughly lies between that of the two PMs in the 'same' schedule, as the overall IPC is also much better balanced across the cluster.

Figure 5.1d illustrates the comparison of the 'mixed' and 'same' VM schedules, and highlights the benefits of the 'mixed' schedule. It plots three key metrics to capture this: (1) **Energy savings:** We estimate the energy reduction in executing each combination of VMs using 'mixed' over 'same' schedule. This is calculated by measuring the total energy consumption for a VM combination with two schedules, and then taking their difference. We can observe that across all utilization levels, the 'mixed' schedule is clearly more energy efficient compared

to the 'same' schedule. At higher utilization rates (50% and beyond), it achieves as high as 50% energy savings. This primarily happens due to the high speedup achieved by it compared to 'same' schedule while keeping the average power consumption at a similar level. The next two metrics provide details on both the speedup and average power consumption achieved by the two schedules. (2) **Average Weighted Speedup (AWS):** This metric captures how fast the workload runs on the 'mixed' schedule compared to 'same' schedule. The AWS is based on a similar metric defined in [91]. It is defined as:

$$AWS = \frac{\sum_{VM_i} \frac{T_{same_i}}{T_{alone_i}}}{\sum_{VM_i} \frac{T_{mixed_i}}{T_{alone_i}}} - 1 \qquad (5.1)$$

where, T_{alone_i} is the execution time of VM_i when it runs alone on a PM, and T_{same_i} and T_{mixed_i} are its execution time as part of a VM combination with 'same' and 'mixed' schedules respectively. To calculate AWS, we normalize T_{same_i} and T_{mixed_i} against T_{alone_i} for each VM, and then take ratio of the sum of these normalized times across all the VMs in the combination as shown in equation 5.1. AWS > 0 implies that the VM combination runs faster with 'mixed' schedule and vice versa. Figure 5.1d clearly shows, that the 'mixed' schedule is able to achieve significant speedup. The AWS reaches as high as 57% due to efficient resource sharing and contributes significantly to the energy savings discussed above. (3) **Increase in power consumption**: This metric captures the difference between the average power consumption of the PMs under the 'mixed' and 'same' schedule. This is important, since we need to make sure that the speedup achieved does not result in much higher average power consumption across the cluster. Figure 5.1d shows that the increase in system power consumption is trivial (<3%) across all the utilization levels. Thus, high speedups at almost similar average power consumption results in significant energy savings illustrated in Figure 5.1d.

In summary, this discussion provides us key insights into the VM management problem: (1) VM characteristics provide invaluable information on both the power as well as performance profile of VMs. (2) VM scheduling policies should try to co-schedule VMs with heterogeneous characteristics on the same PM. This results in efficient sharing of resources across the cluster and as a consequence

is beneficial from both energy efficiency and performance point of view. This is achievable in virtualized environments, since VMs can be dynamically migrated at runtime across PMs at low overhead using '*live migration*' [23].

This provides strong motivation to use online characterization of VMs for system wide VM management. In the next section, we describe the overall architecture of vGreen, and present details on how it constructs VM characteristics dynamically at run time using a novel hierarchical approach.

5.5   vGreen Design

In this section, we present the details on the design, architecture and implementation of our system, vGreen. Building upon the discussion in the last section, we show how the system is structured to capture the CPU and memory utilization rates of individual VMs, and how it uses it to manage the VMs in an efficient fashion across a PM cluster.
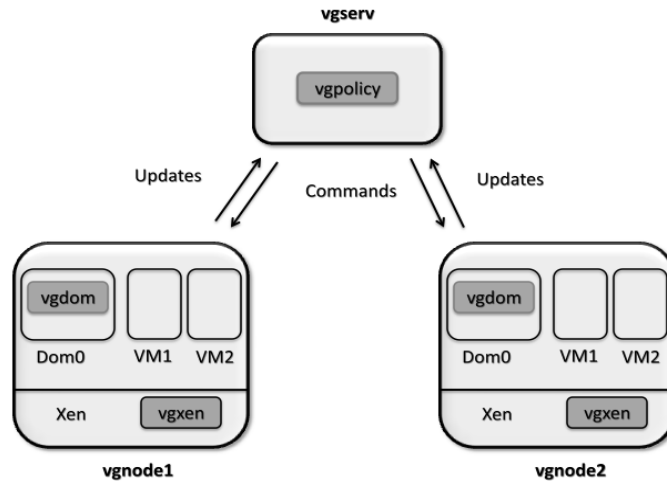
Figure 5.2: Overall Architecutre of vGreen system.

Figure 5.2 illustrates the overall architecture of vGreen, which is based on a client-server model. Each PM in the cluster is referred to as a vGreen client/node (*vgnode*). There is one central vGreen server (*vgserv*), which manages VM scheduling across the *vgnodes* based on a policy (*vgpolicy*) running on the *vgserv*. The *vgpolicy* decisions are based on the value of different metrics, which capture MPC, IPC, and utilization of different VMs, that it receives as updates from the *vgnodes*
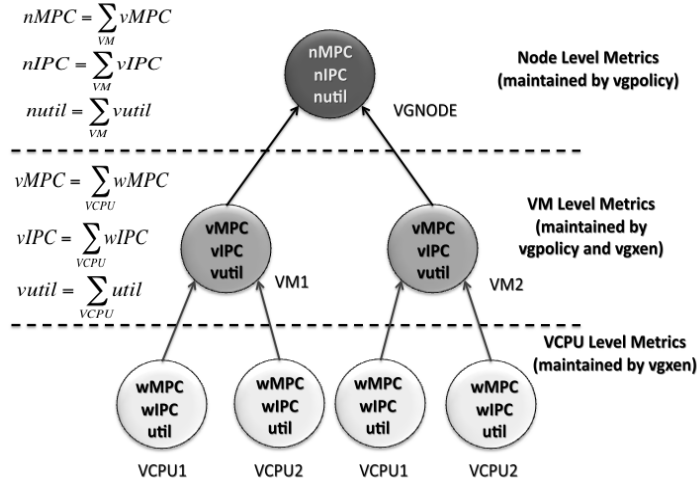
Figure 5.3: An example of Hierarchical Workload Characterization in vGreen.

running those VMs. The metrics are evaluated and updated dynamically by the vGreen modules in Xen (*vgxen*) and Dom-0 (*vgdom*) on each *vgnode*. Regular updates from the *vgnodes* on the metrics allow the *vgpolicy* to balance both the power consumption and overall performance across the PMs. We now describe the vGreen components and the metrics employed in detail.

vgnode A *vgnode* refers to an individual PM in the cluster. A *vgnode* might have multiple VMs running on it at any given point in time as shown in Figure 5.2. Each *vgnode* has vGreen modules (*vgxen* and *vgdom*) installed on them.

vgxen: The *vgxen* is a module compiled into Xen (see Figure 5.2) and is responsible for characterizing the CPU and memory behavior (specifically IPC and MPC) of running VMs. Since multiple VMs with possibly multiple vCPUs might be active concurrently, it is important to cleanly isolate the characteristics of each of these different entities. vGreen adopts a hierarchical approach for this purpose as illustrated in Figure 5.3. The lowest level of the hierarchy is the vCPU level, which is the fundamental unit of execution and scheduling in Xen. When a vCPU is scheduled on a PCPU by the Xen scheduler, *vgxen* starts the CPU performance counters of that PCPU to count the following events: **(1) Instructions Retired (INST)**, **(2) Clock cycles (CLK)**, and **(3) Memory accesses (MEM)**.

When that vCPU consumes its time slice (or blocks) and is removed from the PCPU, *vgxen* reads the performance counter values and estimates its MPC

(MEM/CLK) and IPC (INST/CLK) for the period it executed. This process is performed for every vCPU executing in the system across all the PCPUs. To effectively estimate the impact of these metrics on the vCPU power consumption and performance, *vgxen* also keeps track of the CPU utilization (*util*) of each vCPU, i.e. how much time it actually spends executing on a PCPU over a period of time. This is important, since even a high IPC benchmark will cause high power consumption only if it is executing continuously on the PCPU. Hence, the metric derived for each vCPU is weighted by its *util*, and is referred to as the current weighted MPC and IPC ($wMPC_{cur}$ and $wIPC_{cur}$) as shown below:

$$wMPC_{cur} = MPC \cdot util$$
$$wIPC_{cur} = IPC \cdot util \qquad (5.2)$$

They are referred to as 'current', since they are estimated based on the IPC/MPC values from the latest run of a vCPU. To also take into account the previous value of these metrics, we maintain them as running exponential averages. The equation below shows how weighted MPC is estimated:

$$wMPC = \alpha \cdot wMPC_{cur} + (1 - \alpha) \cdot wMPC_{prev} \qquad (5.3)$$

where, the new value of weighted MPC ($wMPC$) is calculated as an exponential average of $wMPC_{prev}$, the previous value of $wMPC$, and $wMPC_{cur}$ (equation 5.2). The factor α determines the weight of current value ($wMPC_{cur}$) and history ($wMPC_{prev}$). In our implementation we use α=0.5, thus giving equal weight to both. The IPC metric is computed in a similar fashion as discussed above. We store these averaged metrics in the Xen vCPU structure to preserve them faithfully across vCPU context switches. This constitutes the metric estimation at the lowest level of the hierarchy as shown in Figure 5.3.

At the next level, *vgxen* estimates the aggregate metrics (vMPC, vIPC, vutil) for each VM by adding up the corresponding metrics of its constituent vCPUs, as shown in the middle level of Figure 5.3. This information is stored in VM structure of Xen to personalize metrics at per VM level and is exported to Dom-0 through a shared page, which is allocated by *vgxen* at the boot-up time.

vgdom: The second vGreen module of *vgnode* is the *vgdom* (see Figure 5.2). Its main role is to periodically (T_{up_period}) read the shared page exported by *vgxen* to get the latest characteristics metrics for all the VMs running on the vgnode, and update the *vgserv* with it. In addition, *vgdom* also acts as an interface for the *vgnode* to the *vgserv*. It is responsible for registering the *vgnode* with the *vgserv* and also for receiving and executing the commands sent by the *vgserv* as shown in Figure 5.2.

vgserv The *vgserv* acts as the cluster controller and is responsible for managing VM scheduling and power management across the *vgnode* cluster. The *vgpolicy* is the core of *vgserv*, which makes the scheduling and power management decisions based on periodic updates on the VM metrics from the *vgnodes*. The metrics of each VM are aggregated by the *vgpolicy* to construct the top level or node level metrics (nMPC, nIPC, nutil) as shown in Figure 5.3. Thus, the knowledge of both the node level and VM level metrics allow the *vgpolicy* to understand not only the overall power and performance profile of the whole *vgnode*, but also fine grained knowhow of the breakdown at VM level.

Based on these metrics, the *vgpolicy* runs its balancing and power management algorithm periodically (T_{p_period}). The basic algorithm is motivated by the fact that VMs with heterogeneous characteristics should be co-scheduled on the same *vgnode* (section 5.4). The problem of consolidation of VMs in minimum possible PMs has been explored in previous work [41, 103], and is similar to bin-packing problem, which is computationally NP-hard. As discussed in Section 5.3, the existing solutions perform the consolidation based on just CPU utilization. Our balancing algorithms build on top of these existing algorithms to perform balancing based on MPC and IPC as well. The overall algorithm runs in the following four steps:

(1) **MPC balance**: This step ensures that nMPC is balanced across all the *vgnodes* in the system for better overall performance and energy efficiency across the cluster. Table 5.1 gives an overview of how the MPC balance algorithm works for a *vgnode* $n1$. The algorithm first of all checks, if the nMPC of $n1$ is greater than a threshold $nMPC_{th}$ (step 1 in Table 5.1). This threshold is representative of

Table 5.1: MPC Balance Algorithm.

Input: *vgnode n1*

1: **if** $nMPC_{n1} < nMPC_{th}$ **then**

2:     **return**

3: **end if**

4: $pm\_min \leftarrow NULL$

5: **for** all vgnodes n_i except $n1$ **do**

6:     **if** $(nMPC_{n_i} < nMPC_{th})$ and $(nMPC_{n1} - nMPC_{th}) > (nMPC_{th} - nMPC_{n_i})$ **then**

7:         **if** $!pm\_min$ or $nMPC_{pm\_min} > nMPC_{n_i}$ **then**

8:             $pm\_min \leftarrow n_i$

9:         **end if**

10:     **end if**

11: **end for**

12: $vm_{mig} \leftarrow NULL$

13: **for** all vm_i in $n1$ **do**

14:     **if** $(nMPC_{th} - nMPC_{pm\_min}) > vMPC_{vm_i}$ and $vMPC_{vm_i} > vMPC_{vm_{mig}}$ **then**

15:         $vMPC_{vm_{mig}} \leftarrow vMPC_{vm_i}$

16:     **end if**

17: **end for**

18: **if** pm_min and vm_{mig} **then**

19:     $do\_migrate(vm_{mig}, n1, pm\_min)$

20: **end if**

---

whether high MPC is affecting the performance of the VMs in that *vgnode*. This is based on the observation in section 5.4, that for lower MPC workloads (like *eon*), the memory subsystem is lightly loaded and has little impact on the performance of the workload. Hence, if nMPC is smaller, the function returns, since there is no MPC balancing required for $n1$ (step 2 in Table 5.1). If it is higher, then in steps 4-11, the algorithm tries to find the target *vgnode* with the minimum nMPC (pm_min) to which a VM from $n1$ could be migrated to resolve the MPC imbalance, subject to the condition in step 6. The condition states that the target *vgnode* (n_i) nMPC ($nMPC_{n_i}$) must be below $nMPC_{th}$ by atleast ($nMPC_{n1} - nMPC_{th}$). This is required, since otherwise migration of a VM from $n1$ to n_i cannot bring $n1$ below the MPC threshold or might make n_i go above the MPC threshold. In step 7 and 8, it stores the node n_i as target minimum nMPC *vgnode* (pm_min), if its nMPC ($nMPC_{n_i}$) is lower than the nMPC of the *vgnode* currently stored as pm_min. This way, once the loop in step 5 completes, it is able to locate the *vgnode* in the system with the least nMPC (pm_min).

Once the pm_min is found, the algorithm finds the VM (vm_{mig}), that could be migrated to pm_min for resolving the MPC imbalance (steps 12-17). For this purpose it scans the list of VMs on $n1$ to find the VM with the maximum vMPC, which if migrated, does not reverse the imbalance by making nMPC of pm_min more than $nMPC_{th}$ (steps 14-15). If such a VM is found, the algorithm invokes the *do_migrate* function to live migrate vm_{mig} from $n1$ to pm_min [23] in step 19. The decisions taken by the *vgpolicy* (updates, migration) are communicated to the *vgnodes* in form of commands as shown in Figure 5.2, while the *vgdom* component on the *vgnode* actually accomplishes the migration.

The complexity of the MPC balance algorithm (Table 5.1) is linear ($O(n)$), where 'n' is the number of *vgnodes* in steps 5-11, and number of VMs on $n1$ in steps 13-17) for resolving an MPC bottleneck, since it requires a single scan of vgnodes and VMs to detect and resolve it. Hence, in terms of implementation and performance the algorithm is simple and scalable.

(2) **IPC balance**: This step ensures nIPC is balanced across the *vgnodes* for better balance of power consumption across the PMs. The algorithm is similar
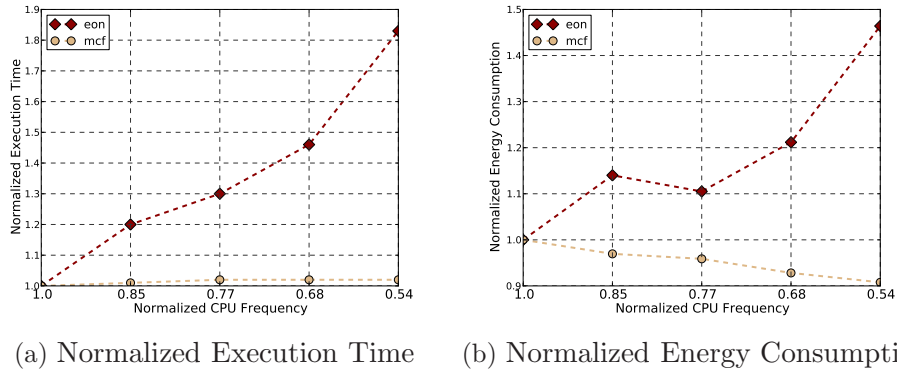
(a) Normalized Execution Time   (b) Normalized Energy Consumption

Figure 5.4: Comparison of execution time and energy consumption of *mcf* and *eon* at different frequency levels.

to MPC balance, but uses nIPC instead of nMPC.

(3) **Util balance**: This step balances the CPU utilization of *vgnodes* to ensure there are no overcommitted nodes in the system, if there are other underutilized *vgnodes*. The algorithm is again similar to MPC balance, but uses *nutil* instead of nMPC.

(4) **Dynamic Voltage Frequency Scaling (DVFS)**: The *vgpolicy* may issue a command to scale the voltage-frequency setting (v-f setting) of a *vgnode*, if it deems that it is more energy efficient than VM migration. This may happen, if there are not enough heterogeneous VMs across the cluster to be able to balance the resource utilization evenly. The DVFS policy is itself based on state of the art DVFS policies [29, 56], that exploit the characteristics of the workload to determine the best suited v-f setting for it. Specifically, it aggressively downscales the v-f setting if the overall MPC is high ($> nMPC_{th}$), otherwise keeps the system at the highest v-f setting.

Figure 5.4 gives the intuition behind the policy using an example of two benchmarks, *mcf* and *eon*, running at 90% CPU utilization level. It plots the execution time (Figure 5.4a) and energy consumption (Figure 5.4b) at five different v-f settings. The execution time, energy consumption and the v-f settings are normalized against the values at the highest v-f setting. We can observe that as the frequency is decreased, the execution time of *eon* almost increases in proportion to the drop in frequency. For instance, at normalized frequency of 0.54, the increase

in execution time is more than 80% ($\sim \frac{1}{0.54}$). This happens since *eon* has high IPC, and uses the pipeline of the processor intensively, which makes its execution time a function of the clock rate of the pipeline or the CPU frequency. This huge performance degradation has a direct impact on the energy consumption of eon at lower v-f settings as shown in Figure 5.4b. We can observe that at all the frequencies the system consumes more energy compared to the highest v-f setting, reaching as high as 40% more. This implies, that for high IPC workloads, DVFS is actually energy inefficient.

