ADAPTIVE POWER AND PERFORMANCE MANAGEMENT OF COMPUTING SYSTEMS

by

Bithika Khargharia

A Dissertation Submitted to the Faculty of the

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

In Partial Fulfillment of the Requirements

For the Degree of

DOCTOR OF PHILOSOPHY

In the Graduate College

THE UNIVERSITY OF ARIZONA

2008

THE UNIVERSITY OF ARIZONA

GRADUATE COLLEGE

As members of the Dissertation Committee, we certify that we have read the dissertation prepared by Bithika Khargharia entitled ADAPTIVE POWER AND PERFORMANCE MANAGEMENT OF COMPUTING SYSTEMS and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy

_____Date: 04/21/08

Salim Hariri

_____Date: 04/21/08

Bernard Zeigler

_____Date: 04/21/08

Jerzy Rozenblit

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copies of the dissertation to the Graduate College.

I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

_____Date: 04/21/08

Dissertation Director: Salim Hariri

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at the University of Arizona and is deposited in the University Library to be made available to borrowers under the rules of the library.

Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgement of the source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the author.

SIGNED: Bithika Khargharia

To my parents – Dharitri and Nila K. Khargharia

ACKNOWLEDGEMENTS

I would like to thank my Ph.D. advisors, Dr Salim Hariri and Dr Mazin S. Yousif, for their continued support and guidance throughout my Ph.D. studies. Because of their varied backgrounds and affiliations, my PhD work has been bolstered both by the academic vision leading towards significant theoretical contribution as also by the technology vision leading towards practical solutions and productizable prototypes. They have allowed me the freedom to pursue my research interests, and provided me with encouragement and patience. By working with them, I have gained not only tremendous research experience, but also strong research ethics that I treasure the most.

In addition, I would like to thank Dr Bernard Zeigler, Dr Jerzy Rozenblit and Dr Neelam Gupta for serving on my oral committee and offering valuable suggestions to improve this dissertation. Thanks particularly to Dr. Jerzy Rozenblit, Department Head, Electrical and Computer Engineering. He has never failed to support and motivated me whenever I needed it, during the course of my PhD. In addition I would like to thank my colleagues at Intel Corporation, who I had the pleasure of working with during my summer internships, who helped provide me initial directions in my PhD work. I would also like to thank Intel Corporation for providing me the infrastructure (equipments, benchmarks, documents) for my validation work. I would like to thank the National Science Foundation for their financial support of this research work.

Last but not least, I am indebted to my friend, critique, mentor and fiancé Amar Singh Chawla for his continued support and motivation all these years. Finally I would like to thank my parents without whom this achievement would not have been possible.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

With the rapid growth of servers and applications spurred by the Internet economy, power consumption in today's data centers is reaching unsustainable limits. This has led to an imminent financial, technical and environmental crisis that is impacting the society at large. Hence, it has become critically important that power consumption be efficiently managed in these computing power-houses of today. In this work, we revisit the issue of adaptive power and performance management of data center server platforms.

Traditional data center servers are statically configured and always over-provisioned to be able to handle peak load. We transform these statically configured data center servers to clairvoyant entities that can sense changes in the workload and dynamically scale in capacity to adapt to the requirements of the workload. The over-provisioned server capacity is transitioned to low-power *state*s and they remain in those *state*s for as long as the performance remains within given acceptable thresholds. The platform power expenditure is minimized subject to performance constraints. This is formulated as a *performance-per-watt* optimization problem and solved using analytical power and performance models. Coarse-grained optimizations at the platform-level are refined by local optimizations at the devices-level namely – the processor & memory subsystems.

Our adaptive interleaving technique for memory power management yielded about 48.8% (26.7 kJ) energy savings compared to traditional techniques measured at 4.5%. Our adaptive platform power and performance management technique demonstrated 56.25% energy savings for memory-intensive workload, 63.75% savings for processor-intensive workload and 47.5% savings for a mixed workload while maintaining platform performance within given acceptable thresholds.

1 INTRODUCTION

1.1 Problem Statement

In recent years, a growing demand for sophisticated applications and services, has led to unprecedented advances in the computing technology – be it portable and hand-held devices or high-performance data centers fueling the Internet economy. However, advanced features and improved performance come at the cost of increased power consumption. While this increases the battery size thus increasing the bulk of portable and hand held devices, it leads to staggering costs of electricity consumption and technically challenging cooling infrastructure in the domain of data center servers. Hence, one of the biggest impediments to advances in the computing technology today is developing energy-efficient computing systems without compromising the quality of service of the target applications and services. To that effect, this thesis addresses the issue of development of energy and performance efficient computing systems.

In this work we focus on the issue of power and performance management for high-performance data center server platforms. A data center is a building that houses thousands of dense servers within relatively small real-estate. It hosts different types of applications/services for different customers setting-up service level agreement (SLA) contracts with them. These SLA contracts guarantee a certain quality of service and the corresponding fee for service. In recent years these data centers, where technology defines the competitive advantage, have voraciously adopted rapid technological advances in computing and network infrastructure, programming paradigms and software services to become and to remain competitive. This has led to an unprecedented data

center expansion. While the sheer size of data centers has increased steadily, the number of servers has grown exponentially. This has put tremendous pressure both on enterprises and service providers, to manage a host of problems associated with deploying and managing the servers needed to support wide range of data center applications [Egenera2003]. Power management is one of the most critical of datacenter management problems of today.

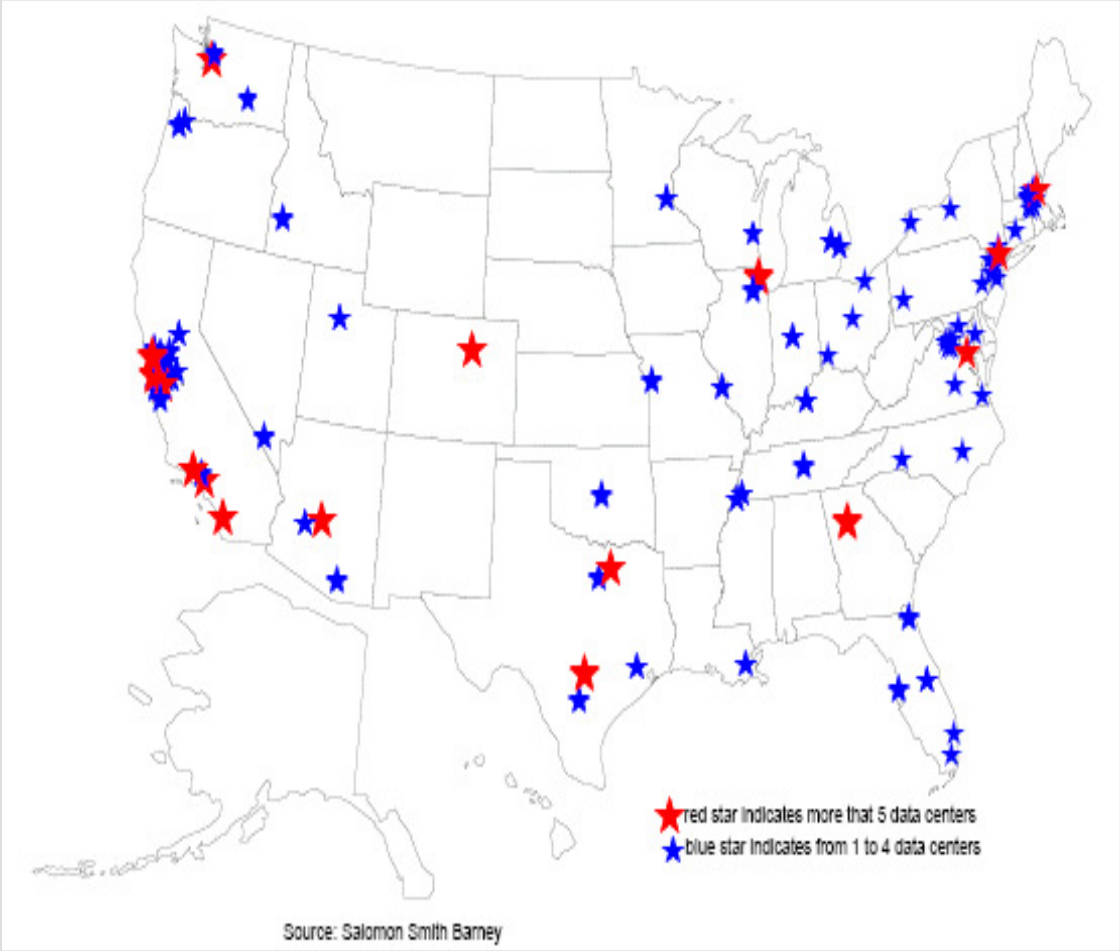

Figure 1-1: U.S. Data Center Hosting Facility Locations [Mitchell-Jackson2001]

[Mitchell-Jackson2001] has compared and contrasted the decentralized nature of the Internet economy supported by the nation's fiber optic backbone and the concentration of electricity demands at specific locations along the same fiber optic backbone. Figure 1-1

plots these data center hosting facilities on the map of United States. They present a marked demand on electricity in these locations. In March 2007, EPA reported that data centers consumed about 61 billion kilowatt-hours (kWh) in 2006 (1.5 percent of total U.S. electricity consumption) for a total electricity cost of about $4.5 billion. This estimated level of electricity consumption is more than the electricity consumed by the nation's color televisions and similar to the amount of electricity consumed by approximately 5.8 million average U.S. households (or about five percent of the total U.S. housing stock). Federal servers and data centers alone account for approximately 6 billion kWh (10 percent) of this electricity use, for a total electricity cost of about $450 million annually. The upward trend in power consumption by data centers is demonstrated by the fact that the energy use of the nation's servers and data centers in 2006 is estimated to be more than double the electricity that was consumed for this purpose in 2000 [EPA2007].

These power hungry data centers are posing a technical, financial and environmental threat to the economy and the society at large which leads to several issues.

1. First, high energy consumption leads to a high Total Cost of Ownership (TCO) for datacenter owners. For example, American Power Conversion (APC) [APC2005] computed the TCO data for a typical data center rack with the following characteristics:

Power rating: 100KW
Power density: 50W/sq ft
Life Cycle: 10 years
Average rack power: 1500W
Redundancy: 2N

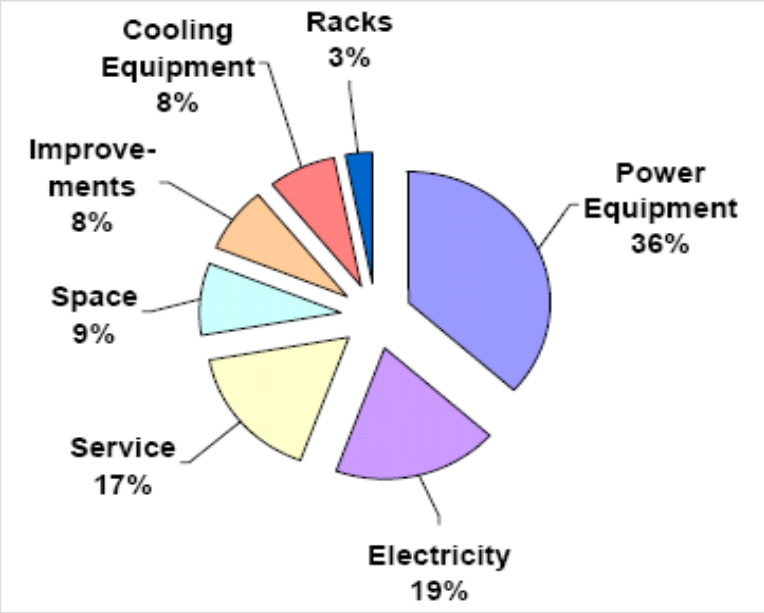

Figure 1-2: Breakdown of data center electrical requirements

They found that the TCO of a rack in a data center is approximately $120K over the data center lifetime. Approximately half of the lifetime per rack TCO of $120K is capital expense, and half is operating expense. These costs break down into categories as shown in Figure 1-2. Thus power equipment, cooling equipment, and electricity together are responsible for 63% of the TCO of the physical IT infrastructure of a data center.