In contrast, for *mcf*, which has high MPC, we observe that the execution time (Figure 5.4a) is actually fairly independent of the CPU frequency. This is a consequence of the high degree of CPU stalls that occur during its execution due to frequent memory accesses, which makes its execution time insensitive to actual CPU frequency. The low performance degradation translates into system level energy savings (see Figure 5.4b), which reaches 10% at the lowest frequency.

This example further illustrates the fact presented in chapter 3, that the effectiveness of DVFS for energy savings is not very significant in modern server class systems. These observations also motivate our approach to focus more on efficient VM scheduling to achieve higher energy savings rather than on aggressive DVFS. Rather, the system resorts to DVFS only when no further benefits are achievable through scheduling and the MPC is high enough to achieve energy savings. As we show in section 5.7, such an approach enables energy savings under both heterogeneous and homogeneous workload scenarios through VM scheduling and aggressive DVFS respectively.

The four steps described above in the overall algorithm have relative priorities to resolve conflicts, if they occur. MPC balance is given the highest priority, since memory bottleneck severely impacts overall performance and energy efficiency as identified in Section 5.4 (Figure 5.1a). IPC balance results in a more balanced power consumption profile, which helps create an even thermal profile across the cluster and hence reduces cooling costs [6], and is next in the priority order. Finally, Utilization balance results in a fairly loaded system, and is representative of the prior state of the art scheduling algorithms. DVFS step (step

4), as explained above, is invoked only if the system is already balanced from the perspective of MPC, IPC and CPU utilization, and no further savings are possible through VM scheduling.

5.6   vGreen Implementation

Our testbed for vGreen includes two state of the art 45nm Dual Intel Quad Core Xeon X5570 (Intel Nehalem architecture with 16 PCPUs each) based server machines with 24GB of memory, which act as the *vgnodes*, and a Core2Duo based desktop machine that acts as the *vgserv*. The *vgnodes* run Xen3.3.1, and use Linux 2.6.30 for Dom-0.

The *vgxen* module is implemented as part of the Xen credit scheduler (the default scheduler) to record vCPU and VM level metrics. It stores all the VM level metrics in a shared page mapped into the address space of Dom-0. It further exposes a new hypercall, which allows the *vgdom* to map this shared page into its address space, when it gets initialized (as explained in section 4.1.1). The *vgdom* module is implemented in two parts on Dom-0:

(1) *vgdom Driver*: A Linux driver that interfaces with *vgxen* to get the VM characteristics and exposes it to the application layer. When initialized, it maps the shared page storing the VM metrics into its address space using the hypercall discussed above. Such a design makes getting the VM metrics a very low overhead process, since it is a simple memory read.

(2) *vgdom App*: An application client module that is responsible for interfacing and registering the *vgnode* with the *vgserv*. The primary responsibility is to get the VM metrics data from the driver and pass it on to *vgserv* as shown in Figure 5.2. It also accepts *vgserv* commands for VM migration or DVFS and processes it.

vGreen requires no modifications to either the OS or the application running within the VMs. This makes the system non intrusive to customer VMs and applications and hence easily deployable on existing clusters. The *vgserv* and *vgpolicy* run on Linux 2.6.30, and are implemented as application server modules.

The system is designed to seamlessly handle dynamic entry and exit of *vgnodes* in the system without any disruption to the *vgpolicy*. On initialization, *vgserv* opens a well known port and waits for new *vgnodes* to register with it. When *vgnodes* connect, *vgserv* instructs them to regularly update it with VM characteristics (T_{up_period}), and accordingly updates the node and VM level metrics. It runs the *vgpolicy* every T_{p_period} and performs balancing or DVFS decisions, which it communicates to the *vgnode* through commands as described in section 5.5. If a *vgnode* goes offline, the connection between it and the *vgserv* is broken. This results in a dynamic cleanup of all the state associated with that *vgnode* on the *vgserv*.

For real world deployments, vGreen can be either installed as a standalone system for VM management, or as part of bigger infrastructure management systems like OpenNebula [78] or Grid Virtualization Engine [100] as well. For instance, in context of GVE, *vgxen* and *vgdom* can be incorporated into the '*GVE agent service layer*', which is the monitoring layer, while *vgserv* and *vgpolicy* can be implemented as part of the '*GVE site service layer*', which is the control layer.

5.7   Evaluation Methodology

For our experiments, we use benchmarks with varying characteristics from the SPEC-CPU 2000 benchmark suite. The used benchmarks and their characteristics are illustrated in Table 5.2. We run each of these benchmarks inside a VM, which is initialized with eight vCPUs and 4GB of memory. We generate experimental workloads by running multiple VMs together, each running one of the benchmarks. For each combination run we sample the system power consumption of both the vgnodes every 2s using the power sensors in the PM, which we query through the IPMI interface [55].

We compare vGreen to a VM scheduler that mimics the Eucalyptus VM scheduler [76] for our evaluation. Eucalyptus is an open source cloud computing system, that can manage VM creation and allocation across a cluster of PMs. The default Eucalyptus VM scheduler assigns VMs using a greedy policy, i.e. it

Table 5.2: Benchmarks Used.

Benchmark	Characteristics
eon	High IPC/Low MPC
applu	Medium IPC/High MPC
perl	High IPC/Low MPC
bzip2	Medium IPC/Low MPC
equake	Low IPC/High MPC
gcc	High IPC/Low MPC
swim	Low IPC/High MPC
mesa	High IPC/Low MPC
art	Medium IPC/High MPC
mcf	Low IPC/High MPC

allocates VMs to a PM until its resources (number of CPUs and memory) are full. However, this assignment is static, and it does not perform any dynamic VM migration based on actual PM utilization at runtime. For fair comparison, we augment the Eucalyptus scheduler with the CPU utilization metrics and algorithm proposed in the previous section, which allow it to redistribute/consolidate VMs dynamically at run-time. This enhancement is representative of the metrics employed by the existing state of the art policies, which use CPU utilization for balancing (see section 5.3). We refer to this enhanced scheduler as E+. For further fairness in comparison, we use the same initial assignment of VMs to PMs as done by the default Eucalyptus scheduler for both E+ and vGreen.

We report the comparative results of vGreen and E+ for two primary parameters:

(1) **System Level Energy savings:** We estimate the energy reduction in executing each combination of VMs using vGreen over E+. This is calculated by measuring the total system level energy consumption for a VM combination with E+ and vGreen, and then taking their difference. Note, that the combinations may execute for different times with E+ and vGreen, and since we do not know the state of the system after the execution (could be active if there are more jobs,

or be in sleep state if nothing to do), we only compare the energy consumed during active execution of each combination.

(2) **Average Weighted Speedup:** We also estimate the average speedup of each VM combination with vGreen. For this, we use the weighted speedup (AWS) based on a similar metric defined earlier in section 5.4 (refer to equation 5.1). It is defined as:

$$AWS = \frac{\sum_{VM_i} \frac{T_{e+_i}}{T_{alone_i}}}{\sum_{VM_i} \frac{T_{vgreen_i}}{T_{alone_i}}} - 1 \qquad (5.4)$$

where, T_{alone_i} is the execution time of VM_i when it runs alone on a PM, and T_{e+_i} and T_{vgreen_i} are its execution time as part of a VM combination with E+ and vGreen respectively. AWS > 0 implies that the VM combination runs faster with vGreen and vice versa.

For all our experiments, we use P_{p_period} and P_{up_period} as 5s. Based on our experiments across different benchmarks, we choose $nMPC_{th}$ as 0.02 and $nIPC_{th}$ as 8. These threshold values allowed us to cleanly separate memory and CPU intensive VMs from each other.

5.8   Results

5.8.1   Heterogeneous Workloads

In the first set of experiments, we use combinations of VMs running benchmarks with heterogeneous characteristics. Each VM consists of multiple instances of the benchmark to generate different CPU utilization levels. In total, we run four VMs, varying the overall CPU utilization of *vgnodes* between 50% to 100%. We choose this range of CPU utilization, since it is representative of a consolidated environment, where multiple VMs are consolidated to get higher overall resource utilization across the cluster [103]. We run CPU intensive benchmarks in two VMs, and memory intensive benchmarks in the other two. We did experiments across all possible heterogeneous VM combinations, but for the sake of clarity and brevity, have included results for 19 workloads in the following discussion. The excluded
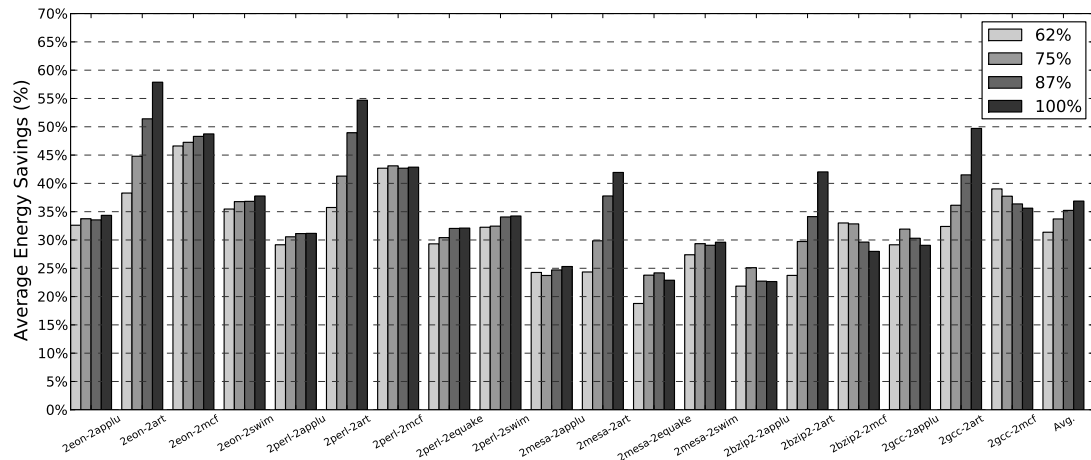
(a) System Level Energy Savings
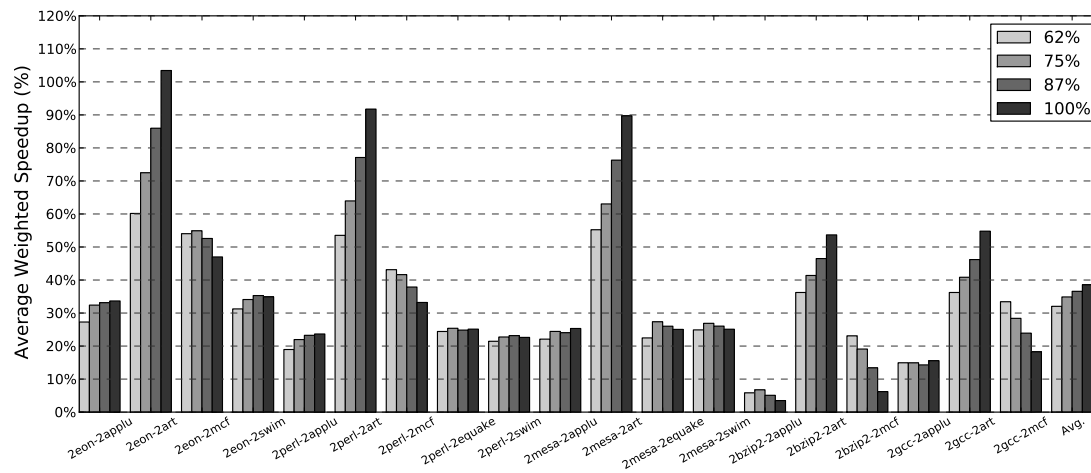
(b) Average Weighted Speedup

Figure 5.5: Comparison of E+ and vGreen. The results are normalized against E+ system.

results lead to similar average metrics and conclusions as reported below.

Figure 5.5 shows the overall results across different utilization levels for the vGreen system normalized to that with E+. The x-axis on the graphs shows the initial distribution of VMs on the physical machines by the default Eucalyptus scheduler. For instance, *2gcc/2art* means that two VMs running *gcc* are on the first PM, while the two VMs running *art* are on the second. We can observe in Figure 5.5a, that vGreen achieves an average of between 30-40% system level energy savings across all the utilization levels, reaching as high as 60%. The high energy savings are a result of the fact that vGreen schedules the VMs in a much more efficient fashion resulting in higher speedups while maintaining similar average power consumption. This results in energy savings, since now the benchmarks run and consume active power for a smaller duration.

Figure 5.5b shows that vGreen achieves an average of 30-40% speedup over E+ across all the combinations at all utilization levels, reaching as high as 100%. The reason for this is that E+ co-locates the high IPC VMs on one *vgnode*, and the high MPC ones on the second one. Thereafter, since the CPU utilization of both the *vgnodes* is balanced, no dynamic relocation of VMs is done. With vGreen, although the initial assignment of the VMs is same as with E+, the dynamic characterization of VMs allows the *vgserv* to detect a heavy MPC imbalance. This initiates migration of a high MPC VM to the second *vgnode* running the high IPC VMs. This results in an IPC and utilization imbalance between the two *vgnodes*, since the second *vgnode* now runs a total of three high utilization VMs. This is detected by *vgserv* and it responds by migrating a high IPC VM to the first *vgnode*. This creates a perfect balance in terms of MPC, IPC and utilization across both the *vgnodes*. This results in significant speedup as observed in Figure 5.5b. We can see in Figure 5.5b, that some combinations achieve higher weighted speedup compared to others. For instance, for *2eon/2applu* combination it is around 30%, while for *2eon/2art* it is over 100%. This difference is due to the fact that co-location of *art* and *eon* VMs significantly benefits *art* from the point of view of larger cache and memory bandwidth availability, since it has very high MPC. In contrast, *applu* benefits lesser due to its lower overall MPC compared to *art*, which
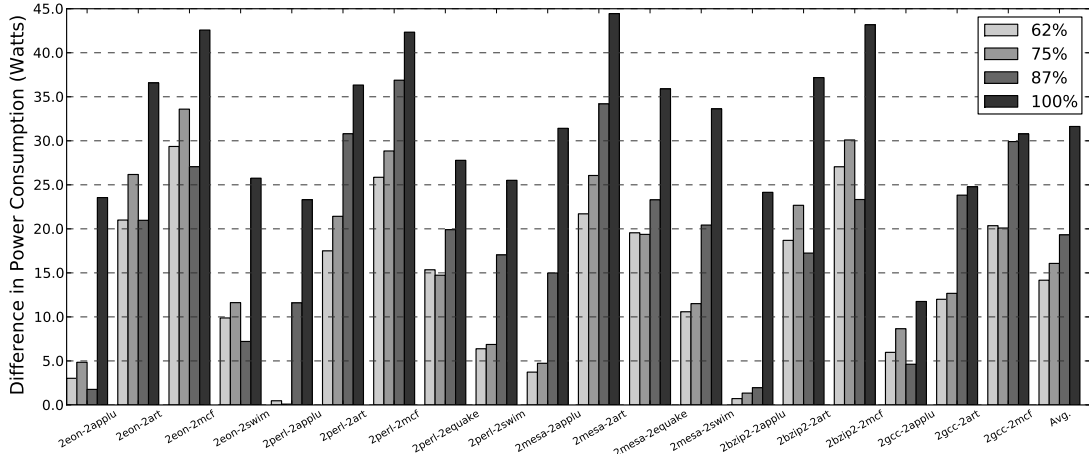
Figure 5.6: Power consumption imbalance in E+: The difference in power consumption between the two PMs under the E+ scheduling algorithm.

results in relatively smaller weighted speedup.

Another disadvantage of not taking the characteristics of the workload into account for scheduling is that there could be significant imbalance in power consumption across the nodes in a cluster. For instance, the node running high IPC workloads might have much higher power consumption compared to the node running high MPC workloads (as observed in section 5.4). This can create power hot spots on certain nodes in the cluster, and be detrimental to the overall cooling energy costs [6]. Figure 5.6 illustrates the imbalance in power consumption across the two *vgnodes* under the E+ system. We can see that the average imbalance in power consumption could be as high as 30W, with the highest imbalance close to 45W. With vGreen system, this imbalance is almost negligible due to the better balance of IPC and utilization across the machines. This results in a better overall thermal and power profile and reduces power hotspots in the cluster.

Comparison with DVFS policies: A possible way for saving energy with the E+ system is to augment it with a DVFS policy. For comparison, we consider two policies for the E+ system:

(1) The 'naive' policy: This policy simply resorts to throttling the CPU in order to reduce the energy consumption in the system. We refer to the system with the 'naive' policy as E+nDVFS.

(2) The 'smart' policy: This policy is the same as incorporated into the
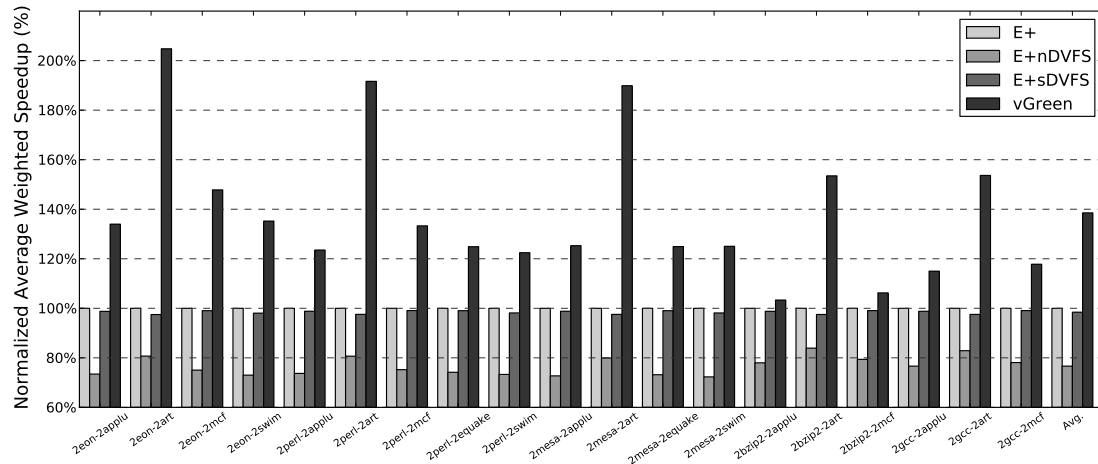
(a) Normalized Average Weighted Speedup
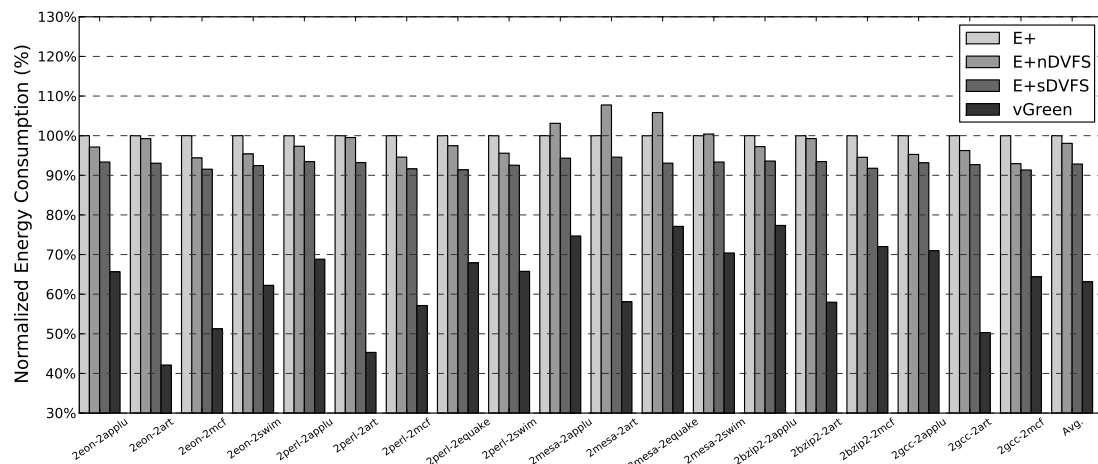
(b) Normalized Energy Consumption

Figure 5.7: Comparison of E+, E+nDVFS, E+sDVFS and vGreen. The results are baselined against the E+ system.

vGreen system (see section 5.5). The policy throttles the CPU only if it deems it would result in lower performance impact and higher energy savings. We refer to the system with the 'smart' policy as E+sDVFS.

Figure 5.7 shows the average weighted speedup and energy consumption results for the E+sDVFS, E+nDVFS and the vGreen system normalized against the results for the E+ system. Figure 5.7a illustrates the average weighted speedup results across all the combinations at 100% CPU utilization. The vGreen results are the same as those plotted in Figure 5.5b, but have been included for the sake of comparison. We can observe that across all the combinations, both the DVFS policies perform slower than the baseline E+ system. This is intuitive, since the DVFS policies run the system at a lower frequency. However, the E+sDVFS clearly outperforms the E+nDVFS system across all the workload combinations. While the E+sDVFS system is on an average always within 2% of the E+ system, E+nDVFS system is on average 22% slower than the E+ system. This happens, since the E+sDVFS system exploits the characteristics of the VMs, and performs aggressive throttling only on the nodes running VMs with high MPC. As discussed in the section 5.5, this results in minimal performance degradation, since such high MPC workloads are highly stall intensive and have little dependence on CPU frequency. In contrast, the E+nDVFS system naively throttles even the nodes running high IPC VMs, resulting in the high performance slowdown as observed in Figure 5.7a.

The average weighted speedups have a direct impact on the energy savings as illustrated in Figure 5.7b. The E+nDVFS system gets an average of just 1% energy savings across all the combinations. For some workloads, like *2mesa/2art*, it infact consumes more energy than the baseline system. This indicates that the power reduction due to E+nDVFS system is outweighed by the huge performance slowdown. The E+sDVFS system does better by achieving around 9% energy savings due to the small performance slowdown. However, both are clearly outperformed by the vGreen system, which achieves close to 35% energy savings. This shows that efficient resource utilization across a cluster is key to energy efficient computing in virtualized environments.
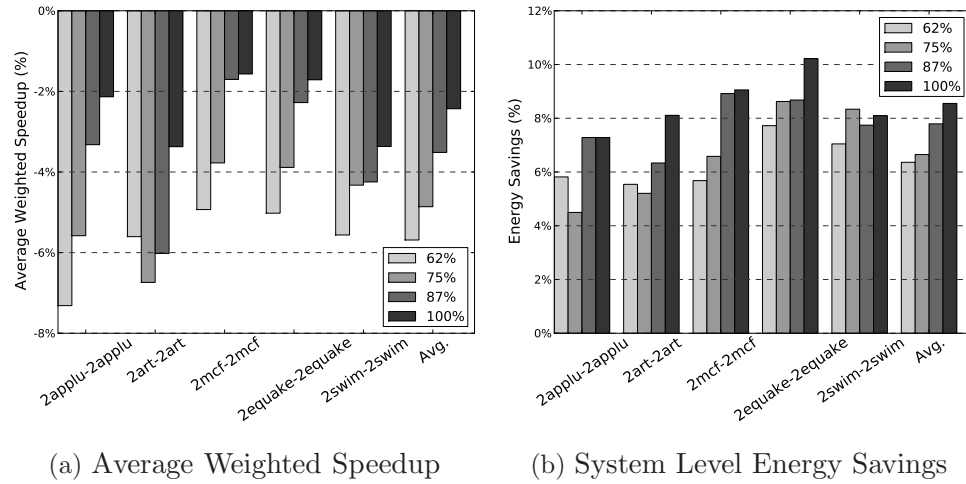
(a) Average Weighted Speedup　　　(b) System Level Energy Savings

Figure 5.8: Comparison of E+ and vGreen with homogeneous workloads. The results are baselined against the E+ system.

5.8.2　Homogeneous Workloads

We also experimented with combination of VMs running homogeneous benchmarks to evaluate the performance of our system under cases, where there is no heterogeneity across VMs. We did experiments for all the benchmarks in Table 5.2, where all the four VMs ran the same benchmark. We observed that in all the experiments, there was no possibility of rebalancing based on characteristics, since the MPC and IPC of the VMs were already balanced. However, for the case of high MPC workloads, the vGreen system effectively applies DVFS to get energy savings. Figure 5.8 illustrates the average weighted speedup and energy savings achieved across the homogeneous set of high MPC workloads. We can observe, that vGreen achieves average system level energy savings of between 6-9% across all the utilization levels. The slowdown due to DVFS is between 2-5% as indicated in Figure 5.8a. For high IPC workloads, the results were identical to E+ system, since vGreen neither does any VM migration nor DVFS.

5.8.3　Different Machine Architecture and Configurations

To verify the scalability of our system and ideas, we also experiment on a machine with a different microarchitecture and configuration. Table 5.3 compares the configurations of the two machines: Machine-1 refers to the Intel Ne-

Table 5.3: Comparison of Machines.

| Characteristic	Machine-1	Machine-2
Microarchitecture	Intel Nehalem	Intel Core
CPU	Xeon X5570	Xeon E5440
# of PCPUs	16	8
Caches	L1-L2-L3	L1-L2
Thermal Design Power	95W	80W
Memory	24GB	8GB
Memory Type	DDR3	DDR2
Memory Controller	On-Die (2.93GHz)	Off-Chip (1.33GHz)
Memory Channels	3	2

halem based machine, which we used in the earlier parts of the evaluation section; Machine-2 refers to the other machine, for which we present results in this section. A quick look at Table 5.3 shows, that the two machines significantly vary from each other in terms of CPU microarchitecture (Intel Nehalem vs Intel Core), power characteristics (different TDP), caches (L3 vs L2) as well as the memory technology (DDR3 vs DDR2 and on-die vs off-chip memory controller).

The methodology for experiments on Machine-2 was the same as that for Machine-1 with small changes: (1) We use 2GB as the memory size for each VM due to less memory in Machine-2; (2) We do experiments for 75% and 100% utilization only, since the machine has 8 PCPUs. The 62% and 87% workloads on this machine would need a total of 5 and 7 threads respectively, which cannot be evenly divided across two VMs.

Figure 5.9 shows the average weighted speedup and energy savings results across the heterogeneous set of workloads. We can observe that vGreen achieves close to 22% average weighted speedup across both the utilization levels. Similar to Figure 5.5, we can observe that for some workloads (eg. 70% for *2eon/2mcf*), the speedup is more than others (eg. 45% for *2eon/2swim*). This is again due to the fact that some workloads benefit more due to the efficient resource utilization than others due to their higher aggregate MPC (*mcf* having higher MPC than *swim* in this case). This speedup results in around 25% system level energy savings across
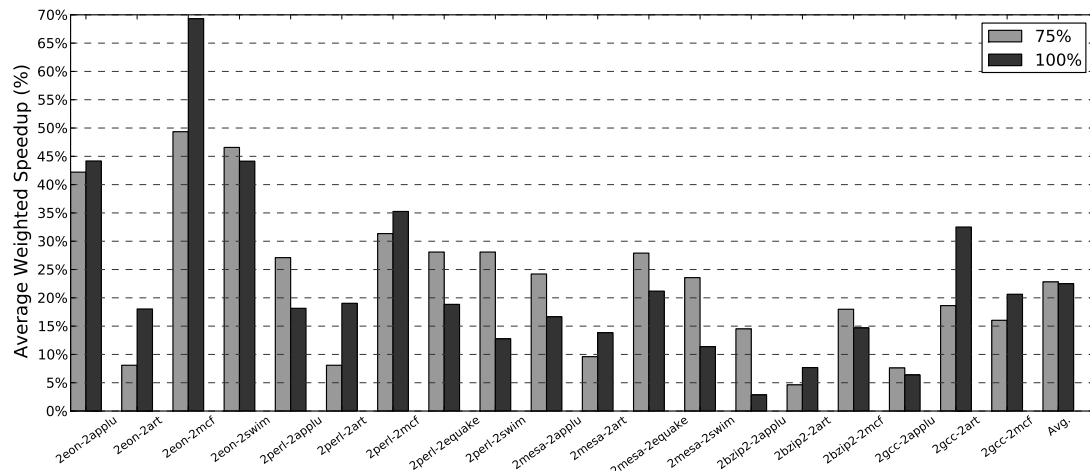
(a) Average Weighted Speedup
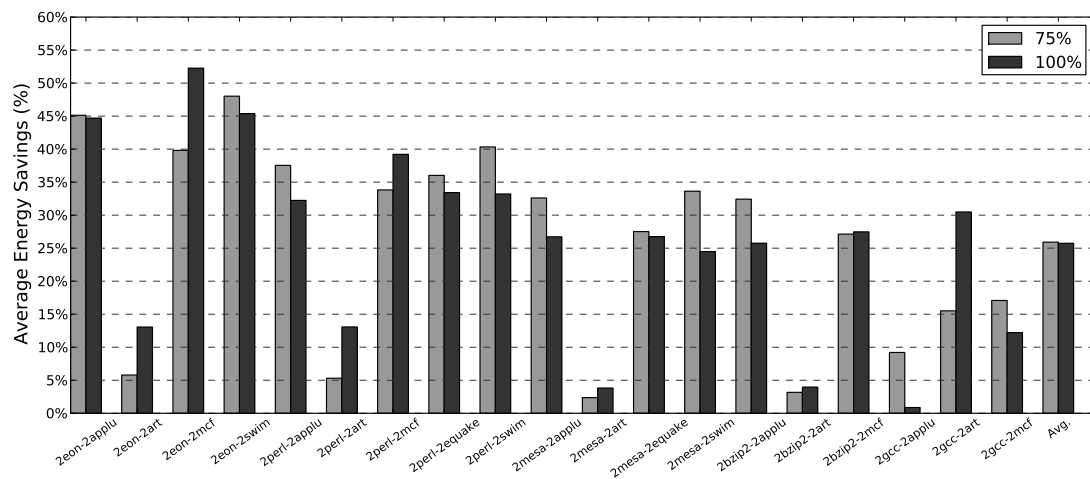
(b) System Level Energy Savings

Figure 5.9: Comparison of E+ and vGreen on the Intel Core based Machine. The results are baselined against E+ system.

both the utilization levels.

A quick comparison of overall results with Figure 5.5 indicates that both the average weighted speedup and energy savings are higher on the Machine-1 by around 10%. This is explained by the faster memory technology (DDR3), memory controller (on-die) and higher number of channels (3) of Machine-1 compared to Machine-2 (refer to Table 5.3). As a consequence of a faster memory subsystem, the high MPC workloads benefit more on Machine-1 than Machine-2, when the memory subsystem load is relieved, i.e. they run much faster on Machine-1. Since vGreen balances the aggregate MPC of workloads across the machines, it results in higher weighted speedup on Machine-1 compared to Machine-2. In future systems, the machine architectures are going to move towards even faster memory technology and architectures, and these results indicate that a vGreen like system is even more beneficial for exploiting their design, and delivering higher performance and energy efficiency.

5.8.4   Overhead

In our experiments we observed negligible runtime overhead due to vGreen. On the *vgnodes*, *vgxen* is implemented as a small module, which does simple performance counter operations and vCPU and VM metric updates. The performance counters are hardware entities with negligible cost (order of 100 cycles) on software execution as accessing them is just a simple register read/write operation. The *vgdom* executes every T_{up_period} (5s in our experiments), and as explained in section 5.5 just reads and transmits the VM metrics information to the *vgserv*. In our experiments, we observed negligible difference in execution time of all the benchmarks ($< 1\%$) with and without *vgxen* and *vgdom*.

vGreen achieves energy efficiency through VM scheduling, which requires VM migration. We observed negligible overhead of VM live migration on execution times of benchmarks, which is consistent with the findings in [23]. VM migration, however, involves extra activity in the system on both the source and the destination PMs. The primary source of activity is the processing of network packets of VM data, and processing associated with creating new VM on the destination

and cleanup on the source. However, our methodology takes all of these costs into account. As described in section 5.1, we record the performance of the benchmarks within the VMs and sample power consumption of the whole server in a power log every 2s for the entire run. If a VM migration occurs in between, the extra power consumption due to VM migration related processing (discussed above) on the network card, Dom0 and the hypervisor is taken in to account in the power log. The impact of extra processing due to VM migration on the performance of the benchmark is also taken in to account, since we record the execution times of the benchmarks. These recorded power and performance numbers are used to estimate the energy savings and average weighted speedup (equation 4) for vGreen. Hence, all our results in section 5 already incorporate these power and performance overheads, which indicate that the cost is clearly overwhelmed by the benefits of VM migration.

5.9   Conclusion

In this chapter we presented vGreen, a system for energy efficient VM management across a cluster of machines. The key idea behind vGreen is linking workload characterization of VMs to VM scheduling and power management decisions to achieve better performance, energy efficiency and power balance in the system. We designed novel hierarchical metrics to capture VM characteristics, and developed scheduling and DVFS policies to achieve the above mentioned benefits. We implemented vGreen on a real life testbed of state of the art server machines, and showed with benchmarks with varying characteristics, that it can achieve improvement in average performance and system level energy savings of up to 55% over state of the art policies at a very low overhead. We further demonstrated the applicability and scalability of the system across machines with different architecture and configurations.

However, the focus of vGreen is only on batch workloads, where the primary goal is to maximize the overall instruction throughput of the workload within a given power budget. Next chapter explores challenges involved in managing both throughput intensive batch and latency sensitive service workloads in a data cen-

ter. It shows that the vGreen system by itself is not sufficient enough to maximize the energy efficiency if there is a constraint of guaranteeing the performance objectives of the latency sensitive workloads, and introduces novel resource management algorithms to alleviate this problem.

Chapter 5, in part, is a reprint of the material as it appears in Proceedings of the 14th ACM/IEEE International Symposium on Low Power Electronics and Design, 2009. Dhiman, G.; Marchetti, G. and Rosing, T.S. The dissertation author was the primary investigator and author of this paper.

Chapter 5, in part, is a reprint of the material as it appears in ACM Transactions on Design Automation of Electronic Systems, 2010. Dhiman, G.; Marchetti, G. and Rosing, T.S. The dissertation author was the primary investigator and author of this paper.

Chapter 6

Energy Efficient Consolidation of Batch and Service Workloads

6.1   Introduction

The vGreen system maximizes both the overall energy and performance efficiency of the cluster by balancing the resource utilization of all the PMs in the cluster. However, the objective of all the workloads considered in the system is identical – higher overall instruction throughput or IPC/MIPS (million instructions per second). As discussed before, there may be some workloads running in the cluster for whom the instruction throughput is not the most important metric. For such service workloads responsiveness is more important than the actual instruction throughput achieved.

In modern data centers, if a mix of batch and service jobs are present, it is common for them to be partitioned into separate areas [42]. This is primarily because we lack a mechanism for managing the diverse performance requirements of the two types of jobs. Service jobs typically have strict response time guarantees and the cost of violating those agreements is high [42]. Batch jobs often have long-term performance targets, where immediate response is not vital. Thus, due to the high risk factor associated with the service-job performance, mechanisms for effective consolidation with batch jobs have received little attention. Recent work

on VM consolidation and resource management has primarily focused on one class of applications, i.e. either assuming only services (research in [103, 79]) or only batch jobs (the vGreen system and the research in [70, 73]) in the data center.

This chapter demonstrates that the opportunity for consolidating these diverse job types is large. Even when an application server is carefully configured to minimize energy while meeting the service level agreements of the service jobs (using, for example, the minimum number of CPU cores), there are still significant resources available to run batch jobs due to the typically diverse resource demands of the two workloads. By using these available resources, we can achieve significant throughput of the batch jobs with marginal energy costs that are much lower than running the batch jobs on a separate server. To capture this phenomenon, we introduce a new metric for heterogeneous workloads called qMIPS/Watt. This metric captures the overall work done per joule for the batch jobs, derated by any negative effect they have on the ability of the service jobs to meet their performance goals. If qMIPS/Watt is higher than a baseline system where the batch jobs run alone, we have gained overall system energy efficiency through consolidation.

We further show that state of the art techniques that focus on consolidating homogeneous workloads (like vGreen and research in [73, 79, 103, 70]) do not scale well for resource management with heterogeneous workload consolidation. This happens due to the contrasting requirements of these workloads in terms of both computation and QoS (quality of service). Towards this end we present *Themis*[1], a system for VM resource management in virtualized clusters. Themis includes a monitoring and management infrastructure which allows it to actively monitor and manage both the performance and QoS requirements of different kinds of consolidated applications within the cluster.

6.2   Batch and Service Workload Comparison

In this section, we discuss the QoS and resource requirements of the batch and service workloads, and its implication on their scheduling and management.