2. The tightly packed server racks in data centers create high power density zones within the data center making cooling both expensive and complicated. Insufficient cooling leads to intermittent failures of computing nodes. Furthermore, constraints on the amount of power that can be delivered to server racks make energy conservation critical in order to fully utilize the available space on these racks [Chase et. al. 2001].

3. Finally electricity production also harms the environment. For example, excessive production of CO_2 from coal-burning power plants can significantly contribute to global warming [Chase et. al. 2001].

APC points out [APC2005] that the ideal data center or network room architecture would be "right-sized" and only incur the infrastructure costs that are actually required at a given time. To achieve the theoretically available cost savings, the ideal data center or network room architecture would only have the power and cooling infrastructure needed at the moment; it would only take the space that it needed at the moment, and it would only incur service costs on capital infrastructure capacity that was actually being used. It would be perfectly scalable. However, today's data centers are far from ideal. This is corroborated by the statistics presented in [White et. al. 2004] that reports about 3 watts of cooling and backup for every 1 watt of usage or average utilization of 20% with the remaining 80% over-provisioned for increased availability and rare peak loads.

The question then boils down to determine if it is possible to convert traditional statically provisioned, power hungry data centers to a more dynamic infrastructure with the objective of minimizing power consumption but providing equally good quality of service as that of traditional data centers. This would mean building a dynamic data center that can scale well to incoming traffic such that it handles peak traffic efficiently and is power efficient during off-peak hours.

1.2 Research Objectives

The goal of this research is to develop a theoretical and experimental framework and general methodology for autonomic power and performance management of high-

performance server platforms in Internet data centers to achieve (a) online modeling, monitoring, and analysis of power consumption and performance; (b) automatic detection of application execution phases and properties for a wide range of workloads and applications;(c) employing knowledge of application execution phases to dynamically identify strategies that minimize power consumption while maintaining the required quality of service (c) dynamically reconfigure the high-performance computing systems in data centers according to the selected optimization strategies; and (d) seek new and effective optimization techniques that can scale well and can perform in a dynamic and continuous fashion, making the system truly autonomic. Our methodology exploits emerging hardware and software standards for servers and devices that offer a rich set of hardware features for effective power management.

We consider power consumption as a first-class resource that is managed autonomously along with performance for high-performance computing systems. We incorporate power and performance management within an autonomic management framework that is modeled after the human autonomic nervous system. This management framework can be extended for the simultaneous management of multiple autonomic objectives such as – power, performance, fault and security.

1.3 Research Challenges

Automatic modeling and online analysis of multiple objectives and constraints such as power consumption and performance of high-performance computing platforms is a challenging research problem due to the continuous change in workloads and applications, continuous changes in topology, the variety of services and software

modules being offered and deployed, and the extreme complexity and dynamism of their computational workloads. In order to develop effective autonomic control and management of power and performance, it becomes highly essential for the system to have the functionality of online monitoring; adaptive modeling and analysis tailored towards real-time processing and proactive management mechanisms. This work develops innovative techniques to address the following research challenges:

1. How do we efficiently and accurately model power and energy consumption from a platform-level perspective that involves the complex interactions of different classes of devices such as processor, memory, network and I/O? Current research focuses on components in isolation such as processor, or memory or hard-disk to exploit power savings. A component view of power management does not provide much benefit when we look at power management of the whole platform because while this leads to the power-efficiency of that one component within the platform, the other components become the bottleneck in achieving platform power savings. A platform/system level view of these components would present effective opportunities for power savings since we can exploit the non-mutually exclusive behaviors of these components to set them to low-power states such that they collaboratively contribute to platform power management. The research challenge is to accurately model these complex interactions between the various device classes from various vendors. Modeling this accurately is a significant research and development challenge. We use the DEVS [Zeigler2000] formalism to model the platform components and their interactions. In this work we focus on the platform

memory and processor subsystems for platform power and performance management.

2. How do we predict in real-time, the behavior of platform resources and their power consumptions as workloads change dynamically by several orders of magnitude within a day or a week? We address this challenge by utilizing a novel concept called application execution flow (*Appflow*) that characterizes both resource requirements and the spatial and temporal behaviors of applications during their life-time for any resource class, computing, storage or network.
3. How to design "instantaneous" and adaptive optimization mechanisms that can continuously and endlessly learn, execute, monitor, and improve in meeting the collective objectives of power management and performance improvement? We have exploited mathematically rigorous optimization techniques to address this research challenge.

1.4 Overview of Research Approach

Traditional data center servers are statically configured and always over-provisioned to be able to handle peak load. In our approach, we proactively detect resource over-provisioning in data center servers and identify appropriate power and performance management strategies that reconfigure the server components to adapt to the incoming workload. The over-provisioned resources are then transitioned to low-power *state*s (supported by the hardware) and they continue to remain in that *state* until there is a surge in the workload that would require scaling-up the server capacity again. In this manner we reduce power consumption at the platform-level while meeting application

performance requirements. In this work we focus on two power-hungry server resources – the processor and memory subsystems. Our technique can be easily extended to encompass other server resources such as the hard disk and the NIC with the main differences being in the specifics of the hardware features supported by these devices for power and performance management and the nature of the workloads experienced by them.

We transform traditional statically configured data center servers to dynamically scalable/reconfigurable entities by designing autonomic extensions to classical data center servers that transform them into intelligent self-managing entities that scale-in and scale-out adaptively like clairvoyant entities that can sense the changes in the workloads at any time granularity and configure themselves appropriately to save power while maintaining performance. This involves accurate and efficient runtime monitoring and analysis of certain key attributes of the server platform that help predict trends and patterns in changes of the incoming workload. In our approach, the server power management algorithm maintains these key attributes within a threshold range in order to maintain the platform's power and performance. We quantify this with the *performance-per-watt* metric. Maximizing this metric maximizes the amount of performance that the platform can deliver for the same amount of 'watt' (power) consumed.

We address the problem very systematically by casting it within a theoretical and experimental framework and general methodology for hierarchical autonomic power and performance management of high-performance data centers servers. Our hierarchical management approach contributes to the development of real-time, light-weight, effective and scalable solutions. We optimize for power & performance (*performance-per-watt*) at

each level of the platform hierarchy using mathematically rigorous optimization techniques. At each hierarchy, power expenditure is minimized subject to associated performance constraints and it is formulated as a *performance-per-watt* maximization problem. Optimization solutions at the upper-level are refined at the lower-level and feedback from the lower-level flows back to the upper-level. In this manner the platform components cooperate and coordinate towards the common goal of reducing platform power while maintaining performance.

At any hierarchy, we model the *managed system* (platform, processor or memory) as a set of *states* and transitions. A *state* represents a specific configuration of the *managed system*. For example, on our experimental server unit (2 Dual-Core Intel™ Xeon processors, 5000P Memory Controller Hub, 8 GB DDR2 FBDIMM memory) that was used to validate our approach, the different memory configurations supported were 8 GB, 6 GB, 4 GB or 2 GB. A *state* is defined by fixed base power consumption and variable performance parameters. Whenever, a performance parameter exceeds a threshold value, the associated *autonomic manager* searches for a target *state* among all possible *managed system states* that consumes the minimum power while satisfying the performance constraints. This target *state* is obtained as the solution of the *performance-per-watt* optimization problem.

In this work we first develop an efficient power and performance management algorithm for the memory subsystem in a data center server platform. We then build on that work to encompass power and performance management of the whole platform. For the memory subsystem our technique has been validated on a real server using SPECjbb benchmark [SPECjbb2005] and on an open-source trace-driven memory simulator called

DRAMsim [Wang et. al. 2005] using ***gcc*** memory traces. On the server, our techniques are shown to give about 48.8% (26.7 kJ) energy savings compared to traditional techniques measured at 4.5%.The maximum improvement in *performance-per-watt* was measured at 88.48%. In the simulator our technique showed 89.7% improvement in *performance-per-watt* compared to the best performing traditional technique. For the server platform, we used the DEVS framework [Zeigler2000] Zeigler B. P., Praehofer H. and Kim T. G., *Theory of modeling and simulation*. 2nd ed. New York Academic Press, 2000.Zeigler2000] or modeling and simulation. We modeled a high-performance computing platform with multi-core processors, a unified L2 cache and a high-performance memory subsystem that mimics the memory architecture of our experimental server unit that was used to validate our technique related to memory power and performance management. Our experimental results showed 56.25% platform energy savings for memory-intensive workload, 63.75% savings for processor-intensive workload and 47.5% savings for a mixed workload.

1.5 Dissertation Organization

The rest of the dissertation is organized as follows. In Chapter 2, we review existing techniques for power management and discuss how this research field has evolved over time. We position our technique in the map of existing techniques and compare and contrast our work with existing techniques.

In Chapter 3, we lay the foundations of the autonomic computing paradigm. We then discuss how we apply the autonomic computing paradigm to design a theoretical framework and general methodology for joint management of power and performance of

high performance computing systems. We also introduce our *Appflow* methodology that is used by an autonomic computing system to dynamically predict application execution phases and identify effective power and performance management strategies to adapt to those phases.

In Chapter 4, we use this theoretical framework to design an autonomic data center that continuously strives to manage its power and performance. We present our model of a three-tier Internet data center with autonomic power and performance management at each hierarchy. In the following Chapters we focus on the platform and device levels of the data center hierarchy for power and performance management.

In Chapter 5, we focus on the power and performance management of the memory subsystem within a data center server platform. We present a novel adaptive interleaving technique that employs dynamic data migration and incorporates knowledge about the underlying memory architecture to deliver as much as 89.7% improvement in *performance-per-watt* compared to the best performing traditional technique.

In Chapter 6, we build on the work related memory power and performance management to encompass the whole platform that is composed of multi-core processors and multi-rank memory subsystem. We discuss our modeling and simulation approach that is developed using the Discrete Event Modeling and Simulation (DEVS) Environment. Our experimental results show 56.25% platform energy savings for memory-intensive workload, 63.75% savings for processor-intensive workload and 47.5% savings for a mixed workload.

In Chapter 7, we conclude this dissertation and give future research directions.

2 BACKGROUND AND RELATED WORK

2.1 Introduction

As shown in Figure 2-1, power management techniques can be broadly classified into software techniques and hardware techniques. Hardware techniques deal with hardware design of power-efficient systems such as low-leakage power supplies, power-efficient components such as transistors, DRAMs and so on. Software techniques on the other hand, deal with power management of hardware components (processor, memory, hard disk, network card, display etc) within a system – such as a server, laptop, PDA or system-of-systems such as e-commerce data centers, by transitioning the hardware components into one of the several different low-power *state*s when they are idle for a long period of time. This technique is also known as Dynamic Power Management (DPM). DPM is by far the most popular software power management technique and can be grouped into three most discernable sub-classes as shown in Figure 2-1 – predictive techniques, heuristic techniques and QoS and energy trade-offs. The main distinction between these techniques is in the manner in which they determine when to trigger power *state* transitions for a hardware component from a high-power to a low-power *state* or vice-versa. Whereas heuristic techniques are more adhoc and static in their approach, predictive techniques employ simple to sophisticated predictive mechanisms to determine how long into the future the hardware component is expected to stay idle and use that knowledge to determine when to reactivate the component into a high power *state* so that it can service jobs again. Recently researchers have started looking into QoS and energy trade-offs as a DPM technique to determine how long a hardware component can be put

to ‘sleep’ such that it does not hurt the performance seen by the applications. This is a more aggressive power management technique that takes advantage of the fact that “acceptable performance degradations” can be employed to let the hardware components sleep longer and hence save more power. This is specifically true for power-hungry domains such as huge data centers that have specific Service Level Agreement (SLA)

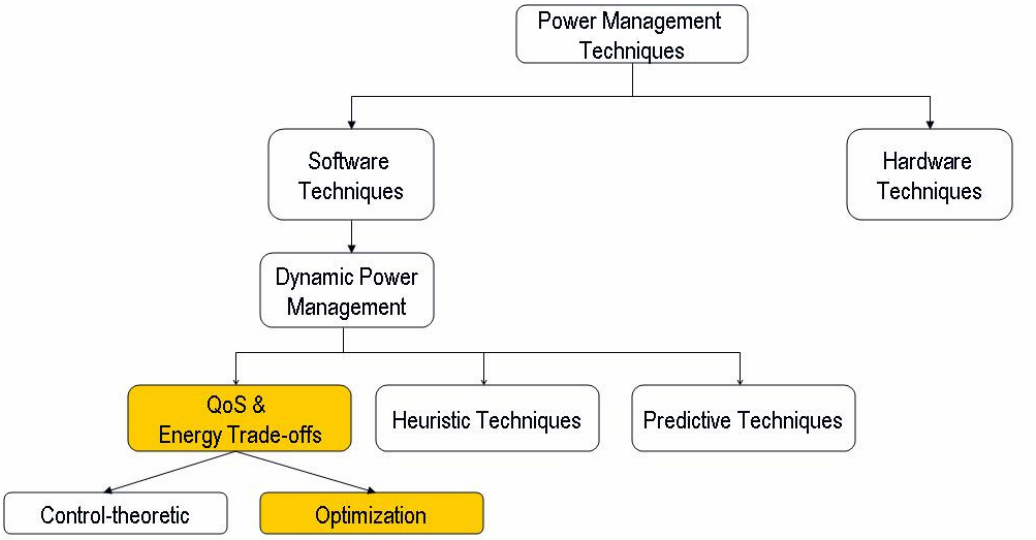

Figure 2-1: Our approach in relation to existing power management techniques

contracts with their customers that clearly define the “acceptable performance degradations” such as 97% response time, .05% MTBF (mean time between failures) and so on. This concept is depicted in Figure 2-2, where the data center size (and hence the power consumed) is always configured a little below what is required by the incoming traffic such that at that configuration the perceivable degradation in performance is still acceptable.

As shown in Figure 2-1, QoS and energy trade-off DPM techniques can be either Control-theoretic or Optimization based. Our approach to power and performance (QoS) management is Optimization based. This is depicted in the yellow shaded boxes in Figure 2-1. We formulate the power and performance management problem as an optimization

problem, where we constantly maintain the hardware components in a power *state* such that they consume the minimal possible power while maintaining the performance within the acceptable threshold bounds.

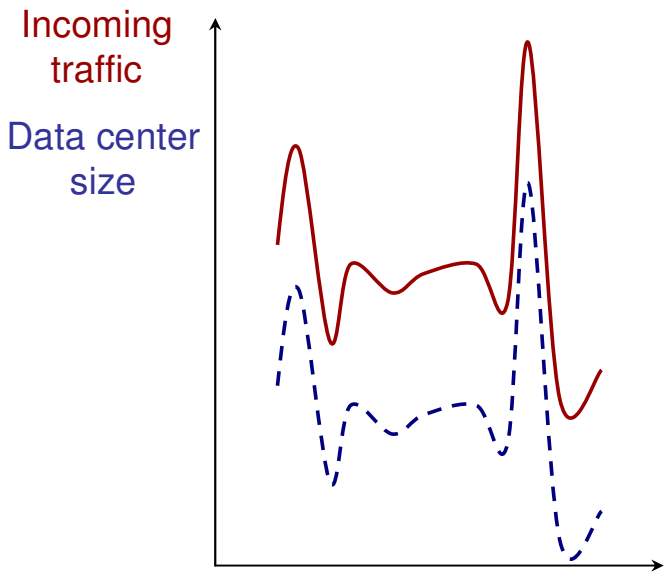

Figure 2-2: Amplitude lag between data center size and incoming traffic

Software power management techniques can also be subdivided based on where they reside within the system. This is shown in Figure 2-3. They could be power-aware applications, power-aware compilers that perform optimizations on the code for power efficiency, power-aware operating systems that employ the DPM techniques mentioned earlier. And then we have the hardware design optimizations for power-efficiency sitting at the very bottom. Note that, it is possible for a high-level technique to build on top of low-level techniques or share borders across techniques as shown by the dotted boxes in Figure 2-3. For example, power-aware compilers can perform code transformations for power-efficiency such that the targeted hardware component can sleep longer. The power-aware operating system then triggers the transition of the hardware component into the low-power sleep *state* and reactivates it back when required. Our technique depicted by

the yellow shaded box in Figure 2-3, resides under the cover of the operating system, very close to the hardware, while sharing some borders with it.

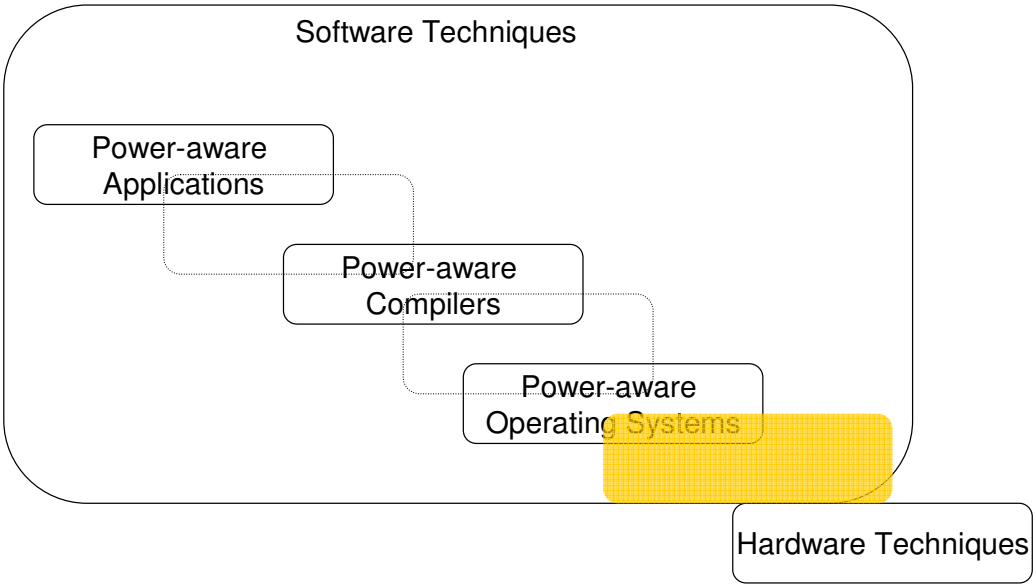

Figure 2-3: Location-based classification of software power management techniques

In what follows, we chart the territory of Dynamic Power Management (DPM) by presenting existing research in this field. We also present our technique and compare it with existing techniques.

2.2 Applications of Dynamic Power Management (DPM)

Most software power management techniques exploit the over-provisioning of components, devices or platforms for power savings. This technique, also known as Dynamic Power Management (DPM), is extensively used for reducing power dissipation in systems by slowing or shutting-down components when they are idle or underutilized. DPM techniques can be used in isolation for power management of a single system component such as the processor, system memory or the NIC card. They can also be used for joint power management of multiple system components or power management of the

whole system. Our technique employs DPM for power management from a whole system perspective by power managing multiple system components in a co-operative fashion.

2.2.1 Power Management of System Components in Isolation

Most DPM techniques utilize power management features supported by the hardware. For example, frequency scaling, clock throttling, and dynamic voltage scaling (DVS) are three processor power management techniques that are extensively utilized by DPM 10. [Shin et. al. 2004] Shin H. and Lee J., Application Specific and Automatic Power Management based on Whole Program Analysis, *Final Report,* http://cslab.snu.ac.kr/~egger/apm/final-report.pdf, August 20, 2004. Shin et. al. 2004] or example, extends the operating system's power manager by an *adaptive Power Manager* that uses the processor's DVS capabilities to reduce or increase the CPU frequency thereby minimizing the overall energy consumption. [Elnozahy et. al. 2002] combines the DVS technique at the processor-level together with a server turn on/off technique at the cluster-level to achieve high power savings while maintaining the response time for server clusters. [Pinheiro et. al. 2001] introduces a scheme to concentrate the workload on a limited number of servers in a cluster such that the rest of the servers can remain switched-off for a longer time. [Sharma et. al. 2003] proposes power-aware QoS management in web servers where the algorithms reduce processor voltage and frequency as much as possible but not enough to cause per-class response time constraint violations. Other techniques use a utilization bound for schedulability of a-periodic tasks [Abdelzaher et. al. 2003] [Abdelzaher et. al. 2001] to maintain the timeliness of processed jobs while conserving power.

Similarly, for dynamic memory power management [Lebeck et. al. 2000] uses multiple power modes of RDRAM memory and dynamically turns off memory chips with power-aware page allocation in operating system. Hard disk dynamic power management also requires them to support multiple disk rotation speeds, a technique that did not remain hugely popular mainly due to performance issues. For example, disk timeout determines how long a hard disk must be idle before it spins down [PC Guide2001]. [Douglis et. at. 1994] uses periods of inactivity between disk accesses to determine if the disk can be transitioned to a low power *state*.

2.2.2 Joint Power Management of System Components

Researchers have also explored joint power management techniques that involve techniques to jointly maintain power consumption of multiple system components such as the memory and the hard disk. For example, [Cai et. al. 2005] has used the relationship between memory and disk (smaller the memory size, the higher the page misses and the higher the disk accesses) to achieve power savings by proactively changing disk I/O by expanding or contracting the size of the memory depending on the workload. They minimize power consumption by computing the optimal values for disk timeout and memory size dynamically at runtime under varying workloads. However they do not consider the impact of their scheme on performance. [Felter et. al. 2005] addresses base power consumption for web servers by using a *power-shifting* technique that dynamically distributes power among components using work-load sensitive polices. They use the processor and memory power-budget redistribution to achieve that objective.

2.2.3 Holistic System-level Power Management

Most techniques for dynamic power management justify the need to consider components in isolation. For example, [Bohrer et. al. 2002] makes the case that processor is the major power consuming factor in servers. Following this thread [Elnozahy et. al. 2003] presents a request-batching scheme where jobs are forwarded to the processor in a batch after a certain time such that the response time constraint is met for all classes of customers. This lets the processor be in a lower power *state* for a longer period of time and process the jobs in the batch at a later time. [Zhu et. al. 2004] on the other hand states that data center storage devices can consume over 25% power. Instructions invoking memory operations have a relatively high power cost, both within the processor and in the memory system [Tiwari et. al.1994]. This has spawned research in memory power management. However there has not been much effort to exploit these existing techniques for different classes of resources (processor, memory, cache, disk, network card etc) in a unified framework from a whole system/platform perspective. While the closest to combining device power models to build a whole system has been presented in [Gurumurthi et. al. 2002], our approach aims at building a general framework for autonomic power and performance management where we bring together and exploit existing device power management techniques from a whole system's perspective. We extend it for power and performance management of a high-performance server platform within a data center. We introduce a hierarchical framework for power management that starts at individual devices within a server to server clusters and cluster of clusters enabling power management at every level of the hierarchy of a data center. The power management solutions are more and more refined as we travel down the hierarchy from

cluster of heterogeneous servers to independent devices. The closest to our approach is the work done by [Rong et. al. 2005] that solves the problem of hierarchical power management for an energy managed computer (EMC) system with self-power managed components while exploiting application level scheduling.