---

[1]Themis is the name of the Greek goddess of justice, typically depicted with scales to judge between opposing parties, which represents the contrasting workloads in the context of this work.

6.2.1   Workload Performance and QoS requirements

The objective of the two types of workloads is quite different. Batch jobs tend to have straightforward requirements and seek to maximize throughput and, therefore, minimize total job completion time. Service jobs have much more complex performance metrics, focused on the response times of a multitude of individual requests, and thus represent the bulk of our discussion in this section. An interactive service job's QoS requirement is defined in terms of end user or client response time for a request or service. However, there could be different services running in the data center concurrently, each with very different response time expectations. For instance, *Olio* has some requests that involve heavy database and file store activity, like photo uploads or event tagging, for which the expected response time is longer than for simple web browsing requests. *RUBiS* is dominated by the latter.

Expected response time requirement   In this chapter we assume that the client expected response time is defined in terms of the target 90th%ile of response time. It must be noted that the ideas presented in this chapter are independent of the target response time requirement. The value of 90th%ile is representative of realistic data center response time requirement and has been used in prior research as well [31, 69, 13]. As described above, *RUBiS* and *Olio* represent two very different sample points in this regard. For *Olio*, the target 90th%ile response times are hard-coded in the benchmark. Since the request types in *Olio* are diverse, each request type has its own target, which varies from 1s to as much as 4s. For *RUBiS*, the target response time has been set for all requests at 150ms.

Metric to monitor QoS   The expected response time, as defined above, represents the QoS requirement of a service workload. However, different service workloads in the data center can have different response time expectation or requirement (as is the case with *RUBiS* and *Olio*). To represent and monitor the QoS requirement of diverse service workloads in a uniform fashion, we further
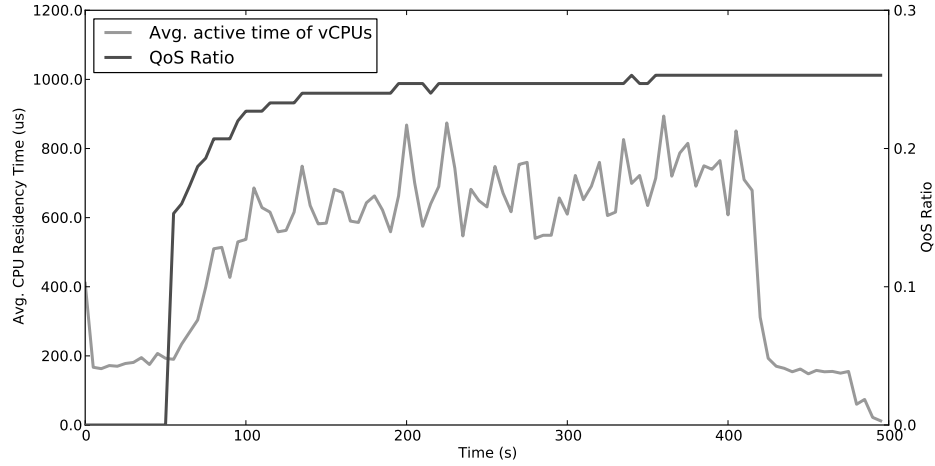
Figure 6.1: Average active time across 5s samples for *RUBiS* web server configured with 2 vCPUs.

define the *QoS ratio* as a metric:

$$QoS_{ratio} = \frac{Current_{QoS}}{Target_{QoS}} \tag{6.1}$$

where the current and target QoS requirements are application specific targets in terms of the 90th%ile response time. This metric has two advantages – (1) as discussed above, it allows us to combine varied response time expectations in a single ratio, and (2) gives us a continuous metric (as opposed to a binary pass/fail QoS result) which tells us how much slack we have to work with before we start failing to meet the requirements. If this ratio exceeds 1, it implies that the QoS requirement has been violated.

For the batch jobs, the goal is typically to maximize executed instruction throughput (e.g., million instructions committed per second or MIPS) or minimize the overall execution time.

Workload Resource Usage   The way services and batch workloads use the CPU resources tends to vary significantly. For interactive services, the primary CPU usage is in accepting and servicing user requests. Since most of the servers are implemented as multi-process or multi-threading modules, the number of active processes/threads is directly proportional to the number of concurrently active
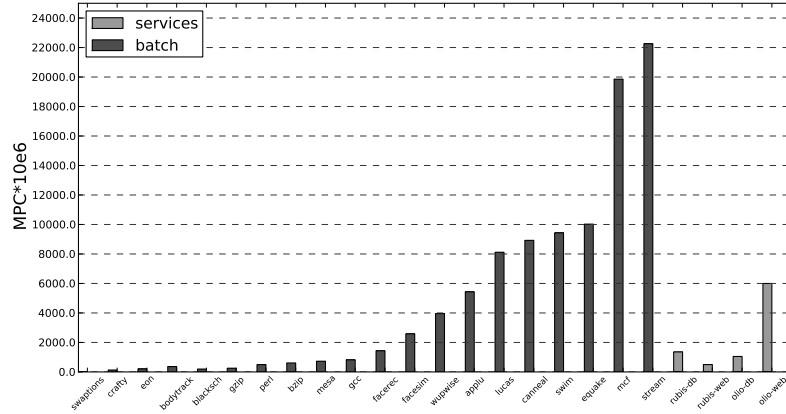
Figure 6.2: Memory accesses per cycle (MPC) for different batch and service applications.

requests, which in turn determines the CPU usage at any point in time. Service threads tend to be I/O intensive with short occupancy times on the CPU.

Figure 6.1 plots samples of average active time of all vCPUs of a service VM running *RUBiS* web server (configured with 2 vCPUs) averaged over 5s periods on the first y-axis. The second y-axis plots the QoS ratio achieved by *RUBiS* over the course of the experiment. The experiments were done on a dual quad core Intel Nehalem based server machine. The 'active time' is the duration for which a vCPU actively runs on a physical CPU before going idle – it goes idle when all queued jobs have completed or are stalled for I/O (disk, communication, etc.). Each sample point in the plot represents the averaged active times over 5s time intervals. We collected this information using SystemTap [83], a Linux tool to dynamically instrument the kernel. We used the context switch probes of SystemTap to gather the information shown in Figure 6.1.

We can observe that the QoS ratio is comfortably met for this run (i.e., is less than 1). A close look at the figure indicates that the CPU activity of the VM is very fine grained, running very short jobs. The average activity period for vCPUs is around 700μs. The CPU occupancy time for individual requests was even lower, as the active time typically includes the time for multiple requests to be handled before the runqueue is emptied. When we changed the number of vCPUs to 3 and 4, the average activity time went down almost proportionally.

With 4 vCPUs the average active time dropped to around $350\mu s$ – runqueues are shorter and empty more quickly. This resulted in a reduction in QoS ratio, as well (to ∼0.15). This implies that the service workload is easily distributed, and in fact will quickly adapt to changes in the number of vCPUs – for example if we wish to reduce the QoS ratio by increasing vCPUS, or possibly increase energy efficiency by decreasing vCPUs when sufficient QoS ratio headroom exists.

In contrast, the CPU usage pattern of the batch workloads we are running is very different. They are primarily throughput and CPU intensive, and they rarely relinquish the CPU. They make extensive use of pipeline, cache and memory bandwidth. The batch jobs do vary significantly in how they stress the memory hierarchy, which will determine in large part how much they interfere with other jobs (e.g. service job threads) when running. Figure 6.2 shows the memory accesses per cycle for VMs running different batch and services applications alone. We see that the batch jobs can differ by more than an order of magnitude in this respect. For instance, in our experiments we observed batch jobs like *streamcluster* and *mcf* to have 5-6x more memory accesses per cycle (MPC) compared to *RUBiS* and *Olio* web and database servers. At the same time for workloads like *perl* and *bodytrack* the MPC was 5-6x lower than the services. Thus, while the batch jobs tend to occupy the CPU at full utilization, they vary widely in whether the CPU is spending most of its time on computation or waiting for memory. Additionally, unlike the services, they do not adapt to make use of additional resources, if they are already sufficiently provisioned. For instance, if we run a single instance of our batch workload in a VM with either 1 or 2 vCPUs, it would not affect the MIPS it commits. This is in sharp contrast to what we observed for the service jobs above, where adding more vCPUs changes the way they are utilized and also affects the QoS ratio.

6.3   Energy Efficiency of Diverse Workloads

Increased resource utilization and energy efficiency through VM consolidation is a big reason virtualization has gained prominence in modern data center
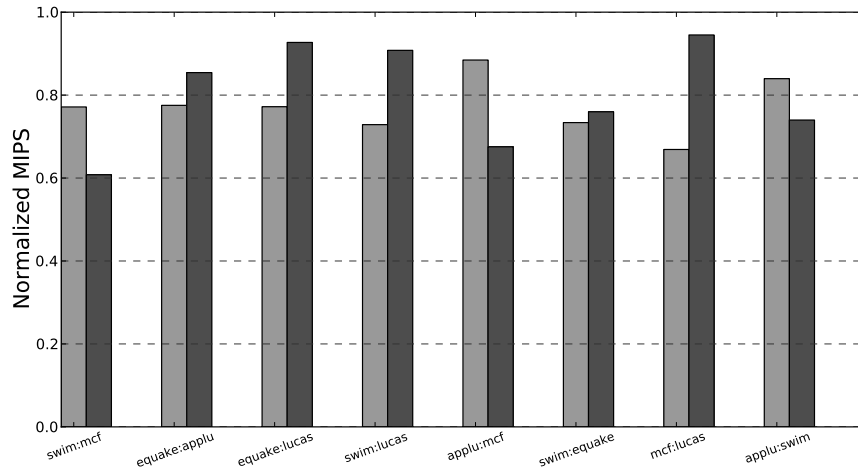
Figure 6.3: Impact of batch VM consolidation on MIPS. The plot shows the MIPS of each workload in a consolidated combination normalized against the MIPS it had when running alone. For each combination b1:b2, the gray bar shows the normalized MIPS of b1 and the black bar shows the normalized MIPS of b2.

deployments. The energy efficiency is largely a consequence of the fact that modern servers have a very non-energy proportional profile [9, 42], i.e. their power consumption does not scale down linearly with decreasing utilization. Even at close to 0% utilization, the systems consume nearly 50% of what they consume at 100% utilization [9] due to a number of non-linear, non-energy proportional components like power supply, memory, etc. Thus, higher utilization rates represent the most energy efficient zone for these servers. This implies that aggressive consolidation of VMs in a cluster not only frees up machines (which could be either moved to low power states or used to do additional work), but also pushes the active machines towards operation in the more energy efficient zone.

6.3.1 Why consolidate batch and services VMs?

Most resource management and VM scheduling algorithms focus on only one class of workloads. They may be used to manage heterogeneous workloads as well by independently consolidating server and batch VMs on separate machines. However, this can lead to poor performance and bottlenecks as we show next.
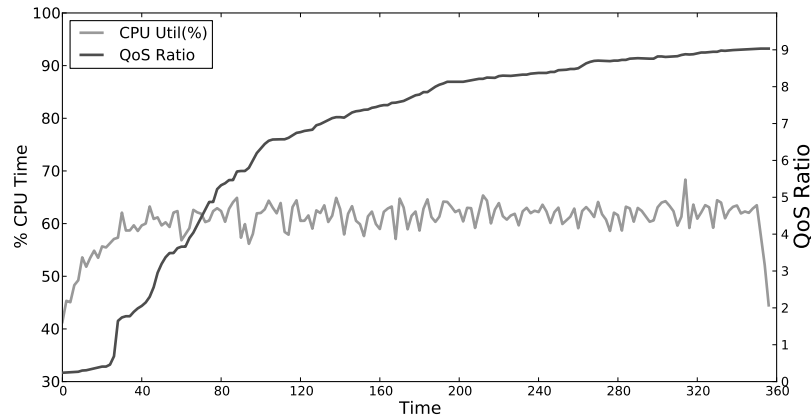
Figure 6.4: Impact of I/O bottleneck on the QoS ratio of *RUBiS* web server.

Poor throughput due to batch VM consolidation Batch applications tend to use the PM resources very intensively. Since modern machines have shared resources across the hierarchy such as last level cache, memory bandwidth etc., it can lead to severe contention, which can significantly deteriorate the MIPS achieved by the batch VMs – even when the batch jobs do not share a CPU core. We ran experiments that consolidate memory intensive batch VMs running different workloads, and estimate the MIPS of the VMs when running alone and then run as part of the combination. Figure 6.3 shows the MIPS of workloads when running in consolidated environment normalized against the case when they run alone. In some cases the drop in MIPS due to the consolidation is very high (as high as 40%), even though they are running on separate cores of the machine. This is inline with the observations and analysis in the previous chapter when memory intensive VMs were co-scheduled on a PM (see section 5.4).

I/O bottleneck from services consolidation Services VMs share the CPU and memory resources of a PM well. First, services VMs are on the modestly intensive side while using shared resources, as indicated by the MPC values shown in Figure 6.2. Second, they multiplex the CPU resources very well among each other due to the small residency time for each request as indicated in Figure 6.1. However, the primary problem in putting them together comes from the I/O resources (like network) they share on the PM. Figure 6.4 shows an example of the

impact of a network bottleneck on QoS. In this example, we run two similar *RUBiS* web-servers on two different sockets of the same machine, so that they do not share any CPU or memory resources. We can see that although ample CPU headroom is available ($\sim$40%), the QoS is severely compromised due to the network bottleneck. However, when running alone the same web-server has a similar CPU utilization of around 60% and comfortably meets the QoS.

This study motivates consolidation of services and batch VMs, since their heterogeneous resource usage profiles (see section 6.2) complement each other – neither workload seriously exacerbates the bottleneck resource of the other. The next section addresses the challenges in accomplishing heterogeneous VM consolidation.

6.3.2   Diverse Workload Challenges

There are many potential challenges to creating a system that effectively merges both interactive and throughput-oriented workloads, from the difficulty of predicting the interactions between the two classes of jobs when consolidated to the inability of current software systems to properly time-share the system in the presence of heterogeneous jobs. In this section, we will discuss the challenges in consolidation of services and batch workloads.

Lack of QoS support in Proportional Share Schedulers   Most of the modern hypervisor schedulers are based on proportional sharing of CPU resources. For instance, Xen makes use of a credit scheduler, where each VM's proportional share is specified through a 'weight'. The minimum value of the weight is 256, while the maximum possible is 65535. Based on the weight, the CPU resources (or credits) are distributed to the virtual CPUs (vCPUs) of the VMs in proportion to their weight, and the vCPU priority (or task priority) is also recalculated based on the credits the task has. There are three priority levels: (1) 'Over': When a task exhausts its credits, its priority is set to 'over', which represents a low priority. (2) 'Under': When it still has credits, it is set to 'under', which represents a higher priority. (3) 'Boost': To accommodate low latency, the scheduler supports a boost
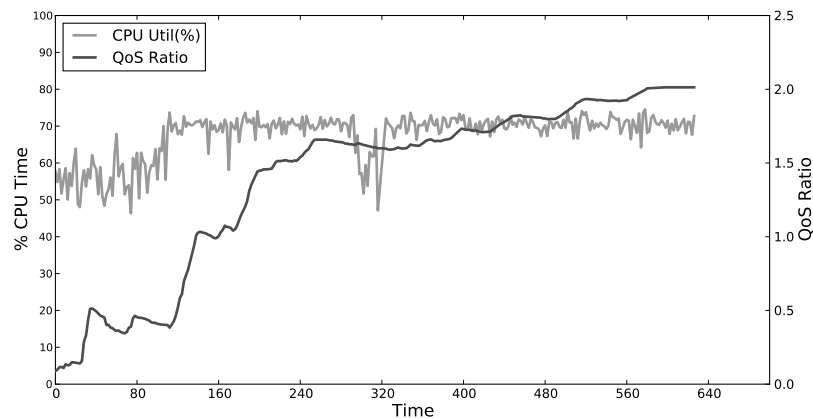
(a) *Olio*-db without QoS priority state
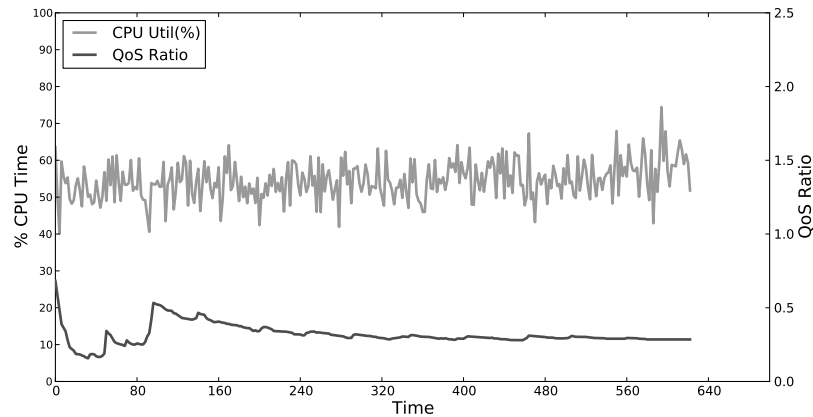
(b) *Olio*-db with QoS priority state

Figure 6.5: Illustration of lack of QoS support in Xen.

priority, where blocked task waiting for an I/O event are boosted upon receiving an event/interrupt. It has the highest priority.

Such a model works well if VM workloads use the CPU resources in a homogeneous fashion. However, the coarse grained priority levels of the scheduler are not designed to accommodate urgency of CPU requirement, which is very important for services. From the discussion in the section 6.2, we know that the CPU residency times of services workloads is very low (see Figure 6.1). But at the same time it is important for them to get the CPU when they are ready to run. The mechanisms provided by the proportional schedulers fail to achieve that,

since proportional sharing only promises a higher proportion of CPU without any timing guarantees. Figure 6.5(a) illustrates the problem with a plot of samples of CPU utilization and QoS ratio taken every 2s of the run. We run a 2 vCPU batch VM with *equake* as the workload and a 4 vCPU service VM running *Olio* database and share the same quad-core socket. We give the lowest weight of 256 to the batch VM, and the highest possible 65535 weight to the service VM. We can observe in Figure 6.5a that despite having ample CPU headroom, and no resource bottleneck, the QoS ratio of *Olio* is poor. This happens due to lack of guarantee of timely access to CPU resources for the service VM.