2.3 Classifications of DPM Techniques

DPM techniques can be predictive, heuristic or they may involve QoS trade-offs for energy savings.

2.3.1 Heuristic and Predictive Techniques

Heuristic-based approaches [Srivastava et. al.1996], [Hwang et. al.1997], [Hsu et. al. 2003], [Weiser et. al. 1994] employ simple heuristics to transition a system component to a low-power mode after it is observed to have remained idle for a pre-determined period of time. [Delaluz et. al. 2001] proposed various threshold predictors to determine the maximum amount of time that a memory module must remain idle before it is transitioned back to a low power *state*. [Fan et. al. 2001] investigated memory controller policies in cache-based systems and concluded that the simple policy of immediately transitioning the DRAM chip to a lower power *state* when it becomes idle is superior compared to more sophisticated policies that try to predict the idle time of the DRAM chip.

Researchers have also looked at co-operative hardware-software schemes for Dynamic Power Management. [Lebeck et. al. 2000] studied page allocation techniques to cluster an application's *pages* onto a minimum number of memory modules thereby

increasing the idleness for the other modules. [Zhou et. al. 2004] used such page allocation schemes combined with the page Miss Ratio Curve metric to determine the optimal memory size that would give the maximum possible *hit ratio* for the application. [De La Luz et. al. 2002] proposed a scheduler-based policy that uses prior knowledge of memory modules used by a specific process to allocate the same memory modules the next time the process is scheduled. [Huang et. al. 2003] built on this idea to develop Power-Aware Virtual Memory [PAVM] where the OS and the Memory Controller communicate to enhance memory energy savings by leveraging NUMA memory infrastructure to reduce energy consumption on a per-process basis. [Huang et. al. 2005] used a similar idea to actively reshape memory traffic to aggregate idle periods.

2.3.2 QoS and Energy Trade-offs

While heuristic and predictive DPM techniques take into consideration the performance attribute in their energy saving schemes, research that has focused on QoS trade-offs for energy savings has specifically studied the impact of power savings on performance. They often bind the power and performance problem in more mathematically rigorous formulations providing statistical guarantees of performance for their power management schemes. They investigate and generate power saving opportunities that can be obtained at the cost of QoS trade-offs within acceptable limits.

This has given rise to the application of proactive mathematically rigorous optimization techniques as well as reactive control theoretic techniques for power management while maintaining performance. For example [Paleologo et. al. 1999], [Qiu et. al. 2001], [Chung et. al. 2002], [Simunic2002] have developed a myriad of stochastic

optimization techniques for portable devices. In the server domain, [Chen et. al. 2005] has presented three online approaches for server provisioning and DVS control for multiple applications – namely a predictive stochastic queuing technique, reactive feedback control technique and another hybrid technique where they use predictive information for server provisioning and feedback control for DVS. [Sharma et. al. 2003] has studied the impact of reducing power consumption of large server installations subject to QoS constraints. They developed algorithms for DVS in QoS-enabled web servers to minimize energy consumption subject to service delay constraints. They used a utilization bound for schedulability of a-periodic tasks [Abdelzaher et. al. 2003] to maintain the timeliness of processed jobs while conserving power. [Mastroleon et. al. 2005] investigated autonomic power control policies for internet servers and data centers. They used both the system load and thermal status to vary the utilized processing resources to achieve acceptable delay and power performance. They used Dynamic Programming to solve their optimization problem. [Lefurgy et. al. 2007] presented a technique that controls the peak power consumption of a high-density server by implementing a feedback controller that performs precise system-level power measurements to periodically select the highest performance *state* while keeping the system within a fixed power constraint. [Li et. al. 2004] proposed a *Performance-directed Dynamic* (PD) algorithm that dynamically adjusts the thresholds for transitioning devices to low-power *state*s, based on available slack and recent workload characteristics. A departure to this approach is provided by the work of that showed that limiting power is as effective an energy-conservation approach as techniques explicitly designed for performance-aware energy conservation.

2.4 Discussions of Our Technique

We formulate the problem as a system *performance-per-watt* optimization problem. We first solve this problem for the fully-interleaved memory subsystem in a high-performance server platform. We then build on this work to encompass power and performance management from the whole system perspective. We solve the system *performance-per-watt* optimization problem first for the whole platform and then for the platform components (processor and memory in our case) with a hierarchical refinement of solutions from the platform-level to the individual component level.

2.4.1 Adaptive Memory Interleaving Technique for Power & Performance Management

Our memory management scheme can be viewed as a dynamic memory interleaving technique that is adaptive to incoming workload in a manner that reduces memory energy consumption while maintaining the performance at an acceptable level. Our algorithm addresses power/performance management of interleaved memory systems where current techniques cannot be applied.

1. It uses migration to dynamically reduce the size of the interleaving (m-way to n-way, where m>n) in order to reduce power while maintaining performance.
2. It migrates cache lines that belong to *working set pages* to meet application memory requirements. The working set is predicted using the MRC metric from [Zhou et. al. 2004]
3. Our scheme exploits the internal memory architecture (that consists of branch, channel, rank etc) in doing the migration.

4. It is application and OS independent.

5. It can be coupled with any suitable fine-grained power management technique.

Although in the same domain, our work is distinctly different from the work of [Zhou et. al. 2004]. They power down memory chips that do not contain predicted *working set pages*. Our work is different from them because of points 1, 2, 4 above. Similarly, our work is distinctly different from [De La Luz et. al. 2002]. They migrate arrays in multi-bank memory based on temporal locality. Our work is different from them because of points 1, 2, 3, 4, 5 above.

Next we take the memory work and combine it with multi-core processor management for power and performance management of the whole platform.

2.4.2 Power & Performance Management of Server Platforms

The synergy between system components (processor and memory) has been clearly presented in the work of [Fan et. al. 2003]. [Felter et. al. 2005] addresses base power consumption for web servers by using a *power-shifting* technique that dynamically distributes and maintains power budget among components using workload sensitive polices. In our approach, we maintain the performance instead, while aggressively reducing platform power consumption. We apply this technique at different hierarchies of a high-performance server platform going from devices, to the whole server. This makes our approach more comprehensive as compared to [Rong et. al. 2005]. Also, our hierarchical approach to power and performance management lends itself very well to the server cluster and data center domain in terms of maintaining the scalability of the solutions.

2.4.3 Autonomic Management Framework for Joint Management of Power & Performance

Our generic autonomic management framework not only enables us to experiment with different types of power management techniques ranging from heuristic approaches to stochastic optimization techniques but also provides a mechanism to consolidate power management with other autonomic management objectives pertinent to data centers such as – fault-tolerance, security, reliability, thermal etc. In this work we demonstrate the usability of the framework for joint management of power and performance.

3 THE AUTONOMIC COMPUTING PARADIGM

3.1 Introduction

Autonomic computing is a programming paradigm and management technique that is based on strategies used by biological systems to deal with complexity, dynamism, heterogeneity and uncertainty. It is inspired by the human autonomic nervous system that handles complexity and uncertainties, and it aims at realizing computing systems and applications capable of managing themselves with minimum human intervention.

In this Chapter we first discuss the mechanisms of the human autonomic nervous system and use it to motivate our approach to develop an autonomic computing system. Our approach builds on Ashby's ultra-stable system in his study of the design for a brain [Ashby1960]. We then illustrate how an autonomic computing system jointly manages its power and performance.

3.2 Motivations: The Human Autonomic Nervous System

The human nervous system is, to the best of our knowledge, the most sophisticated example of autonomic behavior existing in Nature today [Kimball2007]. It is the body's master controller that monitors changes inside and outside the body, integrates sensory input, and effects appropriate response. In conjunction with the endocrine system, which is the body's second important regulating system, the nervous system is able to constantly regulate and maintain homeostasis. Homeostasis is one of the most remarkable properties of highly complex systems. A homeostatic system (a large organization, an industrial firm, a cell) is an open system that maintains its structure and functions by means of a

multiplicity of dynamic equilibriums that are rigorously controlled by interdependent regulation mechanisms. Such a system reacts to every change in the environment, or to every random disturbance, through a series of modifications that are equal in size and opposite in direction to those that created the disturbance. The goal of these modifications is to maintain internal balances. The manifestation of the phenomenon of homeostasis is wide-spread in the human system. As an example consider the mechanisms that maintain the concentration of glucose in the blood within limits—if the concentration should fall below about 0.06 percent, the tissues will be starved of their chief source of energy; if the concentration should rise above about 0.18 percent, other undesirable effects will occur. If the blood-glucose concentration falls below about 0.07 percent, the adrenal glands secrete adrenaline, which causes the liver to turn its stores of glycogen into glucose. This passes into the blood and the blood-glucose concentration drop is opposed. Further, a falling blood-glucose also stimulates the appetite causing food intake, which after digestion provides glucose. On the other hand, if the blood-glucose concentration rises excessively, the secretion of insulin by the pancreas is increased, causing the liver to remove the excess glucose from the blood. Muscles and skin also remove excess glucose and if the blood-glucose concentration exceeds 0.18 percent, the kidneys excrete excess glucose into the urine. Thus, there are five activities that counter harmful fluctuations in the blood glucose concentration [Ashby1960].

The above example focuses on the maintenance of the blood-glucose concentration within safe or operational limits that have been 'predetermined' for the species. Similar control systems exist for other parameters such as systolic blood pressure, structural integrity of medulla oblongata, severe pressure of heat on the skin and so on. All these

parameters have a bearing on the survivability of the organism, which in this case is the human body. However, all parameters are not uniform in their urgency or their relations to lethality. Parameters that are closely linked to survival and closely linked to each other so that marked changes in one leads sooner or later to marked changes in the others, have been termed as essential variables by Ashby in his study of the design for a brain [Ashby1960] which is discussed below.

3.2.1 Ashby's Ultra-stable System

Every real machine embodies no less than an infinite number of variables, and for our discussion we can safely think of the human system as represented by a similar sets of variables, of which we will consider a few. In order for an organism to survive, its essential variables must be kept within viable limits .This is shown in Figure 3-1. Otherwise the organism faces the possibility of disintegration and/or loss of identity (dissolution or death) [AdSys].

The body's internal mechanisms continuously work together to maintain the essential variables within their viable limits. Ashby's definition of adaptive behavior as demonstrated by the human body follows from this observation.

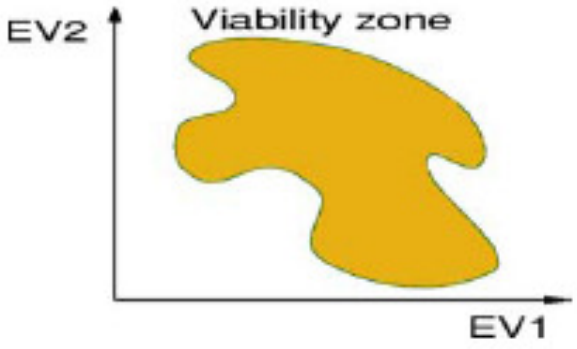

Figure 3-1: Essential variables

He states that a form of behavior is adaptive if it maintains the essential variables within physiological limits [Ashby1960] that define the viability zone. Two important observations can be made:

I. The goal of the adaptive behavior is directly linked with the survivability of the system.

II. If the external or internal environment pushes the system outside its physiological equilibrium state the system will always work towards coming back to the original equilibrium state.