The lack of QoS support in proportional schedulers has been identified as a problem in conventional OS schedulers like Linux (Stanford SMART scheduler [75] and QLinux [93]) as well as hypervisor schedulers like Xen in recent research [64]. To solve this problem, we adopt a solution similar to the QLinux [93] approach by introducing real time scheduling for services, while retaining default proportional scheduling for batch VMs. We introduce a new priority state in the scheduler, referred to as the 'QoS' state, which sits between the Boost and Under states. The VMs with QoS priority state always get a real time priority and get scheduled ahead of other non QoS state VMs. The QoS state can be configured through Xen-API for any VM, which has a QoS requirement. Figure 6.5b shows the QoS ratio results for the same experiment with the QoS state enabled for *Olio* VM. We can see that it comfortably meets the QoS requirement, since the scheduler gives a real time priority to the service vCPUs whenever they are ready to run. However, as we show next, providing real time priority is not sufficient by itself to guarantee QoS for the services under consolidated virtualized environments.

Impact of Interference Effects Most of the state of the art VM consolidation algorithms [103, 41] assume that when VMs get consolidated, their CPU utilization adds up. This is true for batch and batch consolidation [73], since their CPU utilization is fairly static, as well as services and services, as they contend more on I/O resources rather than CPU or memory resources. However, when we consolidate batch VM and service VM, the assumption does not necessarily hold true. Due to high utilization of shared resource like memory bandwidth and last level cache, the
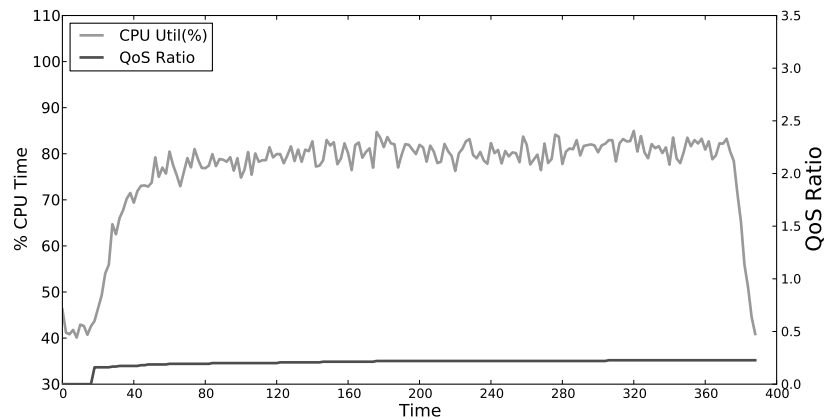
(a) *RUBiS*-web running alone
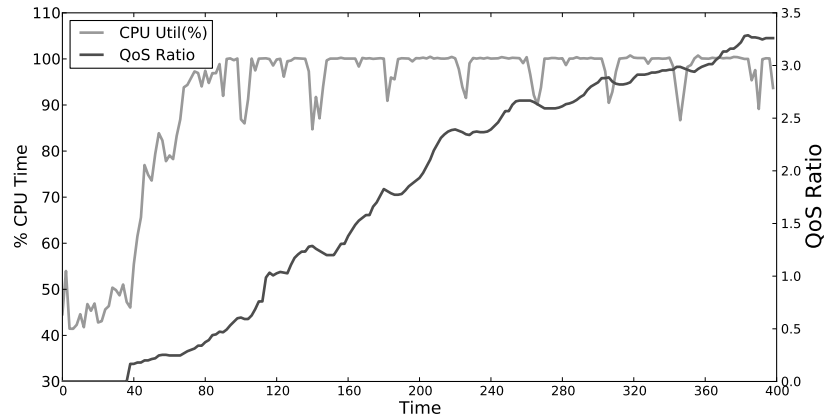
(b) *RUBiS*-web with batch VM running '*swim*'

Figure 6.6: Impact of interference effects on the QoS ratio of service VM (*RUBiS* web server).

batch VM can slow down the service VM and hence increase its CPU utilization to possibly generate a CPU bottleneck. Figure 6.6 illustrates an example of this phenomenon with a plot of CPU utilization and QoS ratio samples collected every 2s. In Figure 6.6a, we run *RUBiS* web server configured with 2 vCPUs on the quad core Nehalem machine, and we can observe that the application comfortably meets the QoS ratio. In Figure 6.6b, we repeat the same experiment with a batch VM (with 2 vCPUs) running *swim* as the workload consolidated on the same quad core machine. We assign the QoS state to the *RUBiS* web server to guarantee

timely access to the CPU resources. The interference effects due to the batch VM slows down the web server, even though they do not share a physical core, only portions of the memory hierarchy and interconnect. This consequently increases the CPU utilization of the web server VM, and creates a CPU bottleneck where there previously was none, and worsens the QoS ratio of the application.

This example shows that just guaranteeing real time priority is not sufficient to ensure QoS for service VMs in consolidated environments. The interference effects due to shared resource usage can dramatically impact the QoS level even when CPU resources are not shared, and must be explicitly accounted for. These interference effects are a function of how the workloads interact with each other, and as we show in later sections, is difficult to model. Consequently, our approach, as described in the next section, is to rather infer the interference effects through CPU utilization and QoS ratio feedback from the service VMs, and to adaptively provision their resource allocation in order to alleviate the problem.

6.4 System Design

In this section we provide details of the design and implementation of our system, Themis, for managing diverse workloads in the data center. The objective of the system is three-fold – (1) satisfy the QoS requirement of the service jobs; (2) maximize the batch job throughput; (3) minimize the power consumption. To capture all these goals, we define a new metric to capture the system level energy efficiency.

6.4.1 Energy efficiency metric

In a data center running a heterogeneous workload mix comprised of both batch and services VMs, VM consolidation policies may co-schedule any combination of workloads across the physical machines. However, to understand the effectiveness of these policies, it is extremely important to take into account how it maximizes the batch job throughput, while ensuring the QoS requirements of services are met while minimizing the energy cost. To quantify energy efficiency

in such environments, we introduce a new metric. It measures the batch job throughput/Watt, multiplied by a factor that reflects whether we are still meeting the services QoS requirement. The point being that batch throughput only has value when it still allows services agreements to be met.

$$qMIPS/Watt = q * (batchjobMIPS)/SystemWatts \qquad (6.2)$$

Here, q is either 0 or 1, depending on whether the QoS ratio is below 1. Since the batch job performance is being measured in terms of MIPS, equation 6.2 actually represents work done per Joule over a period of 1s ($Joule = Watt \times time$). For instance, a value of 10 qMIPS/Watt over a time interval implies that an average 10 million batch job instructions were committed, while meeting the QoS requirement of the service VM, using 1 Joule of energy per second.

Maximizing qMIPS/Watt implies that it is acceptable to sacrifice some services performance as long as we still meet our strict service level agreements, increase batch throughput, and increase the overall energy-efficiency of the system.

6.4.2   Themis Design

The overall objective of Themis is to manage diverse workloads in the data center with the goal of maximizing qMIPS/Watt. For this purpose, the system implements a monitoring framework for dynamic VM profiling and policies for dynamic resource management of VMs. Figure 6.7 gives an overview of Themis. The overall design is similar to vGreen and is composed of three primary entities: (1) Themis Nodes: These are the physical machines in the data center that run the actual workloads, which can be a batch VM, service VM or both. They are similar to *vgnodes* in vGreen. (2) Services Clients: These are the machine(s) that are running applications requesting service from a service VM (which can be a single or multi-tier service) running on the Themis clients. (3) Themis Server: This is the cluster manager and is responsible for implementing policies for node level resource management and VM scheduling. This is similar to the vGreen *vgserv*. We now present these entities in greater detail.
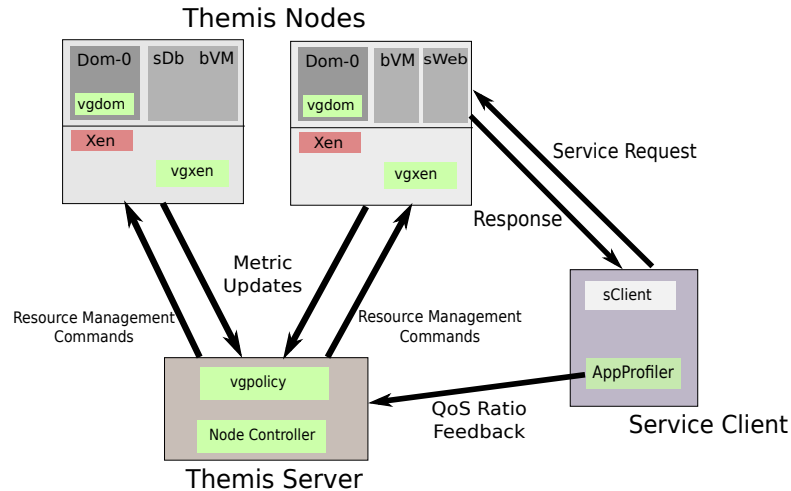
Figure 6.7: Overall Themis Architecture.

Themis Nodes

Themis nodes (referred to as tNodes hereafter) are the physical machines populating the data center for running the actual workloads. The workloads could be either a batch VM or a service VM as shown in Figure 6.7. The tNode contains *vgdom* and *vgxen*, which are the same as in the vGreen system.

Services Clients

Services clients (referred to as sClients hereafter) are the applications being serviced by the service VM(s) running across the tNodes. Figure 6.7 shows an example of a sClient being serviced by a multi-tier service (like *RUBiS* and *Olio*) comprised of web and database servers (sWeb and sDb) running across two tNodes. The sClients use appProfiler to dynamically communicate QoS ratio to the Themis server. We instrumented the workload generators of *RUBiS* and *Olio* to implement the appProfilers. We assume that it is feasible to implement such appProfilers for all the services running in the data center. The QoS ratio is dynamically communicated by the appProfiler to the Themis server.

Themis Server

Themis server (referred to as tServer hereafter) is the cluster manager, and is responsible for resource management across the cluster with the objective of maximizing the qMIPS/Watt. The tServer registers all the tNodes and sClients, and periodically collects the metric updates and QoS ratio from them, and feeds it to the management policies running on the system. The policies convey their management decisions to the *vgdom*, which physically implements them on the intended tNode as illustrated in Figure 6.7.

The tServer employs *vgpolicy* just like vGreen *vgserv* that now consolidates batch and service VMs based on the CPU utilization metrics of the individual VMs running across the cluster provided by the *vgdom*. However, as identified in the previous section, when we co-locate batch VM and service VM on a tNode, the interference effects can impact the QoS ratio of the services. For solving this problem Themis employs a novel resource management policy, referred to as the Node Controller (see Figure 6.7), which exploits the heterogeneity of the batch and service workloads.

Node Controller (tController)   As identified in the previous sections, the CPU requirements of the batch and service workloads are complementary. While batch VMs need CPU resources to maximize their throughput, the service VMs only need the CPU resources for long enough to service the client request within the required time frame. The batch VMs do not benefit from additional CPU resources if their requirement is already met, while the service VMs can benefit from additional CPU resources by spreading their workload more. Consequently, if we can dynamically shrink or expand the CPU resources for the service VMs to converge to a value where the QoS is just satisfied then we can safely commit the remainder to the batch VMs. This observation motivates the design of a resource management policy, referred to as the 'Node Controller' or the'tController' to dynamically control the CPU resource allocation to the service VMs.

The objective of the tController policy is two-fold: (1) Converge to the optimal number of vCPUs sufficient to meet QoS ratio for the service VMs, while

Table 6.1: Performance Model (n_{ref} Estimation).

Input: QoS_{cur}, $util_{cur}$ and n_{cur}

1: **if** $QoS_{cur} < QoS_{th}$ **then**
2:    $n_{next} \leftarrow n_{cur} - 1$
3:    $util_{next} \leftarrow (util_{cur} \times n_{cur})/n_{next}$
4:    **if** $util_{next} > util_{th}$ **then**
5:       return n_{cur}
6:    **end if**
7: **else**
8:    $n_{next} \leftarrow n_{cur} + 1$
9: **end if**
10: return n_{next}

---

maximizing the CPU headroom for batch VMs; (2) Guarantee a stable system. We implement tController using a formal feedback control which is provably stable and can meet the desired objectives. The tController operates at discrete time intervals or periods, when it is periodically fed with metric and QoS ratio updates from the tNodes and the sClients respectively. Every interval, the tController assigns vCPUs to the VMs in a way that guarantees convergence to their desired QoS ratio. There is a tController for every service VM, and operates independently based on the metric and QoS inputs.

The tController is based on a performance model that tries to predict at every time interval what is the projected requirement in terms of number of CPUs (which we refer to as n_{ref}) for the next interval based on the CPU utilization and QoS ratio input. We describe it below:

Performance model Table 6.1 describes the algorithm to estimate n_{ref} at every time interval (for the upcoming interval). The algorithm takes as the input the current QoS ratio (QoS_{cur}), CPU utilization ($util_{cur}$) and the vCPU allocation (n_{cur}) for the service VM the tController is managing. In step one of the algorithm, it compares the QoS_{cur} with the QoS_{th}, which is a threshold to determine if the
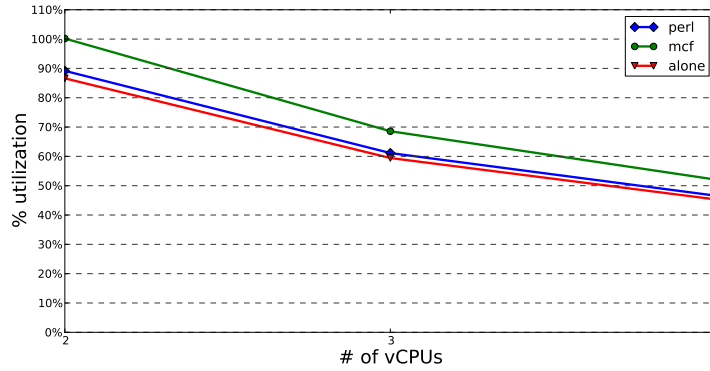
Figure 6.8: Approximate linear scaling of CPU utilization with vCPUs for *RUBiS* web server.

QoS ratio is progressing towards 1. If QoS_{cur} is below the threshold, then the algorithm concludes that the service VM is currently over-provisioned in terms of CPU resources. Consequently, in steps 2 and 3 it tries to predict what its CPU utilization would be, if it reduced the vCPUs allocated to the service VM by 1. The prediction (step 3) uses a linear extrapolation model based on $util_{cur}$, n_{cur} and n_{next} (vCPU allocation for next interval). If the predicted CPU utilization ($util_{next}$) is greater than a CPU utilization threshold ($util_{th}$), which is representative of a bottleneck (step 4), then the algorithm returns n_{cur} as the n_{ref} (step 5). If that is not the case, then the algorithm returns n_{next} as the n_{ref} (step 10). The motivation for using linear scaling is based on modeling of CPU utilization of service workloads from real experiments running with and without interference effects. While the model is simple, Themis' control system allows it to adapt to errors in the model. For instance, Figure 6.8 shows the CPU utilization of a *RUBiS* web server configured with 2, 3 and 4 vCPUs running identical load. The three plots correspond to runs where the web server runs alone (no consolidation), and with batch VM running *perl* and *mcf* as workloads. We can clearly see that in each case the CPU utilization can be roughly modeled as linear with the change in number of vCPUs both with (*mcf*) and without/little (alone and *perl*) interference.

If QoS_{cur} is greater than QoS_{th} (step 7), then the system concludes that the current CPU allocation for the service VM is not enough to meet the QoS ratio.

Consequently, it adds 1 more vCPU on top of the current allocation, and returns it as n_{ref} (steps 8 and 10). The performance model described above is simple, but is not precise due to the following reasons: (1) The model assumes the CPU usage profile of the service VM stays constant, which might not always be true. (2) The characteristics and the interference effects of the co-located batch VM might also change, resulting in imprecise predictions.

Controller It is important to keep track of the deviations in the behavior of the performance model to converge to a desirable and stable allocation. These deviations aggregate over time and they can be modeled as a state. We model the error in selecting the number of vCPUs for each service VM as a state which is related to the input, the number of vCPUs, as below:

$$\delta n(k+1) = \delta n(k) + (n(k) - n_{ref}(k)) \tag{6.3}$$

where $n(k)$ is the currently assigned number of vCPUs and $\delta n(k)$ represents the error accumulated in selecting the number of vCPUs until the current time or the k_{th} interval. The value of $n_{ref}(k)$ is the target number of vCPUs estimated using the performance model described above (refer Table 6.1). In this equation, the value of $\delta n(k+1)$ represents the sum of the accumulated errors over the past k time intervals ($\delta n(k)$) and the error generated during the period between k and $k+1$ which equals $n(k) - n_{ref}(k)$. To maximize the qMIPS/Watt, we must guarantee the convergence of $\delta n(k+1)$ to zero, since that would imply we give minimum possible CPU resources to the service VM at any point in time to meet its QoS ratio, hence maximizing the achievable MIPS for the batch VM. This model can be easily extended to multiple service VMs where each VM needs to meet its respective QoS ratio.

We use closed loop feedback to dynamically manage the number of assigned vCPUs with the objective of converging the value of $\delta(k)$ to zero. We achieve this by using the control law which is the linear feedback of the states under control:

$$n_i(k) = -G_i \delta n_i(k) + n_{ref_i}(k) \tag{6.4}$$

where G_i is the state feedback gain for the ith service VM and $n_i(k)$ is the number of vCPUs for that service VM to meet the desired QoS ratio. The

controller is stable when the gain values are within the range of $1 > G_i \geq 0$ where the value of G_i determines the convergence time of the controller.

The controller is guaranteed to converge to the optimal number of vCPUs for the service VMs, which can satisfy its QoS requirement. For instance, if we consider the example of Figure 6.6, the controller would be able to identify the CPU bottleneck through the feedback of CPU utilization and QoS ratio, and would increase the vCPU allocation in response, to resolve the bottleneck. However, the timeliness of such a response is important to ensure that the QoS ratio always stays below 1. This is a function of the granularity of identifying the bottleneck, which is determined by the interval length (T_s) across which we collect metrics and QoS ratio feedback and QoS_{th}. If T_s is too coarse such that the system is not able to capture worsening QoS ratio due to a bottleneck between two consecutive QoS ratio samples, it might lead to QoS violations. On the other hand, if it is too fine, it would cause avoidable overhead. The QoS_{th} is the function of T_s and the worst case rate of QoS ratio degradation in the presence of a bottleneck. We determined these parameters through experiments with micro-benchmarks designed to create bottlenecks for the service VMs, which allowed us to capture the worst case worsening rate of QoS ratio. We verify that different sets of these parameters work well for our system, but use QoS_{th} of 0.6, T_s of 2s and $util_{th}$ of 95% as representative parameters in our experiments.

6.5 Evaluation Methodology

We conduct experiments on a testbed of four state of the *art* dual Quad core Nehalem based machines with 24GB memory. Two machines act as the tNodes, and run the service and batch VMs. The third machine, used as the sClient, generates the workload for the services. The sClient is representative of the end users of the services that can be located either inside or outside of the datacenter. The last machine acts as the tServer and is responsible for scheduling and resource management of the VMs.

The Cluster Scheduler on the tServer consolidates the VMs on the basis of

either CPU utilization [103, 41], workload characteristics in terms of MPC [28, 70], or does not consolidate at all to avoid interference effects. The Node Manager on the tServer can dynamically alter the resource allocation of VMs in terms of CPU cap available [73, 79] or number of vCPUs allocated to ensure that the QoS ratio is satisfied for the service VMs. The primary policy of Themis is consolidation based on CPU utilization, and resource management based on the tController, as discussed in the previous section. In order to evaluate Themis against existing state of the *art* systems, we consider the following policies:

(1) **Baseline:** This policy runs the service VMs and the batch VMs on separate tNodes to avoid any interference effects.

(2) **Consolidation:** This policy consolidates the service VMs and the batch VMs on the basis of CPU utilization [103, 41] (if the sum of their CPU utilization is less than $util_{th}$, the parameter that indicates CPU bottleneck in Table 6.1) but does not perform any resource management using the Node Controller.

(3) **Ideal-tChar:** vGreen and similar systems proposed in [70] are capable of identifying the memory intensiveness of the batch VMs, and employ dynamic VM scheduling policies that try to distribute VMs across physical machines to avoid MIPS degradation that can happen due to their consolidation. However, these policies assume there are only batch workloads running in the cluster, and focus on maximizing the overall MIPS/Watt. We extrapolate such policies for Themis so that they only consolidate non memory intensive batch VMs with the service VMs. If the batch VM is memory intensive, they let the VMs run independently on separate tNodes. Such a policy ensures the QoS ratio is satisfied by limiting the degree of consolidation. We refer to this policy as 'tChar'. We pre-pend it with Ideal, as we make it perform consolidation only if it is guaranteed that the QoS ratio is satisfied, which we determine offline. It does not perform any dynamic resource management of the service VMs using the Node Controller.

(4) **Ideal-tCap:** Systems proposed in [79, 73] are capable of dynamic resource management to manage QoS on consolidated machine through CPU capping. They do so by placing a CPU cap on the lower priority VM to ensure that the higher priority VM meets its QoS requirements. For instance, a cap of 40%

on a low priority VM implies that it cannot run more than 40% of the time even if it is runnable and has CPU resources available. A CPU cap limits the amount of time the lower priority VM can spend running, hence reducing the interference effects. This policy can be interpreted in our system as placing a CPU cap on the batch VM based on the QoS ratio feedback of the application being serviced by the service VM to ensure that the QoS requirements are met. We refer to this policy as 'tCap', where it first consolidates the batch and service VMs based on their CPU utilization [103, 41] and then uses CPU capping to ensure that the QoS ratio of the service is satisfied. We again refer to it as the Ideal, since we determine the minimum cap required to be imposed on the batch VM so that the QoS ratio is satisfied offline, and use it for this policy.

(5) **Controller:** This is the default Themis policy. It consolidates batch and service VMs based on their CPU utilization [103, 41], but uses the tController described in the previous section for QoS ratio management.

For all our experiments, initially one tNode runs the service VMs (for instance the *RUBiS* web and database servers) and the other tNode runs the batch VM. We then monitor the MIPS of the batch VM, QoS ratio for the service VMs and the power consumption of the active tNodes every two seconds (T_s) for the whole duration of the run of the sClient, i.e. the workload generator. The power consumption is recorded from the power sensors on the machines, which are accessible through an Integrated Power Management Interface (IPMI) [55]. Using these numbers, we estimate the qMIPS/Watt for a particular run. We use only the power consumption of the active tNodes for qMIPS/Watt estimation, since that captures how energy efficient the policies are in terms of resource usage. We refer to it as the 'active power consumption' for a given run.

We report results for all these policies across different workloads running inside the batch VM. For fairness, the initial configuration of the batch and service VMs is identical for all policies. All the VMs run Linux as the guest OS, and are configured with 2 vCPUs and 4GB of memory. The service VMs are assigned the 'QoS priority state' (see section 6.3) with all the policies to ensure timely access to the CPU. The *RUBiS* and *Olio* workload generators are configured such that the

(a) *RUBiS* and Batch VM

(b) *Olio* and Batch VM

Figure 6.9: Comparison of all the policies for overall qMIPS/Watt. The results are normalized against the Baseline policy.

service VMs comfortably meet the QoS ratio when running alone (*RUBiS*: 7000 users, *Olio*: 550 users). For the batch VM, we always have as many threads of the benchmark as the number of vCPUs to represent a fully utilized VM. The mix of our batch jobs has an equal number of CPU and memory intensive benchmarks.
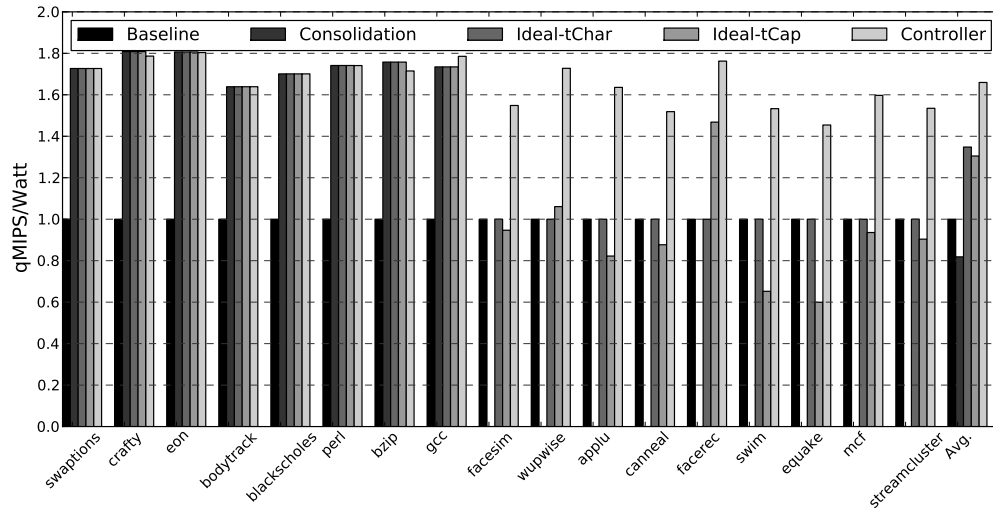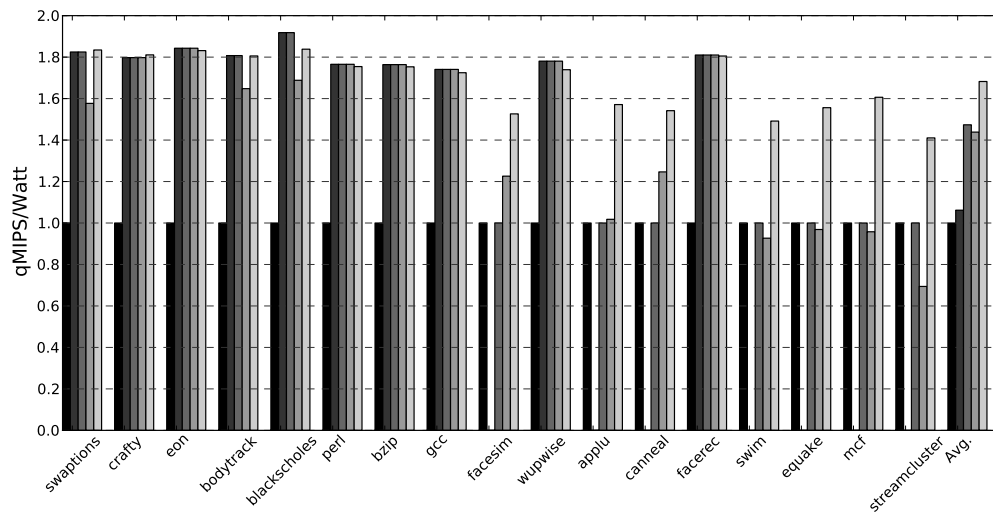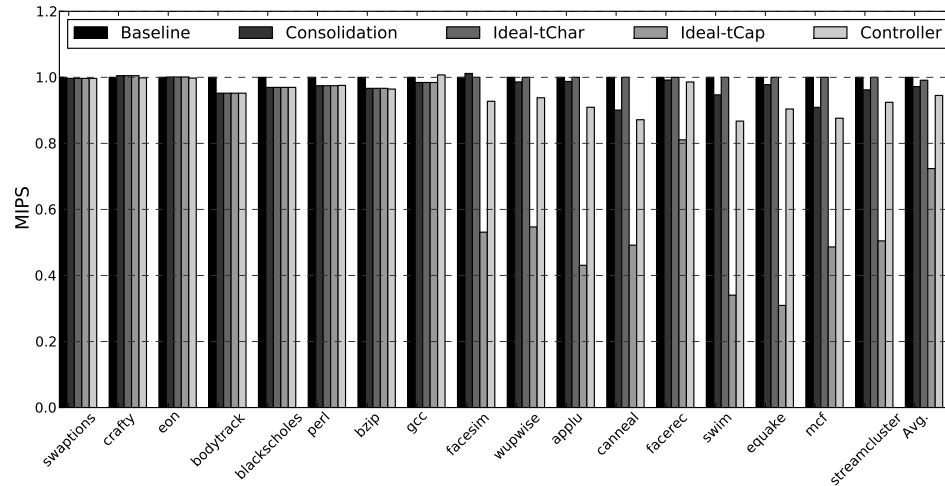
(a) *RUBiS* and Batch VM
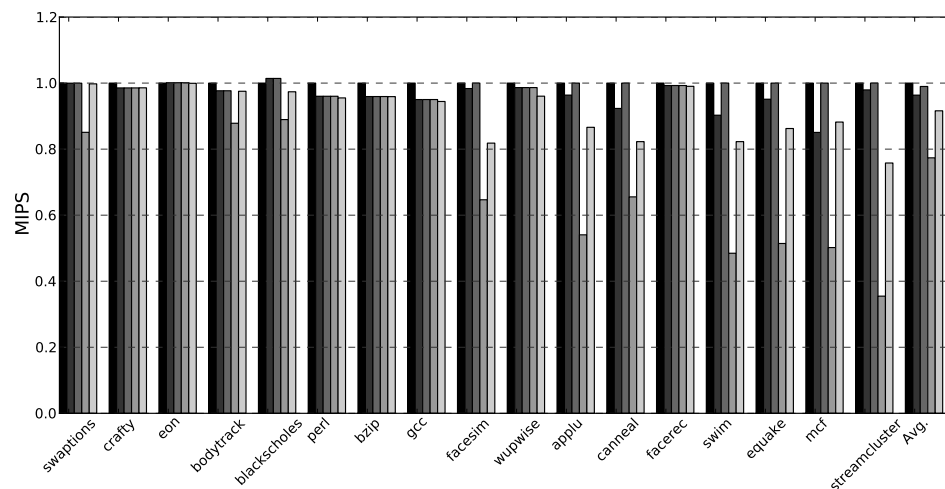
(b) *Olio* and Batch VM

Figure 6.10: Comparison of all the policies for Batch VM MIPS. The results are normalized against the Baseline policy.

6.6   Results

Figures 6.9 illustrates the overall results in qMIPS/Watt for all the policies, normalized against the baseline policy, with different workloads running inside the batch VM. The different combinations are sorted from left to right according to the level of expected interference between service and batch VMs. The more memory intensive the batch job, the more it impacts the execution of service jobs.
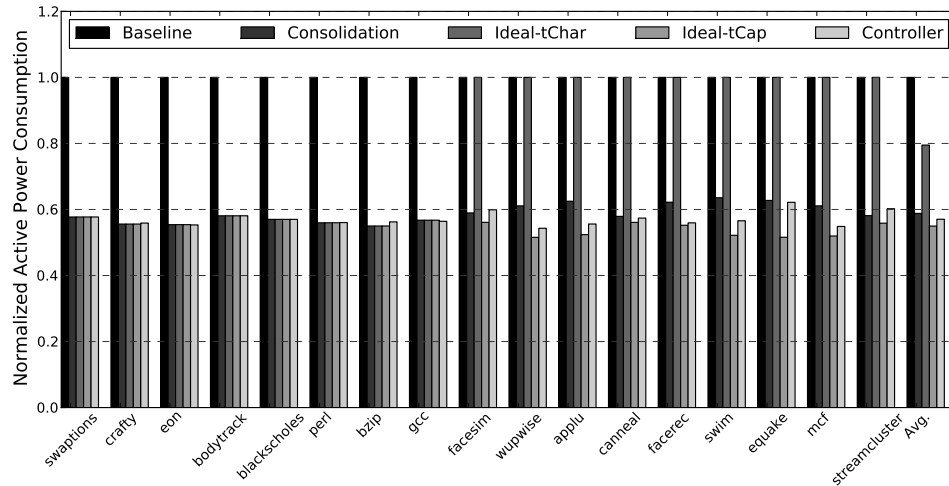
(a) *RUBiS* and Batch VM
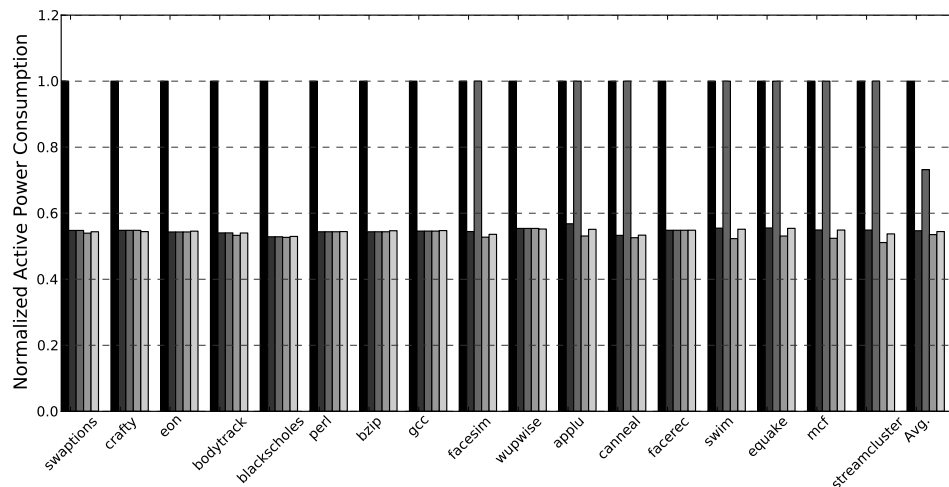
(b) *Olio* and Batch VM

Figure 6.11: Comparison of all the policies for active power consumption. The results are normalized against the Baseline policy.

The Baseline policy gives the best MIPS for the batch jobs but is inherently energy inefficient, since it constantly keeps two machines active, resulting in the highest active power consumption (see Figure 6.11). The Consolidation policy saves active power (∼40%) through VM consolidation. However, it results in poor qMIPS/Watt because consolidation with memory intensive batch VMs results in bottlenecks for the service VMs (as shown in Figure 6.6), and consequently violations of their QoS (failure to meet the QoS requirements corresponds to 0 value

of qMIPS/WATT). So although the policy is reducing the active power consumption across the tNodes though consolidation (see Figure 6.11), it often violates the performance objectives of the system, resulting in average qMIPS/Watt similar to that of the Baseline policy.

The Ideal-tChar policy combines the best of the two previous policies. This policy, based on oracle knowledge, consolidates the VMs when the their QoS is maintained and keeps them separate when not. Ideal-tChar gets a 40% increase in qMIPS/WATT over the baseline for RUBiS and 50% increase for *Olio*. The policy however, misses out on the opportunity to save energy through consolidation for more memory intensive batch VM workloads (like *mcf*, *art* etc.) as evident in Figure 6.11. The Ideal-tCap policy, on the other hand, accomplishes consolidation under all circumstances. To guarantee the QoS of the service VM, it places a limit or cap on the CPU utilization of the batch VM such that the service VM just meets it QoS requirement. A cap on CPU allocation results in lesser time spent by batch VM on the CPU, which reduces the interference effects. However, Ideal-tCap not only fails to improve the useful work done per joule but actually results in its slight reduction. This is explained in Figure 6.10, where we observe that the MIPS of the batch VM because of capping drops considerably. Figure 6.13 shows the details of actual cap values employed by the Ideal-tCap policy on the memory intensive batch VMs, when co-scheduled with *RUBiS* and *Olio*. In some cases like *mcf*, *equake*, which cause a lot of interference effects, the cap applied is ≤50%, which implies more than 50% of the time it is not allowed to run. This implies that the benefit of consolidation in terms of energy savings is more than offset by compromising the MIPS of the batch VM. This results in the Ideal-tCap policy performing even worse than the Baseline policy in terms of qMIPS/Watt for some very memory intensive batch VMs. For less memory intensive batch VMs it performs very well and thus achieves average qMIPS/Watts similar to the Ideal-tChar policy.

Our Controller policy outperforms all the other policies for both the service workloads. For *RUBiS*, the Controller policy does on average 70% better than the Baseline in qMIPS/WATT and ∼35% better than the Ideal policies, while for *Olio* it is 70% better than the baseline and ∼25% better than Ideal policies.