Ashby observed that many organisms undergo two forms of disturbance: (1) frequent small impulses to main variables and (2) occasional step changes to its parameters. Based on these observations he devised the architecture of the ultra-stable system that consists of two closed loops: one that can control small disturbances while the second control loop is responsible for larger disturbances. This is shown in Figure 3-2.

As shown in Figure 3-2, the ultra-stable system consists of two subsystems, the environment and the reacting part (*R*).*R* represents a subsystem of the organism that is responsible for overt behavior or perception. It uses the sensor channels as part of its

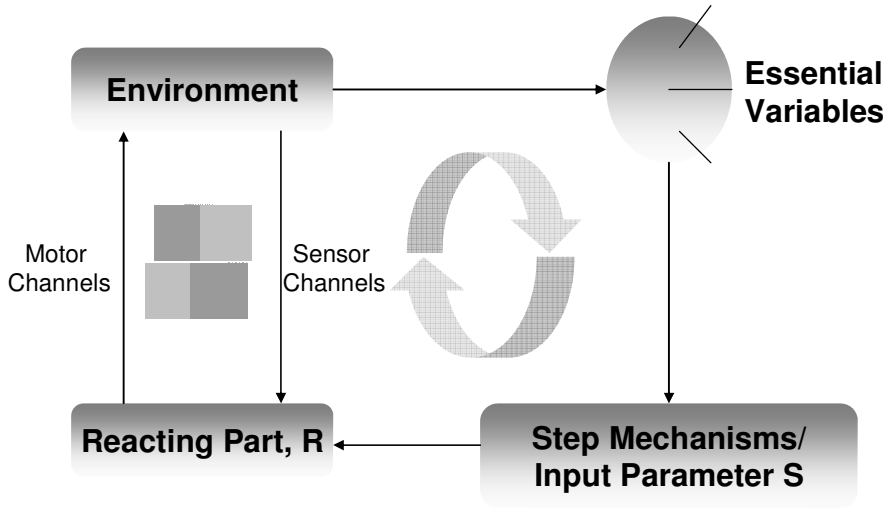

Figure 3-2: The ultra-stable system architecture

perception capability and motor channels to respond to the changes impacted by the environment. These set of sensor and motor channels constitute the primary feedback between *R* and the environment. We can think of *R* as a set of behaviors of the organism that get triggered based on the changes affected by the environment. *S* represents the set of parameters that trigger changes in relevant features of this behavior set. Note that in Figure 3-2, *S* triggers changes only when the environment affects the essential variables in a way that causes them to be outside their physiological limits. As mentioned above, these variables need to be maintained within physiological limits for any adaptive system/organism to survive. Thus we can view this secondary feedback between the environment and *R* as responsible for triggering the adaptive behavior of the organism. When the changes impacted by the environment on the organism are large enough to throw the essential variables out of their physiological limits, the secondary feedback becomes active and changes the existing behavior sets of the organism to adapt to these new changes. Notice that any changes in the environment tend to push an otherwise stable system to an unstable state. The objective of the whole system is to maintain the subsystems (the environment and *R*) in a state of stable equilibrium. The primary feedback handles finer changes in the environment with the existing behavior sets to bring the whole system to stable equilibrium. The secondary feedback handles coarser and long-term changes in the environment by changing its existing behavior sets and eventually brings back the whole system to stable equilibrium state. Hence, in a nutshell, the environment and the organism always exist in a state of stable equilibrium and any activity of the organism is triggered to maintain this equilibrium.

3.2.2 The Nervous System as an Ultra-stable System

The human nervous system is adaptive in nature. In this Section we apply the concepts of Ashby's ultra-stable system to the human nervous system. The goal is to enhance the understanding of an adaptive system and help extract essential concepts that can be applied to lay the foundations of an autonomic computing system.

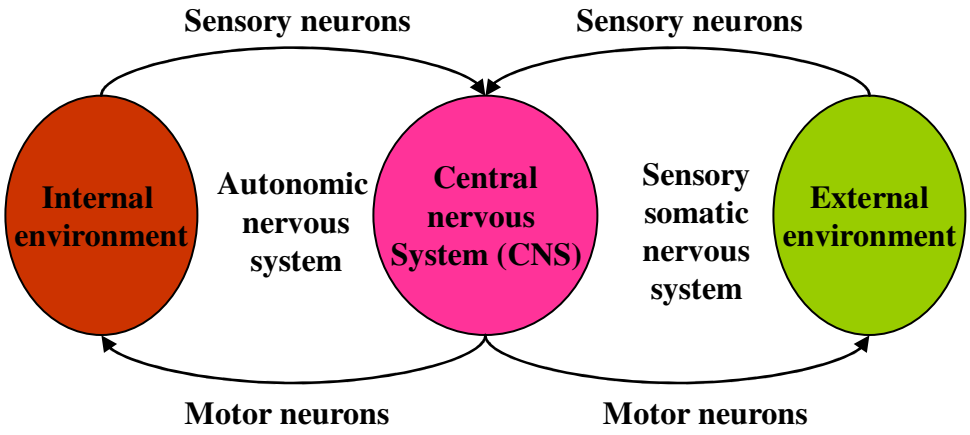

Figure 3-3: Organization of the nervous system [Kimball2007]

As shown in Figure 3-3, the nervous system is divided into the Peripheral Nervous System (PNS) and the Central Nervous System (CNS). The PNS consists of sensory neurons running from stimulus receptors that inform the CNS of the stimuli and motor neurons running from the CNS to the muscles and glands, called effectors, take action. CNS is further divided into two parts: sensory-somatic nervous system and the autonomic nervous system. Figure 3-4 shows the architecture of the autonomic nervous system modeled after Ashby's ultra-stable system.

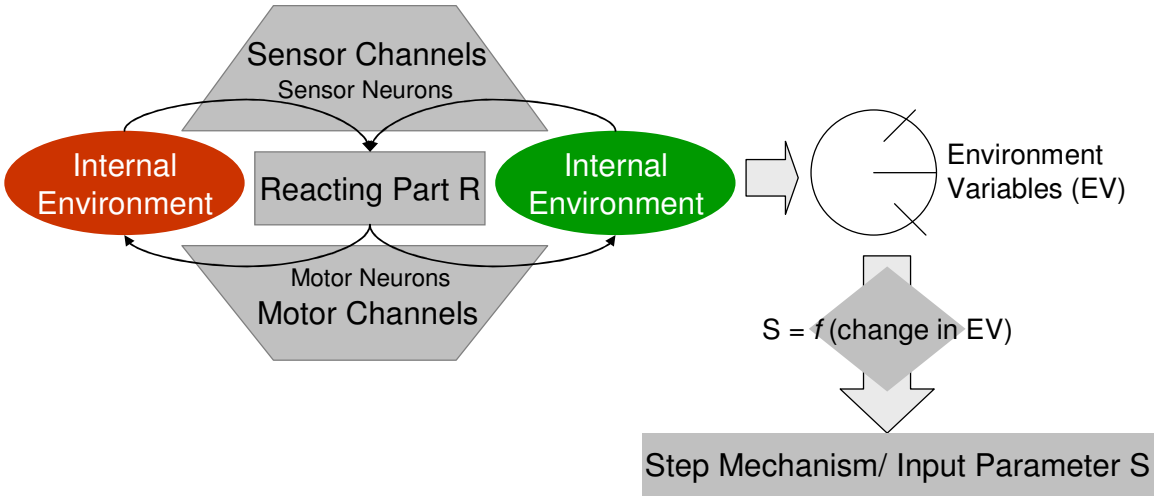

Figure 3-4: Nervous system as part of an ultra-stable system

As shown in Figure 3-4, the sensory and motor neurons constitute the sensor and motor channels of the ultra-stable system. The triggering of essential variables, selection of the input parameter *S* and translation of these parameters to the reacting part *R* constitute the workings of the Nervous System. Revisiting the management of blood-glucose concentration within physiological limits discussed earlier, the five mechanisms that get triggered when the essential variable (concentration of glucose in blood) goes out of the physiological limits change the normal behavior of the system such that the reacting part *R* works to bring the essential variable back within limits. It uses its motor channels to effect changes so that the internal environment and the system (organism) come into the state of stable equilibrium. It should be noted that the environment here is divided into the internal environment and external environment. The internal environment represents changes impacted internally within the human system and the external environment represents changes impacted by the external world. However, the goal of the organism is to maintain the equilibrium of the entire system where all the

subsystems (the organism and the internal and external environments) are in stable equilibrium.

3.2.3 The Autonomic Computing Paradigm

The autonomic computing paradigm, modeled after the autonomic nervous system, must have a mechanism whereby changes in its essential variables (e.g., performance, fault, security, etc.) can trigger changes to the behavior of the computing system such that the system is brought back into equilibrium with respect to the environment. This state of stable equilibrium is a necessary condition for the survivability of the organism. In the case of an autonomic computing system, we can think of survivability as the system's ability to protect itself, recover from faults, reconfigure as required by changes in the environment and always maintain its operations at a near optimal performance. Both the internal environment (e.g. excessive CPU utilization, high power consumption) and the external environment (e.g. protection from an external attack, spike in incoming workload) impact its equilibrium. The autonomic computing system requires sensor channels to sense the changes in the internal and external environment and motor channels to react to the changes in the environment by changing itself so as to counter the effects of changes in the environment and maintain equilibrium. The changes sensed by the sensor channels have to be *analyzed* to determine if any of the essential variables has gone out of their viability limits. If so, it has to trigger some kind of *planning* to determine what changes to inject into the current behavior of the system such that it returns to equilibrium state within the new environment. This planning would require *knowledge* to select just the right behavior from a large set of possible behaviors to

counter the change. Finally, the motor neurons *execute* the selected change. 'Sensing', 'Analyzing', 'Planning', 'Knowledge' and 'Execute' are in fact the keywords used to identify an autonomic system [Kephart et. al. 2003]. In what follows, we enlist the properties of an autonomic computing system. We then present the architecture of an autonomic computing system that provides key autonomic middleware services to support their self-managed execution.

3.3 Properties of an Autonomic Computing System

The properties of an autonomic computing system have been summarized below [Kephart et. al. 2003], [Horn2001]:

1. *Self-Awareness*: an autonomic system knows itself and is aware of its state and its behaviors.

2. *Self-Protecting*: an autonomic system is equally prone to attacks and hence it should be capable of detecting and protecting its resources from both internal and external attack and maintaining overall system security and integrity.

3. *Self-Optimizing*: an autonomic system should be able to detect performance degradation in system behaviors and intelligently perform self-optimization functions.

4. *Self-Healing*: an autonomic system must be aware of potential problems and should have the ability to reconfigure itself to continue to function smoothly.

5. *Self-Configuring*: an autonomic system must have the ability to dynamically adjust its resources based on its state and the state of its execution environment.

6. *Contextually Aware*: an autonomic system must be aware of its execution environment and be able to react to changes in the environment.

7. *Open*: an autonomic system must be portable across multiple hardware and software architectures, and consequently it must be built on standard and open protocols and interfaces.

8. *Anticipatory*: an autonomic system must be able to anticipate, to the extent that it can, its needs and behaviors and those of its context, and be able to manage itself proactively.

Sample autonomic system/application behaviors include installing software when it detects that the software is missing (self-configuring), restart a failed element (self-healing), adjust current capacity when it observes an increase in workload and reduce power consumption (self-optimizing) and take resources offline if it detects that these resources are compromised by external attacks (self-protecting).