The large gains in qMIPS/Watt of the Controller policy over the Ideal policies is a consequence of the fact that it is able to exploit the heterogeneity in the way they use their CPU resources. It resorts to scaling the number of vCPUs of the service VM as a way to control the interference between the two types of VMs, which exactly exploits the heterogeneity. For example, Figure 6.12 shows how the Controller adapts to converge to an optimal number of vCPUs for *RUBiS* web server, when running with *streamcluster* as the workload inside the batch VM. It plots the vCPU selection and QoS ratio of *RUBiS* over the samples collected by the system. We can see that the system initially starts with 2 vCPUs for web server. The interference effects of *streamcluster* results in a CPU bottleneck – this manifests in the form of a poor QoS ratio, which when crosses the QoS_{th} (0.6), makes the Controller switch to 4 vCPUs. Beyond that it reaches a steady state, where it converges to 3 vCPUs, which as we see in Figure 6.12, is a stable solution for satisfying the QoS of *RUBiS*. It sticks with 3 vCPUs even though the QoS ratio is below QoS_{th} since the performance model predicts the CPU utilization of the service VM would cross the $util_{th}$ if increased. Thus, using a combined QoS ratio and CPU utilization based feedback, the Controller is able to converge to a stable solution.

Figure 6.10 demonstrates that the raw MIPS achieved by the Controller is on an average within ∼7% of the maximum possible, i.e. the Baseline, and in the worst case 17% and 22% below the maximum with *RUBiS* and *Olio* respectively. At the same time it is able to dramatically reduce the active power consumption of the system (∼50%) compared to the Baseline case (see Figure 6.11). Hence, overall the Controller is able to maximize the energy efficiency while limiting the impact on the raw performance of the batch VMs. In contrast, the Ideal-tCap policy is on an average ∼30% below and in the worst case almost 70% below the Baseline. The results of the Controller further highlight the importance of heterogeneous consolidation – with batch-only consolidation the worst case drop in MIPS is 40%, while the average is ∼20% (see Figure 6.3).

A close examination of the qMIPS/Watt results of the Ideal policies for *Olio* and *RUBiS* shows some differences even for the same batch VM workloads in Figure

6.9. There are two primary reasons for that: (1) The way the Ideal-tChar classifies *facerec* and *wupwise* benchmarks. When co-scheduled with *RUBiS*, these two workloads generate enough interference effects to violate the QoS requirement of *RUBiS*. As a consequence, they are not consolidated by Oracle-tChar with *RUBiS*, as evident from Figure 6.9. However, with *Olio*, the interference effects are limited, which allows successful consolidation. This results in higher qMIPS/Watt for the Ideal-tChar policy with *Olio* compared to with *RUBiS*. (2) Difference in average cap allocation by Ideal-tCap for batch VMs. This observation is highlighted by Figure 6.13, where we clearly see that the same workload needs different amount of capping across *RUBiS* and *Olio*. As mentioned above, since there are limited interference effects between *Olio*, and *wupwise* and *facerec*, they do not need any cap to be applied. For the other memory intensive workloads as well, the required cap is higher than with *RUBiS*, which allows the Ideal-tCap policy to get a higher qMIPS/Watt when running with *Olio*.

This behavior also demonstrates the difficulty in predicting interference effects between applications when sharing elements of the memory hierarchy. It can be deduced that CPU intensive batch VMs consolidate well with service VMs as long as there is sufficient headroom for both, while very memory intensive batch VMs degrade the QoS of co-scheduled service VMs. However, in between these two extremes, it becomes more difficult to predict what the consolidation outcome would be. This observation validates the use of dynamic QoS feedback for effective resource management, which the Controller policy employs.

6.6.1   Controller Adaptability

In order to study how the proposed Controller policy adapts to changing workload and QoS requirements, we did the following two experiments: (1) Co-schedule service VM with batch VM of alternating characteristics and (2) Co-schedule service VM with batch VM when the load of the former dynamically changes.

Figure 6.14 illustrates the adaptability of the Controller when co-scheduling *RUBiS* web server VM with a batch VM that is memory intensive at the beginning
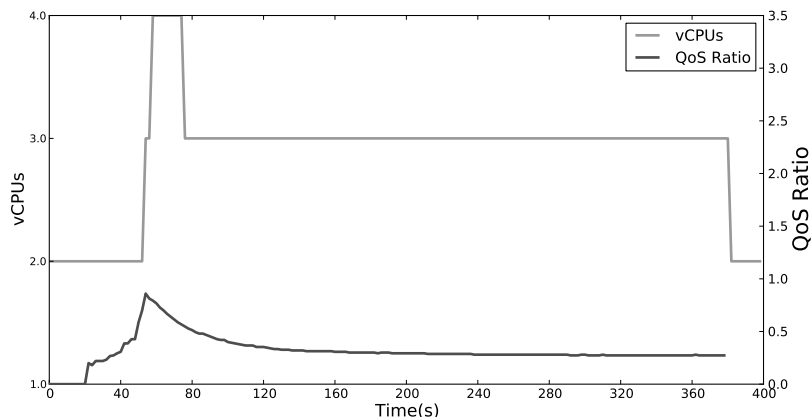
Figure 6.12: vCPU selection timeline of Controller for the *RUBiS* web server with *streamcluster* as the consolidated batch VM.
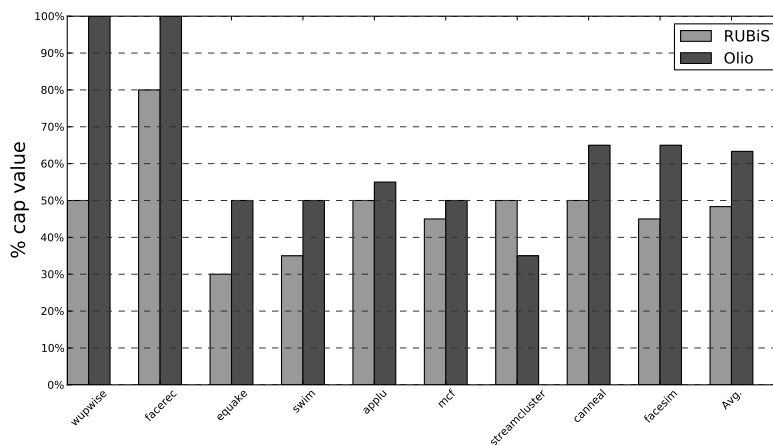
Figure 6.13: % cap applied by the Capping policy on the different batch VMs with both *RUBiS* and *Olio*.

of the experiment (*swim*), CPU intensive in the middle (*perl*) and then memory intensive again at the end (*equake*). The plot shows the vCPU selection for the *RUBiS* VM on the left y-axis and the QoS ratio achieved on the right y-axis. We can see that the Controller manages to adapt with the changing levels of interference between the different workloads and finds the best allocation of vCPUs in order to satisfy the QoS requirement. When QoS ratio increases because 2 vCPUs is not sufficient for service VM the controller reacts and increases the number of vCPUs to 3 (just like it did in Figure 6.12 when running with *streamcluster*). However, when the interference effects drop, the Controller is able to infer that through the

Figure 6.14: Adaptability with changing batch VM workloads (*RUBiS* web server in the service VM).

drop in the CPU utilization of the web server VM, and decreases the number of vCPUs to increase the MIPS the batch VM can get. Similarly, when the batch VM again becomes memory intensive, the Controller again assigns another vCPU to resolve the bottleneck.

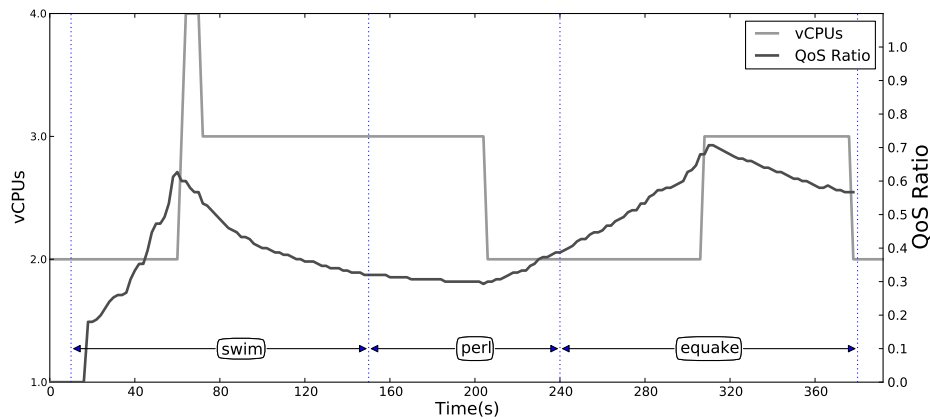In the next experiment (see Figure 6.15), we dynamically change the load on the *Olio* service VM by changing the number of user sessions emulated by the sClient and co-schedule it with *perl* running as the batch VM. For the first 200 seconds of the experiment *Olio* services 550 users, just like the experiments presented earlier in Figure 6.9; for the next 200 seconds the workload increases to 650 users, and for the last 200 secs returns to 550. Initially, the controller converges to 2 vCPUs since *perl* and *Olio* co-exist without interfering and the QoS is comfortably met. As soon as the number of users increases to 650, 2 vCPUs for serviceVM do not suffice and the QoS ratio increases. In response, the Controller increases the number of vCPUs and finally converges to 3 vCPUs to ensure that the QoS ratio is satisfied. In the last part of the experiment the CPU bottleneck is alleviated through reduction in load on the service VM and the controller again downsizes the vCPU allocation to 2 in response.

These experiments highlight the robustness of the Controller and show that its decision-making and stability is independent of workloads or their configuration.
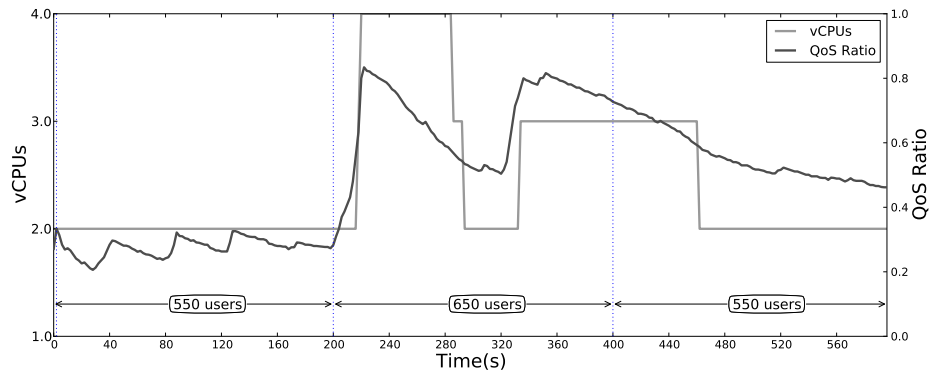
Figure 6.15: Adaptability with changing number of service VM users (*Olio* is the service VM and *perl* is the batch VM).

6.7 Conclusions

The last two chapters evaluated mechanisms and policies to achieve energy efficiency in server class systems using workload consolidation. In the last chapter we showed how workload characterization is extremely important for achieving both performance and energy efficiency in consolidated environments, and discussed the design, implementation and evaluation of vGreen system to accomplish that. However, the system considered only throughput-oriented intensive batch jobs as the workloads managed by it.

This chapter explored the challenges in managing both latency sensitive service and batch jobs within a consolidated environment in data centers. We identified qMIPS/WATT as the proper metric to capture the amount of work done per joule while maintaining a pre-specified level of QoS, and showed that consolidating batch and services workloads provides a compelling opportunity for maximizing it. The evaluation demonstrated that existing state-of-the-art resource management techniques that employ CPU capping or selective VM consolidation (like vGreen) fail to maximize qMIPS/WATT. In contrast the Themis controller, which leverages the heterogeneous characteristics of the workloads in terms of their CPU requirements outperforms an ideal implementation of these policies by up to 35% on average.

Chapter 6, in part, is a reprint of the material under submission at International Conference for High Performance Computing, Networking, Storage and

Analysis, 2011. Dhiman, G.; Kontorinis, V.; Ayoub, R.; Sadler, C.; Tullsen, D. and Rosing, T.S. The dissertation author is the primary investigator and author of this paper.

Chapter 7

Conclusion and Future Work

Energy management is a critical issue in the design of computing systems today in both mobile as well as enterprise space. For the mobile systems it is extremely important from the perspective of battery life, while for the large scale systems it directly impacts the cost of operation.

Towards this end, this thesis identifies and explores three mechanisms to achieve system level energy efficiency: (1) *Active power management (DPM and DVFS):* An online learning based policy is proposed that can manage devices with different power management capabilities across varying workload profiles. (2) *Energy Proportional Design:* A hardware design and runtime software solution to dynamically manage a novel memory hierarchy comprising of PRAM and DRAM is proposed. The new hierarchy converts an inherently non energy proportional conventional DRAM based memory into a more energy proportional component. (3) *VM based Workload Consolidation:* Efficient VM management policies are presented to achieve high energy savings through intelligent workload consolidation and resource management.

The key idea that all the proposed policies in this thesis exploit is 'workload characterization', i.e. an understanding of how the workloads use the system resources. Active power management policies use this knowledge to determine the best possible power state for the workload. The PDRAM system exploits this knowledge to intelligently allocate the pages of a workload across PRAM and DRAM to leverage benefits of both, while the vGreen and Themis systems develop

VM scheduling and resource management algorithms based on the resource requirements of the workloads. Consequently, it allows them to out perform policies that do not take this knowledge into account. The following sections summarize the contributions of the thesis and discuss some ideas on future research directions.

7.1 Thesis Summary

7.1.1 Active Power Management

DPM and DVFS policies achieve energy savings by reducing the power consumption during the idle and busy periods of operation respectively. While a number of heuristic and stochastic DPM policies have been proposed in the past, we found that no single policy solution could adapt well under varying workload conditions. Additionally, for devices like CPU that support both DPM and DVFS, existing work failed to take into account the interplay between the two. This, as we show in this thesis, is extremely important since the energy savings based on DVFS come at the cost of increased execution time, which implies greater leakage energy consumption and shortened idle periods for using DPM.

This thesis proposes a novel setup of performing active power management where online learning [34] is applied to select among a set of DPM policies and v-f settings. The online learning algorithm or the *controller* has a set of *experts* (DPM policies/v-f settings) to choose from and selects an expert that has the best chance to perform well based on the controller's characterization of the current workload. The selection takes into account energy savings, performance delay as well as the user specified energy-performance tradeoff (referred to as e/p tradeoff). The algorithm is guaranteed to converge to best performing expert in the set, thus delivering performance atleast as good as the best expert in the set, across different workloads.

As we show in this thesis, active power management is well suited for energy savings on mobile systems primarily due to their usage pattern that allows aggressive power management [42]. However, due to the energy proportionality problem, it is not that effective for server class systems. We identify two means of

achieving energy efficiency for such systems: (1) Energy proportional design and (2) Workload consolidation using virtualization.

7.1.2 Energy Proportional Design

Lack of energy proportional components in the server systems reduces the effectiveness of active power management techniques for system level energy savings. Although the CPU is highly energy proportional, its contribution to the total power consumption of the system is just around 30%. Consequently, the server systems consume as much 50% of their peak power even when their utilization is negligible due to non energy proportional components like fans, power supply, memory etc. In modern data center deployments, due to increasing data intensiveness of the applications, memory energy consumption has become a big problem, as it contributes 30-40% to the total power consumption [42].

To this end, this thesis proposes a novel memory architecture, which can significantly reduce the memory energy consumption, hence making the memory subsystem more energy proportional than the conventional DRAM based systems. The new architecture leverages the low read and static power of PRAM, and the high write endurance of DRAM to build a hybrid memory hierarchy comprising of both PRAM and DRAM. By exploiting the characteristics of the workloads in terms of how they use the memory (read and write intensity), the proposed system is able to take advantages of both the technologies to achieves much higher energy savings compared to DRAM based memory systems for both memory as well as non memory intensive workloads.

7.1.3 Workload Consolidation

For large scale deployments like data centers and cloud computing infrastructures, another way to achieve energy proportionality is by consolidating workloads into fewer machines. This pushes the active machines into an energy efficient zone of operation (higher utilization; see Figure 1.1), while the freed up machine could be switched off to achieve an energy proportional state of operation. Virtu-

alization has gained a lot of traction is recent years, since it facilitates workload consolidation through ease of dynamic workload and resource management. Policies for performing dynamic virtual machine consolidation and management have been proposed in previous research [86, 74, 103]. However, most of them rely on overall CPU utilization of the physical machines and its VMs as an indicator of their respective power consumption and resource utilization, and use it for guiding the VM management policy decisions.

This thesis shows that based on the characteristics of these different co-located VMs, the overall power consumption and performance of the VMs can vary a lot even at similar CPU utilization levels, which can mislead the VM management policies into making decisions that can create hotspots of activity, violate QoS requirements and degrade overall performance and energy efficiency. Based on this observation, VM scheduling and resource management policies are proposed that can maximize energy efficiency while ensuring that the performance requirements of the workloads within the VM are satisfied as well.

7.2   Future Research Directions

7.2.1   I/O Resource Management in Virtualized Environments

This thesis highlights the importance of CPU resource management in virtualized environments in Chapter 6 for avoiding CPU bottlenecks and facilitating VM consolidation for energy efficiency. However, aggressive VM consolidation can also result in bottlenecks of I/O resources (network, disk etc.). The Themis system in Chapter 6 prevents this by consolidating batch and services workloads, as they complement each other in I/O resource usage.

However, there could be batch jobs like MapReduce [25], which can have phases that are more intensive on I/O than the SPEC and PARSEC workloads assumed by the Themis system. In the presence of such batch jobs, it is important to also take into account I/O resource management to ensure that the QoS

requirements of services is not violated in case of any bottlenecks. The batch and services consolidation would still be beneficial, since if the I/O bandwidth required by the services to just satisfy its QoS ratio is guaranteed, the excess could be safely granted to the batch jobs.

7.2.2   Energy Proportionality for Storage

With virtualization becoming ubiquitous in data centers and cloud computing, energy consumption of storage, that typically host the disk images of the different VMs running across the data centers is becoming very important. The disk images could be residing either on enterprise storage servers or even on cluster of commodity servers, where computation and storage are co-located on the same nodes. Recent work has looked into how to achieve energy proportionality for storage under both the cases. The research in [98] proposes SRCMap, a storage virtualization layer, that dynamically consolidates cumulative workload of VMs on a subset of physical volumes of a storage server. This allows the system to spin down the remaining volumes to achieve energy proportionality. On the other hand, the research in [95] proposes Sierra, a system to achieve energy proportionality in distributed storage subsystems based on commodity servers. They propose a *power-aware layout* to allow a significant fraction of servers to be powered down without losing availability or fault tolerance.