Each of the attributes listed above are active research areas towards realizing autonomic systems and applications. Generally, autonomic computing addresses these issues in an integrated manner, i.e., configuration, optimization, protection, and healing. In this work we design an autonomic computing system that addresses the combined objective of power and performance management for computing systems. Further, autonomic management solutions typically consists of the steps outlined below (1) the application and underlying information infrastructure provides information to enable context and self awareness; (2) system/application events trigger analysis, deduction and planning using system knowledge; and (3) plans are executed using the adaptive capabilities of the system. An autonomic system implements self-managing attributes using the control loops described above to collect information, makes decisions and adapt as necessary.

3.4 Autonomic Computing System: The Conceptual Architecture

Figure 3-5 shows the architecture of an autonomic computing system from the conceptual point of view. This architecture directly derives from Ashby's ultra-stable system shown in Figure 3-2. It consists of the following modules.

3.4.1 High-performance Computing Environment

This comprises of the high performance computing system and their environment. The autonomic system is geared towards self-control and self-management of the high performance computing environment.

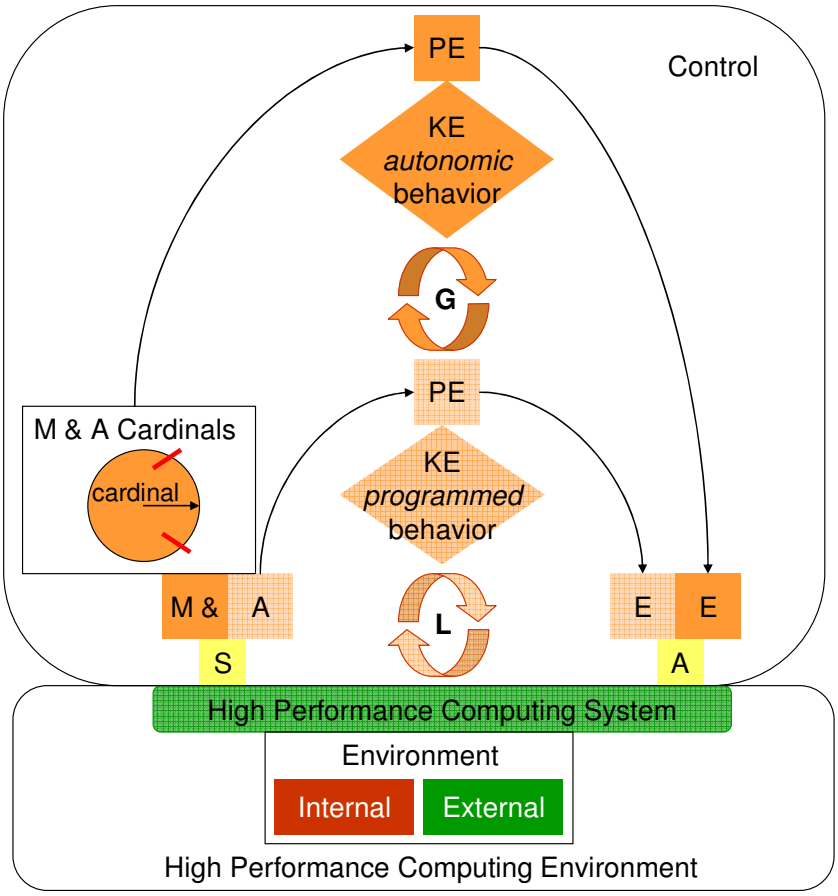

Figure 3-5: Autonomic computing system - the conceptual view

3.4.1.1 Environment

The environment represents all the factors that can impact the high performance computing system. The environment and the high performance computing system can be two subsystems forming a stable system. Any change in the environment causes the whole system to go from a stable state to an unstable state. This change is then offset by reactive changes in the high performance computing system causing the system to move back from the unstable state to a different stable state. Notice that the environment consists of two parts—internal and external. The internal environment consists of changes internal to the system that characterizes the runtime state of the system. The external environment can be thought of as characterizing the state of the incoming workload executed by the high performance computing system.

3.4.2 Control

At runtime, the high performance computing environment can be affected in different ways, for example, it can encounter a failure during execution, it can be externally attacked or it may slow down and affect the performance of the entire application. It is the job of the Control to manage such situations as they occur at runtime. The Control has the following engines to execute its functionalities:

1. *Monitoring and Analysis Engine* (*M&A*): Monitors the state of the high performance computing environment through its sensors and analyzes the information.

2. *Planning Engine* (*PE*): Plans alternate execution strategies (by selecting appropriate actions) to optimize the behaviors (e.g. to self-heal, self-optimize, self-protect etc.) and operations of the high performance computing environment.

3. *Knowledge Engine* (*KE*): Provides the decision support to the Control to pick up the appropriate rule from a set of rules to improve the performance. The Control realizes its control and management objectives with the aid of two closed loop control systems, triggered by application and system sensors (reactive) and online predictive performance models (proactive), and will manage and optimize application execution (Figure 3-5).

3.4.2.1 Local Control Loop

The local or fine control loop will locally manage the behavior of individual and local subsystems within the high performance computing system. This can be viewed as adding self-managing capabilities to conventional subsystems. This loop will control local algorithms, resource allocation strategies, distribution and load balancing strategies, etc. Note that this loop will only handle *known* environment states and the mapping of environment states to behaviors is encapsulated in its *knowledge engine* (*KE*). For example, when the load on the system resources exceeds the acceptable threshold value, the fine loop control will balance the load by either controlling the local resources or by reducing the size of the computational loads. This will work only if the local resources can handle the computational requirements. However, the fine loop control is *blind* to the overall behavior of the whole High Performance Computing System and thus cannot achieve the desired overall objective for the whole system. Thus by itself it can lead to less than acceptable behavior.

3.4.2.2 Global Control Loop

At some point, one of the essential variables of the system eventually exceeds its limits that will trigger the global control loop. The global control loop will manage the

behavior of the overall high performance computing system and will define the knowledge that will drive the adaptations/reconfigurations of the local subsystems. This control loop can handle *unknown* environment states and uses four cardinals for monitoring and analysis of the high performance computing environment i.e., performance, fault-tolerance, configuration and security. These cardinals are analogous to the essential variables described in Ashby's ultra stable system model of the autonomic nervous system. This control loop acts towards changing the existing behavior of the high performance computing environment such that it can adapt itself to changes in the environment. For example, in the previous load-balancing scenario, the existing behavior of the high performance computing environment (as directed by the local loop) was to maintain its local load within prescribed limits. Doing so blindly may degrade the performance of the overall system. This change in the overall performance cardinal triggers the global loop. The global loop then selects an alternate behavior pattern from the pool of behavior patterns for this high performance computing environment. The Planning Engine (PE) determines the appropriate plan of action using its Knowledge Engine (KE). Finally the Execution Engine (EE) executes the new plan on the high performance computing environment in order to adapt its behavior to the new environment conditions.

The most important feature of the autonomic system is the integrated approach of its controller. The controller unit manages in an integrated manner performance, fault, security and configuration of computing systems and their applications. In the classical paradigm, each of these properties is treated separately and in isolation. These practices

have contributed significantly to the control and management challenges of large scale interacting and dynamic computing systems and services. In the next section, we describe our approach to implement an autonomic computing system based on the conceptual architecture.

3.5 Notion of an Autonomic Computing System

Figure 3-6 illustrates an autonomic computing system as inspired by the Ashby's ultra-stable system architecture. It is a self-contained system with specified input/output interfaces and explicit context dependencies. It also has embedded mechanisms for self-management responsible for providing functionalities, exporting constraints, managing its own behavior in accordance with context and policies, and interacting with other autonomic systems. In summary, it is a system augmented with intelligence to manage and maintain itself under changing circumstances impacted by the internal or external environment. It consists of the *managed system (MS)* which is a representative of the high-performance computing environment and the *autonomic manager* (*AM*) is the control element described earlier. Owing to changes in the internal or external environment, the *managed system* may drift out of its operational requirements such that it challenges the "survivability" of the *managed system*. The *autonomic manager* continuously monitors and executes appropriate actions to maintain the *managed system* within its operational bounds. In the case of an autonomic computing system, "survivability" is defined by the system's ability to protect itself, maintain its power and performance, recover from faults, reconfigure as required by changes in the environment and always maintain its operations at a desired performance level.

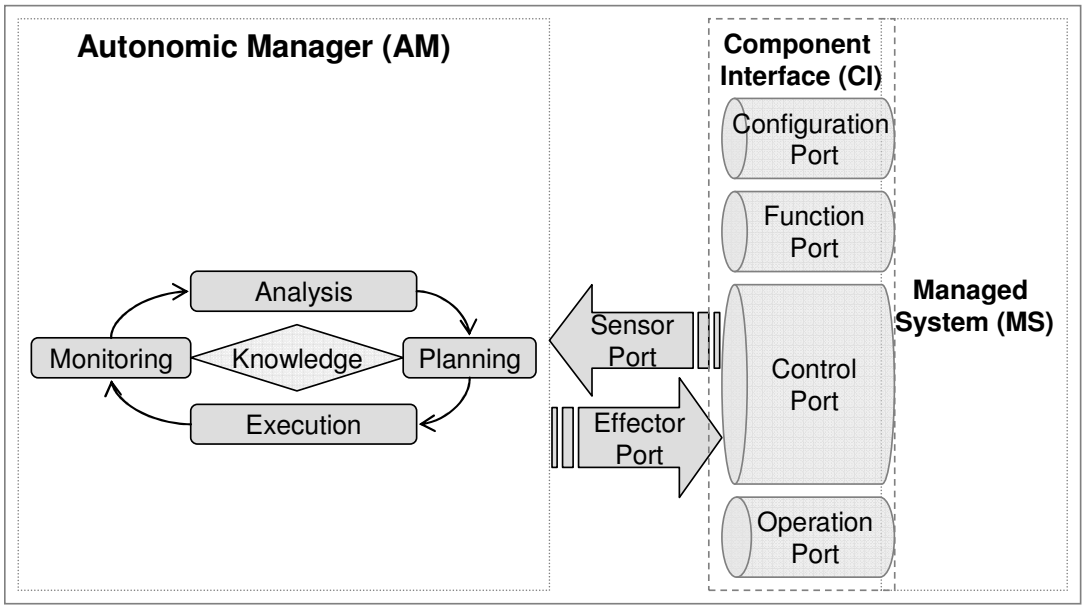

Figure 3-6: Autonomic computing system - the implementation view

The *managed system* exports a set of interfaces called the *component interface (CI)* that defines the procedures to measure the operating states of the *managed system* as well as procedures to change the *managed system* configurations or operations at runtime. In the following sections we describe each element of an *autonomic computing system* in details.

3.5.1 Managed System

The *managed system* is an existing hardware or software resource that is modified to manage itself autonomously. Any *managed system* can be converted into an *autonomic system* by adopting the *component interface* through which the *autonomic manager* manages the system or the software.

3.5.2 Component Interface

The *component interface* consists of four ports referred to as the *configuration port, function port, control port* and *operation port.* The nature and type of information furnished by these ports depend on the self-management objective (self-optimization, self-healing, self-protection or self-configuration). As shown in Figure 3-6, the *control port* consists of two parts – *sensor* and *effecter* ports. The *autonomic manager* uses the *sensor* port of the *component interface* in order to characterize and quantify its current operational state by observing and analyzing the appropriate operational attributes specific to the self-management objective. The *autonomic manager* also uses the *effecter port* to enforce the self-management decision in order to maintain the *managed element* within its operational requirements.

3.5.2.1 Operation Port

This port defines the policies and rules that must be enforced to govern the operations of the *managed system* as well as its interactions with other *managed systems*. Consequently, this port defines two types of rules/policies - *behavior rules* and *interaction rules*. The *behavior rules* define the normal operational region for the *managed system*. *Interaction rules* define the rules that govern the interactions of the *managed system* with other *managed systems*.

3.5.2.2 Function Port

The function port defines the control and management functions provided by the *managed system* that can be used to enforce the management decisions.

3.5.2.3 Configuration Port

This port maintains configuration information related to the *managed system* such as its current configuration, its current activity level etc. This information is used by the *control port sensor* to monitor the state of the *managed system.*

3.5.2.4 Control Port

This is the 'external port' that defines all the information that can be monitored and the control functions that can be invoked on the *managed system*. This *port* uses the policies and rules specified in the other ports (*function, operation* and *configuration* port) to achieve the desired autonomic management behavior for the *autonomic component*. This port consists of the *sensor* port and *effecter* port. The *sensor* port defines the necessary monitoring parameters and measurable attributes that can accurately characterize the state of the *managed system* at any instant of time. The *effecter* port defines all the control actions that can be invoked at runtime to manage the behavior of the *managed element*.

3.5.3 Autonomic Manager

The *autonomic manager* is the augmented intelligence of the *autonomic computing system* to maintain itself under changing circumstances impacted by the internal or external environment. In order to achieve the autonomic management objectives the *autonomic manager* continuously cycles through four phases of operations - *monitoring, analysis, planning* and *execution* as shown in Figure 3-6. In each phase, the *autonomic manager* may use previously stored knowledge to aid this decision making process. The *autonomic manager* controls and manages the *autonomic computing system* using the

component interface ports. Note that different performance parameters and their acceptable values can be set a priori by the user or can be changed dynamically by the *autonomic manager* of the managed resource or software module. Furthermore, we can also dynamically add new methods in the *sensor* and *effecter* ports.

In the following Chapter we design an autonomic computing system that self-manages its power and performance.

3.6 Conclusion

Autonomic computing draws most of its principles from the human autonomic nervous system that has developed successful strategies to handle unprecedented complexity, heterogeneity and uncertainty. Computing research has evolved in many isolated, loosely coupled research fields such as (security, fault tolerance, high performance, AI, network, agent systems, etc.). Each discipline has managed to deliver computing systems and services that meet their target domains (e.g, High Performance Computing, Fault Tolerance Computing). However, if the computing systems and application were to combine these capabilities; i.e., deliver computing systems that provide high performance, security, fault-tolerance, intelligent computing systems and applications are not practically feasible and available. Autonomic computing is the emerging computing field that addresses all these issues in an integrated way and can be viewed as the computing field that will converge all these disciplines into a single field. In this research we demonstrate a small functionality of autonomic computing systems by managing the power and performance of computing systems in a unified framework.

4 OVERVIEW OF RESEARCH APPROACH

4.1 Introduction

In this Chapter, we present our overall research approach to address the problem of data center power and performance management. In the subsequent Chapters we apply this approach towards designing power and performance management efficient data center server platforms.

Statistics of power consumption in data centers show that data centers are always over-provisioned to meet peak loads, suggesting considerable possibilities for power saving. Power and performance management algorithms need to be adaptive to workload changes, flexible, and proactive. The central idea behind this work is to proactively detect and reduce resource over-provisioning in data centers so that the data center is just right-sized to handle the requirements of the application. In this manner we can save power by transitioning the over-provisioned resources to low-power states and simultaneously maintain performance by satisfying the application resource requirements.

We address the data center power and performance management problem by casting it into our general autonomic management framework as discussed in Chapter 3. This transforms a classical data center into an *autonomic component* that is augmented with intelligence to collectively maintain its power and performance under changing workload conditions. The data center uses this augmented intelligence, to proactively and dynamically detect and correct resource over-provisioning for efficient power and performance management.

4.2 Autonomic Power & Performance Managed Data Center

We design an autonomic data center as a special instance of our archetypal *autonomic component* of Chapter 3. Figure 4-1 shows a network of such geographically dispersed autonomic data centers. An autonomic data center consists of the *managed system (MS)* which is the high-performance datacenter and an associated *autonomic manager* (*AM*).

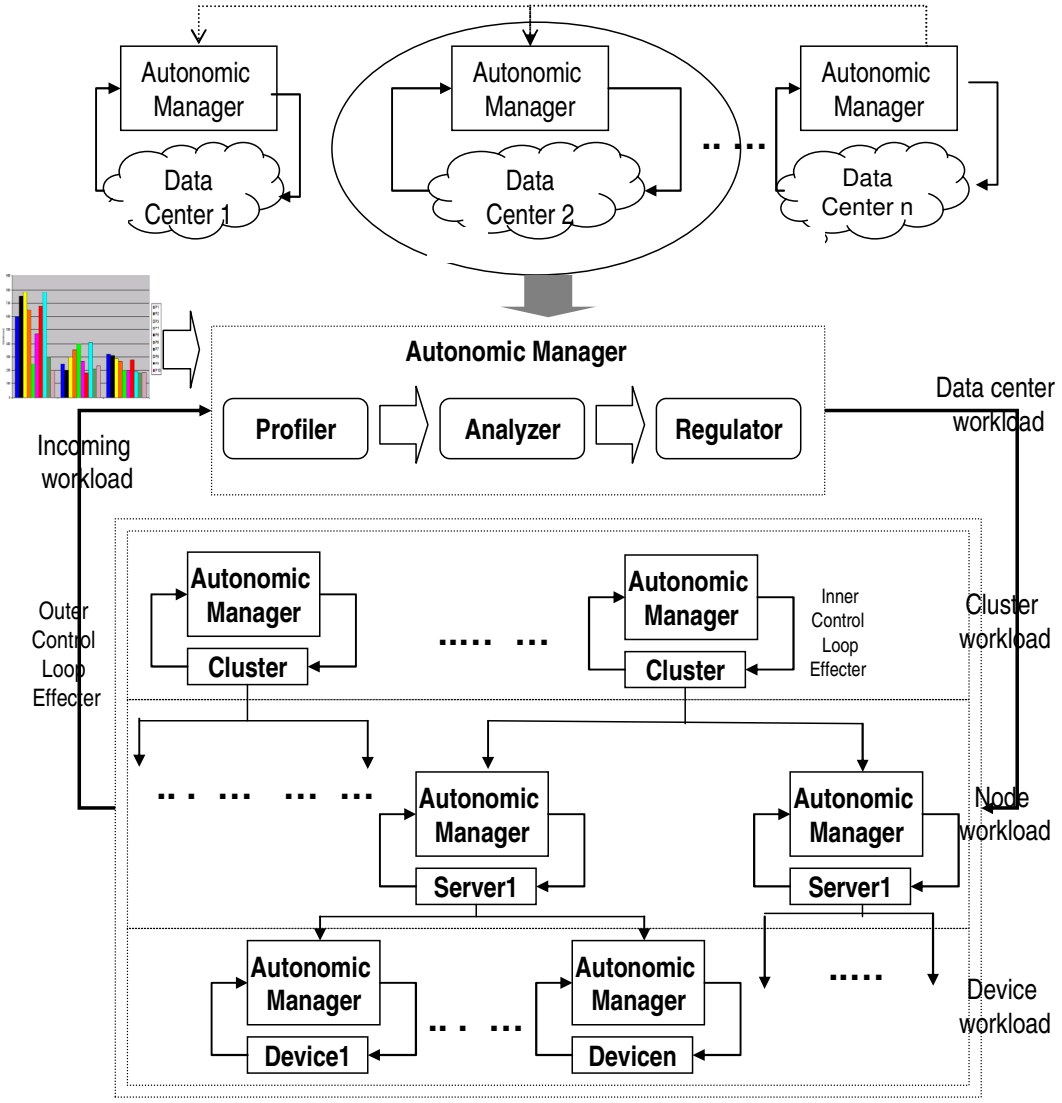

Figure 4-1: Autonomic data center

The data center *managed system* can be logically organized into three distinguishable hierarchies *i)* cluster-level, where the whole data center is modeled as a collection of

networked clusters (front-end, mid-tier and back-end server clusters) *ii)* server-level, where each cluster is modeled as a collection of networked servers and *iii)* device-level, where each server is modeled as a collection of networked devices (processor subsystem, memory subsystem, network cards). Associated with each *managed system* also is a corresponding *autonomic manager* (*AM*). This transforms each *managed system* within the *autonomic data center* into an *autonomic component* based on our archetypal *autonomic component*.

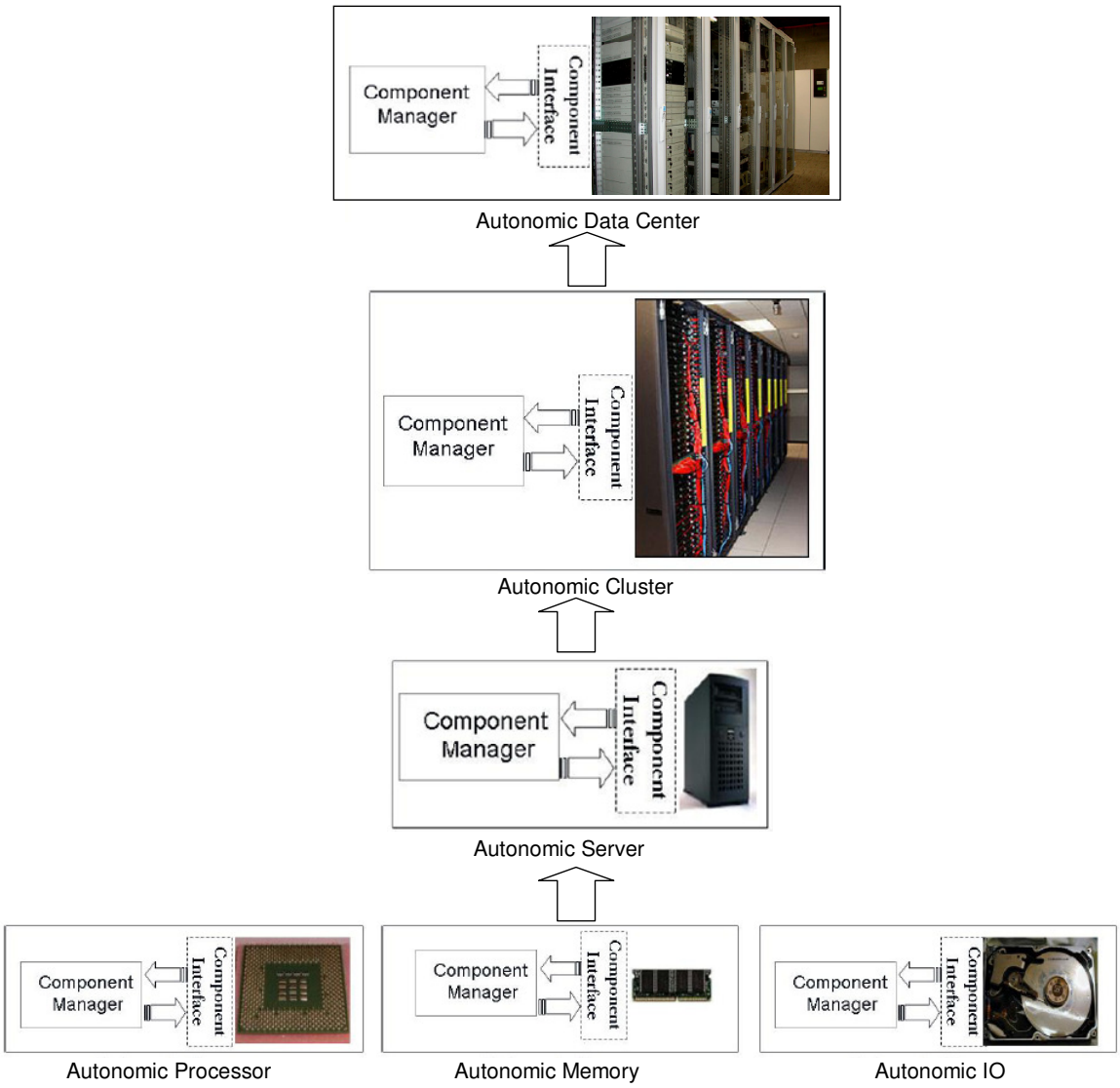

Figure 4-2: Building blocks for an autonomic data center

As shown in Figure 4-2, these *autonomic components* are the building blocks of an *autonomic data center*. For example, an *autonomic memory,* an *autonomic processor* and *autonomic* IO are the building blocks that constitute an *autonomic server* platform, a collection of *autonomic servers* are the building blocks of an *autonomic cluster* and a collection of *autonomic clusters* constitute the building blocks for an *autonomic data center.* Each of these *autonomic components* jointly manages its individual power and performance while collaborating with the upper level hierarchies within the data center to manage the power and performance of the data center as a whole.