A complementary approach towards solving the energy proportionality problem in storage could be to re-design the storage hierarchy by adding non volatile technologies like flash and PRAM. These memory technologies cannot match disk in terms of GB/$, but can provide much faster lower power consumption and faster access times. This motivates a new hierarchy where flash or PRAM could act as a cache of hot pages, which allows longer idle periods for the disk to be spun down. Besides, due to redundancy present across VM disk images (OS and application code for instance), techniques to de-duplicate them can free up physical volumes that could be either spun down or used for additional work to increase the overall energy efficiency.

Bibliography

[1] Mobile AMD Athlon 4 processor model 6 cpga data sheet. http://www.amd.com, 2001.

[2] H. S. Abdelsalam, K. Maly, R. Mukkamala, M. Zubair, and D. Kaminsky. Analysis of energy efficiency in clouds. *Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, Computation World*, pages 416–421, 2009.

[3] Amazon. *Amazon Elastic Compute Cloud (Amazon EC2)*. Amazon Inc., http://aws.amazon.com/ec2/, 2008.

[4] C. Amza, A. Chanda, A. L. Cox, S. Elnikety, R. Gil, K. Rajamani, and W. Zwaenepoel. Specification and implementation of dynamic web site benchmarks. In *In 5th IEEE Workshop on Workload Characterization*, pages 3–13, 2002.

[5] Apache. *http://incubator.apache.org/olio/*.

[6] R. Ayoub, S. Sherifi, and T. Rosing. Gentlecool: Cooling aware proactive workload scheduling in multi-machine systems. In *IEEE Design, Automation Test in Europe, 2010. DATE '10.*, March 2010.

[7] A. Azevedo, I. Issenin, R. Cornea, R. Gupta, N. Dutt, A. Veidenbaum, and A. Nicolau. Profile-based dynamic voltage scheduling using program checkpoints. In *DATE '02*, page 168, 2002.

[8] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[9] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *Computer*, 40:33–37, December 2007.

[10] F. Bedeschi et al. An 8Mb demonstrator for high-density 1.8V phase-change memories. *In Symposium on* VLSI *Circuits*, pages 442–445, 2004.

[11] F. Bedeschi et al. A multi-level-cell bipolar-selected phase-change memory. *Proc ISSCC'08*, pages 427–429, 2008.

[12] L. Benini, A. Bogliolo, S. Cavallucci, and B. Ricco;. Monitoring system activity for os-directed dynamic power management. In *ISLPED '98*, pages 185–190, 1998.

[13] R. Bianchini and R. Rajamony. Power and energy management for server systems. *IEEE Computer*, 37:2004, 2004.

[14] N. L. Binkert et al. The m5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006.

[15] S. Bird, A. Phansalkar, L. K. John, A. Mericas, and R. Indukuru. Characterization of performance of spec cpu benchmarks on intel's core microarchitecture based processor. *SPEC Benchmark Workshop*, 2007.

[16] N. Bobroff, A. Kochut, and K. Beaty. Dynamic placement of virtual machines for managing sla violations. In *Integrated Network Management*, pages 119–128. IEEE, 2007.

[17] A. P. Chandrakasan and R. Brodersen. Minimizing power consumption in digital CMOS circuits. *Proceedings of the IEEE*, 83:498–523, 1995.

[18] J. S. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle. Managing energy and server resources in hosting centers. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 103–116, New York, NY, USA, 2001. ACM.

[19] K. Choi, R. Soma, and M. Pedram. Fine-grained dynamic voltage and frequency scaling for precise energy and performance tradeoff based on the ratio of off-chip access to on-chip computation times. *IEEE Trans. on CAD of Integrated Circuits and Systems*, (1):18–28, 2005.

[20] E.-Y. Chung, L. Benini, A. Bogliolo, Y.-H. Lu, and G. D. Micheli. Dynamic power management for nonstationary service requests. *IEEE Trans. Computers*, 51(11):1345–1361, 2002.

[21] E.-Y. Chung, L. Benini, and G. D. Micheli. Dynamic power management using adaptive learning tree. In *ICCAD'99*, pages 274–279, 1999.

[22] E.-Y. Chung, G. D. Micheli, and L. Benini. Contents provider-assisted dynamic voltage scaling for low energy multimedia applications. In *ISLPED '02*, pages 42–47, 2002.

[23] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.

[24] P. J. de Langen and B. H. H. Juurlink. Leakage-aware multiprocessor scheduling for low power. In *IPDPS*. IEEE, 2006.

[25] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[26] V. Delaluz et al. Scheduler-based dram energy management. In *Proc DAC'02*, pages 697–702, 2002.

[27] G. Dhiman, V. Kontorinis, D. Tullsen, T. Rosing, E. Saxe, and J. 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 '10, pages 437–442, New York, NY, USA, 2010. ACM.

[28] G. Dhiman, G. Marchetti, and T. Rosing. vGreen: A system for energy efficient computing in virtualized environments. In *ISLPED'09*, New York, NY, USA, 2009. ACM.

[29] G. Dhiman and T. S. Rosing. Dynamic voltage frequency scaling for multitasking systems using online learning. In *ISLPED '07: Proceedings of the 2007 international symposium on Low power electronics and design*, pages 207–212, New York, NY, USA, 2007. ACM.

[30] F. Douglis, P. Krishnan, and B. N. Bershad. Adaptive disk spin-down policies for mobile computers. In *MLICS '95: Proceedings of the 2nd Symposium on Mobile and Location-Independent Computing*, pages 121–137, Berkeley, CA, USA, 1995. USENIX Association.

[31] M. Elnozahy, M. Kistler, and R. Rajamony. Energy conservation policies for web servers. In *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, USITS'03, pages 8–8, Berkeley, CA, USA, 2003. USENIX Association.

[32] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A performance counter architecture for computing accurate cpi components. *SIGOPS Oper. Syst. Rev.*, 40(5):175–184, 2006.

[33] X. Fan, W.-D. Weber, and L. A. Barroso. Power provisioning for a warehouse-sized computer. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 13–23, New York, NY, USA, 2007. ACM.

[34] Y. Freund and R. E. Schapire:. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal Computer and System Sciences*, 55(1):119–139, 1997. Special issue for EuroCOLT '95.

[35] R. Ge, X. Feng, W.-c. Feng, and K. W. Cameron. Cpu miser: A performance-directed, run-time system for power-aware clusters. In *ICPP '07: Proceedings of the 2007 International Conference on Parallel Processing*, page 18, Washington, DC, USA, 2007. IEEE Computer Society.

[36] M. Gillespie. Power Management in the Intel PXA27x series application processors. http://www.intel.com/cd/ids/developer/asmo-na/eng/dc/pca/power/176608.htm.

[37] R. Golding, P. Bosch, and J. Wilkes. Idleness is not sloth. Technical Report HPL-96-140, Hewlett Packard Laboratories, Oct. 4 1996.

[38] K. Govil, E. Chan, and H. Wasserman. Comparing algorithm for dynamic speed-setting of a low-power cpu. In *MobiCom '95: Proceedings of the 1st annual international conference on Mobile computing and networking*, pages 13–25, New York, NY, USA, 1995. ACM Press.

[39] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embed-ded benchmark suite. In *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization*, Dec. 2001.

[40] E. L. Haletky. *VMware ESX Server in the Enterprise: Planning and Securing Virtualization Servers.* 2008.

[41] F. Hermenier, X. Lorca, J.-M. Menaud, G. Muller, and J. Lawall. Entropy: a consolidation manager for clusters. In *VEE '09: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution en-vironments*, pages 41–50, New York, NY, USA, 2009. ACM.

[42] U. Hoelzle and L. A. Barroso. *The Datacenter as a Computer: An Intro-duction to the Design of Warehouse-Scale Machines.* Morgan and Claypool Publishers, 1st edition, 2009.

[43] http://lwn.net/Articles/223185/.

[44] http://pxa linux.sourceforge.net/.

[45] http://www.bitmover.com/lmbench/.

[46] http://www.hpl.hp.com/research/cacti/.

[47] http://www.intel.com/technology/architecture-silicon/intel64/45nm core2_whitepaper.pdf.

[48] http://www.intel.com/technology/architecture-silicon/next
     gen/whitepaper.pdf.

[49] http://www.itrs.net/Links/2005ITRS/ExecSum2005.pdf.

[50] http://www.micron.com/products/dram/ddr3/.

[51] http://www.numonyx.com/Documents/WhitePapers.

[52] H. Huang et al. Design and implementation of power-aware virtual memory.
     In *Proc ATEC'03*, 2003.

[53] C.-H. Hwang and A. C.-H. Wu. A predictive system shutdown method for
     energy saving of event-driven computation. In *ICCAD '97*, pages 28–32,
     1997.

[54] Intel.   Intel   XScale   Core   Developer's   Manual.
     http://download.intel.com/design/intelxscale/27347302.pdf.

[55] IPMI. Intelligent platform management interface v2.0 specification. 2004.

[56] C. Isci, G. Contreras, and M. Martonosi. Live, runtime phase monitoring and
     prediction on real systems with application to dynamic power management.
     In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International
     Symposium on Microarchitecture*, pages 359–370, Washington, DC, USA,
     2006. IEEE Computer Society.

[57] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically
     variable voltage processors. In *ISLPED-98*, pages 197–202, Aug. 10–12 1998.

[58] R. Jejurikar, C. Pereira, and R. Gupta. Leakage aware dynamic voltage
     scaling for real-time embedded systems. In *DAC '04: Proceedings of the 41st
     annual conference on Design automation*, pages 275–280, New York, NY,
     USA, 2004. ACM Press.

[59] A. R. Karlin, M. S. Manasse, L. A. McGeoch, and S. Owicki. Competitive
     randomized algorithms for non-uniform problems. In *SODA '90: Proceedings
     of the first annual ACM-SIAM symposium on Discrete algorithms*, pages
     301–309, Philadelphia, PA, USA, 1990. Society for Industrial and Applied
     Mathematics.

[60] T. Kgil et al. Improving nand flash based disk caches. In *Proc ISCA '08*,
     pages 327–338, 2008.

[61] A. Lacaita et al. Status and challenges of pcm modeling. *Solid State Device
     Research Conference, 2007. ESSDERC 2007. 37th European*, pages 214–221,
     Sept. 2007.

[62] A. R. Lebeck et al. Power aware page allocation. *SIGOPS Oper. Syst. Rev.*, 34(5):105–116, 2000.

[63] B. Lee et al. Architecting phase change memory as a scalable dram alternative. *Proc ISCA'09*, 2009.

[64] M. Lee, A. S. Krishnakumar, P. Krishnan, N. Singh, and S. Yajnik. Supporting soft real-time tasks in the xen hypervisor. In *VEE '10: Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 97–108, New York, NY, USA, 2010. ACM.

[65] H.-L. Li et al. Energy-aware flash memory management in virtual memory system. *IEEE Trans. Very Large Scale Integr. Syst.*, 16(8):952–964, 2008.

[66] L. Liu, H. Wang, X. Liu, X. Jin, W. B. He, Q. B. Wang, and Y. Chen. Greencloud: a new architecture for green data center. In *ICAC-INDST '09: Proceedings of the 6th international conference industry session on Autonomic computing and communications industry session*, pages 29–38, New York, NY, USA, 2009. ACM.

[67] P. Mangalagiri et al. A low-power phase change memory based hybrid cache architecture. In *Proc. GLSVLSI '08*, pages 395–398, 2008.

[68] M. McNett, D. Gupta, A. Vahdat, and G. M. Voelker. Usher: an extensible framework for managing custers of virtual machines. In *LISA'07: Proceedings of the 21st conference on Large Installation System Administration Conference*, pages 1–15, Berkeley, CA, USA, 2007. USENIX Association.

[69] D. A. Menasce. Composing web services: A qos view. *IEEE Internet Computing*, 8:88–90, November 2004.

[70] A. Merkel, J. Stoess, and F. Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *EuroSys '10: Proceedings of the 5th European conference on Computer systems*, pages 153–166, New York, NY, USA, 2010. ACM.

[71] S. Microsystems. *http://faban.sunsource.net*.

[72] J. Moore, J. Chase, P. Ranganathan, and R. Sharma. Making scheduling "cool": temperature-aware workload placement in data centers. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 5–5, Berkeley, CA, USA, 2005. USENIX Association.

[73] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-clouds: managing performance interference effects for qos-aware clouds. In *EuroSys '10: Proceedings of the 5th European conference on Computer systems*, pages 237–250, New York, NY, USA, 2010. ACM.

[74] R. Nathuji and K. Schwan. Virtualpower: coordinated power management in virtualized enterprise systems. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 265–278, New York, NY, USA, 2007. ACM.

[75] J. Nieh and M. S. Lam. A smart scheduler for multimedia applications. *ACM Trans. Comput. Syst.*, 21:117–163, May 2003.

[76] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The eucalyptus open-source cloud-computing system. In *Proceedings of Cloud Computing and Its Applications*, 2008.

[77] D. Ongaro, A. L. Cox, and S. Rixner. Scheduling i/o in virtual machine monitors. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 1–10, New York, NY, USA, 2008. ACM.

[78] OpenNebula. Opennebula homepage.

[79] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant. Automated control of multiple virtualized resources. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*, pages 13–26, New York, NY, USA, 2009. ACM.

[80] G. A. Paleologo, L. Benini, A. Bogliolo, and G. D. Micheli. Policy optimization for dynamic power management. In *DAC '98*, pages 182–187, 1998.

[81] A. Pirovano et al. Scaling analysis of phase-change memory technology. *Proc IEDM'08*, pages 29.6.1– 29.6.4, 2003.

[82] A. Pirovano et al. Phase-change memory technology with self-aligned μtrench cell architecture for 90nm node and beyond. *Solid-State Electronics*, 52(9):1467 – 1472, 2008.

[83] V. Prasad, W. Cohen, F. C. Eigler, M. Hunt, J. Keniston, and B. Chen. Abstract locating system problems using dynamic instrumentation. In *Linux Symposium*, 2005.

[84] Q. Qiu and M. Pedram. Dynamic power management based on continuous-time markov decision processes. In *DAC '99*, pages 555–561, 1999.

[85] G. Quan and X. Hu. Minimum energy fixed-priority scheduling for variable voltage processor. In *DATE '02*, page 782, 2002.

[86] R. Raghavendra, P. Ranganathan, V. Talwar, Z. Wang, and X. Zhu. No "power" struggles: coordinated multi-level power management for the data center. In *ASPLOS XIII: Proceedings of the 13th international conference*

on Architectural support for programming languages and operating systems, pages 48–59, New York, NY, USA, 2008. ACM.

[87] P. Ranganathan, P. Leech, D. Irwin, and J. Chase. Ensemble-level power management for dense blade servers. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 66–77, Washington, DC, USA, 2006. IEEE Computer Society.

[88] Z. Ren, B. H. Krogh, and R. Marculescu. Hierarchical adaptive dynamic power management. *IEEE Trans. Computers*, 54(4):409–420, 2005.

[89] C. Ruemmler and J. Wilkes. UNIX disk access patterns. In *USENIX Winter*, pages 405–420, 1993.

[90] T. Simunic, L. Benini, P. W. Glynn, and G. D. Micheli. Event-driven power management. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 20(7):840–857, 2001.

[91] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous mutlithreading processor. *SIGPLAN Not.*, 35(11):234–244, 2000.

[92] M. B. Srivastava, A. P. Chandrakasan, and R. W. Brodersen. Predictive system shutdown and other architectural techniques for energy efficient programmable computation. *IEEE Trans. Very Large Scale Integr. Syst.*, 4(1):42–55, 1996.

[93] V. Sundaram, A. Chandra, P. Goyal, P. Shenoy, J. Sahni, and H. Vin. Application performance in the qlinux multimedia operating system. In *Proceedings of the eighth ACM international conference on Multimedia*, MULTIMEDIA '00, pages 127–136, New York, NY, USA, 2000. ACM.

[94] N. Takaura et al. A GeSbTe phase-change memory cell featuring a tungsten heater electrode for low-power, highly stable, and short-read-cycle operations. *Electron Devices Meeting, 2003. IEDM '03 Technical Digest. IEEE International*, pages 37.2.1–37.2.4, Dec. 2003.

[95] E. Thereska, A. Donnelly, and D. Narayanan. Sierra: practical power-proportionality for data center storage. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 169–182, New York, NY, USA, 2011. ACM.

[96] A. Varma, B. Ganesh, M. Sen, S. R. Choudhury, L. Srinivasan, and J. Bruce. A control-theoretic approach to dynamic voltage scheduling. In *CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 255–266, New York, NY, USA, 2003. ACM Press.

[97] A. Verma, P. Ahuja, and A. Neogi. Power-aware dynamic placement of hpc applications. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 175–184, New York, NY, USA, 2008. ACM.

[98] A. Verma, R. Koller, L. Useche, and R. Rangaswami. Srcmap: energy proportional storage using dynamic consolidation. In *Proceedings of the 8th USENIX conference on File and storage technologies*, FAST'10, pages 20–20, Berkeley, CA, USA, 2010. USENIX Association.

[99] VMware. Vmware distributed resource scheduler. 2009.

[100] L. Wang, G. von Laszewski, J. Tao, and M. Kunze. Grid virtualization engine: design, implementation and evaluation. *IEEE Systems Journal*, 3(4):477–488, 12/2009 2009.

[101] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced cpu energy. In *OSDI '94: Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, page 2, Berkeley, CA, USA, 1994. USENIX Association.

[102] A. Weissel and F. Bellosa. Process cruise control: event-driven clock scaling for dynamic power management. In *CASES '02: Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 238–246, New York, NY, USA, 2002. ACM Press.

[103] T. Wood, P. Shenoy, and Arun. Black-box and gray-box strategies for virtual machine migration. In *NSDI'07*, pages 229–242, 2007.

[104] P. Yang, C. Wong, P. Marchal, F. Catthoor, D. Desmet, D. Verkest, and R. Lauwereins. Energy-aware runtime scheduling for embedded-multiprocessor SOCs. *IEEE Design & Test of Computers*, 18(5):46–58, 2001.

[105] B. Yu et al. Chalcogenide-nanowire-based phase chnage memory. *IEEE Transactions on Nanotechnology*, 7(4), 2008.

[106] Y. Zhu and F. Mueller. Feedback edf scheduling of real-time tasks exploiting dynamic voltage scaling. *Real-Time Syst.*, 31(1-3):33–63, 2005.