4.2.1 Modeling the Managed System (MS)

At any hierarchy, we model the *managed system* as a set of states and transitions. A state denotes a specific capacity of the *managed system* – computing capacity for the processor, storage capacity for the hard disk, maximum connections that can be concurrently supported for a web-server etc. It manifests itself as a specific hardware configuration of the *managed system* corresponding to that capacity. For example let us consider a memory *managed system* within a server platform with 8 GB maximum memory capacity. The memory *managed system* can be configured to 8 GB, 6 GB, 4 GB or 2 GB depending on the memory requirements of the application. Each configuration depicts storage capacity and hence a *state* for the memory *managed system*. Each state is associated with a specific power consumption value and performance values as seen by the application. The smaller the capacity of the *managed system* the lesser the power consumed. However, a smaller capacity may not be sufficient to satisfy the application's needs in which case it would lead to a performance degradation that may be

unacceptable. It is the task of the associated *autonomic manager* to enable the *managed system* to proactively transition to a higher capacity *state* as the application requirements increase and transition back to a lower capacity *state* as the requirements decrease. This ensures that the *managed system* always stays at a *state* where power consumption is minimal without violating any performance constraints. A *state* transition of the *managed system* amounts to a *managed system* reconfiguration.

4.2.2 Modeling the Autonomic Manager

Given the incoming workload, the *autonomic manager* senses the current application resource requirements and analyzes that information to see if the *managed system* is appropriately configured to meet that requirement. If not, the *autonomic manager* intelligently searches for the best *state* (configuration) among all possible *managed system states* that would aptly configure the *managed system* such that it gives the maximum power savings without hurting application performance. At any hierarchy we formulate this as a *performance-per-watt* optimization problem subject to associated performance constraints.

4.2.2.1 Policy optimization for power and performance management

The *autonomic manager* solves the optimization problem to search for the *managed system state* that is optimal among all possible *managed system states* in terms of delivering the minimum power consumption while meeting all the performance constraints. The solutions at the upper levels of the *managed system* hierarchy are more coarse-grained taking into consideration the global view of the system such as a whole

data center or a whole cluster. As we go down the hierarchy, solutions become more fine-grained by taking into consideration a narrow local view of the system such as a server within a cluster or an individual device within a server. This is a manifestation of the local and global control loops as discussed in Chapter 3.

4.2.2.2 *Appflow*: Adding clairvoyance to an *Autonomic Manager*

As discussed earlier, the main objective of an *autonomic manager* is to analyze and predict the dynamic behavior and resource requirements of the workload in a manner that can be intelligently utilized to maximize the *performance-per-watt* of its *managed systems.* To be able to do that effectively, the *autonomic manager* has to be aware of its environment, changes in its environment and it should have the judgment to determine whether the perceived change is acceptable for "survival". If not, it should have the capability to "evolve" itself to adapt to the new environment for "survivability". To conceptualize this notion of awareness of one's environment and the capability to predict and detect changes as and when they occur, we tap onto the management-rich information floating within the datacenter environment itself. With accurate and efficient monitoring of this information defined as a set of monitoring "features", it is possible for an *autonomic manager* to predict trends and patterns in changes and configure its *managed system(s)* to meet these changes ahead of time.

This choice of "features" is an inherently challenging task in itself. We approach this problem by searching for a logical connection that seamlessly weaves together the underling heterogeneous and dynamic datacenter infrastructure in a more insightful fashion than what is immediately apparent to the eye. A little thought reveals this logical

connection in the interactions of the application(s)/workload with the underlying infrastructure. This logical connection establishes causal relationships between individual resource units in the datacenter, temporal variations in their utilization etc, analyzing which may provide valuable information for datacenter manageability. Borrowing from network research [NetFlow2007] we introduce the concept of an *application execution flow* (*Appflow*) that captures the effects of the interactions of the incoming workload on the underlying infrastructure.

Appflow is a three-dimensional array of *features* where the x-dimension captures spatial variability and the z-dimension captures temporal variability for each *managed system* at a specific datacenter hierarchy plotted along the y-dimension. This is shown in Figure 4-3.

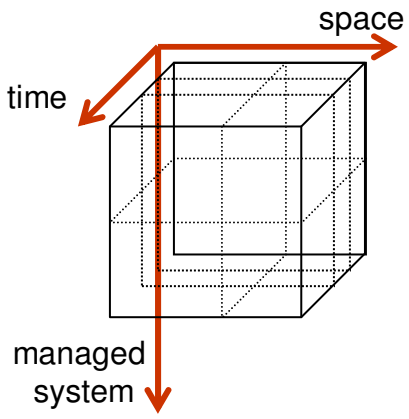

Figure 4-3: *Appflow* structure

For a given *managed system* on the y-axis, the "space" vector on the x-axis captures its behavior as datacenter workloads change dynamically by several orders of magnitude within a day or a week. This is captured by using a set of necessary and sufficient dynamically monitored *features* for the *managed system*. They are categorized into two distinct classes – Capacity Spatial Features (CSF) and Operating Region Spatial Features

(ORSF). Capacity Spatial Features (CSF) can be further broken down into Static Capacity Spatial Features (SCSF) and Dynamic Capacity Spatial Features (DCSF). As the names suggest, SCSF indicates the maximum capacity of the *managed system* and DCSF indicates the dynamic capacity of the *managed system* that is provisioned based on the requirements of the application at runtime. The *ORSF* is a special set of *features* that depend on the *CSF*. They are special because they are in essence the "essential variables" of the *autonomic component*. The *autonomic manager* manages these *ORSF* within a safe operating zone, in order to manage the power and performance of the *managed system*. The *ORSF* encapsulate the runtime behavior of the *managed system* at any instant of time in response to the incoming workload. They establish a managed system *operating point*, in an n-dimensional space, where n is the total number of *ORSF*. During the lifetime of the application, the position of the *operating point* changes in space. This is an indication of how the *managed system* is responding to the incoming traffic. The *autonomic manager* uses this information to determine when a *managed system* reconfiguration may be required to maintain the power and performance.

Within a data center hierarchy each *autonomic component* utilizes its own *Appflow* to develop this clairvoyance for power and performance management of its *managed system.* This is captured in Figure 4-4, where we have *device Appflow* being used by *device autonomic manager*, *server Appflow* being used by *server autonomic manager*, *cluster Appflow* being used by *cluster autonomic manager* and finally *data center Appflow* being used by *data center autonomic manager*.

Note that the *Appflow* gives a coherent and consistent flow of information throughout the data center hierarchy. For example, a *response time ORSF* in the *Appflow* retains the

same interpretation for an *autonomic manager* as we traverse the hierarchy. There is no loss of information as the *autonomic components* in the different hierarchies co-operate and coordinate to work towards the common objective of data center power and performance management.

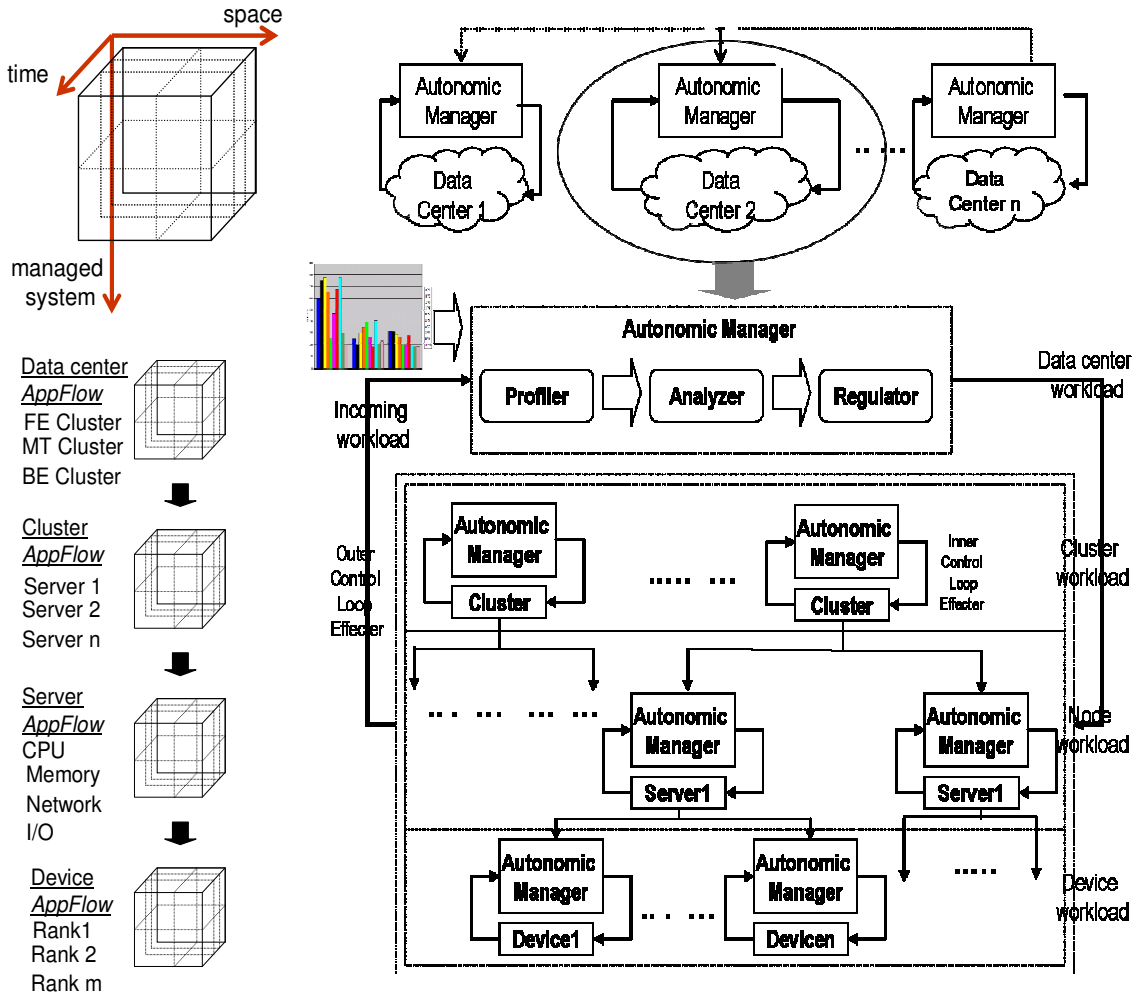

Figure 4-4: *Appflow* for an autonomic data center

4.2.2.3 Data Center *Autonomic Manager*

As shown in Figure 4-1, the top-level *data center autonomic manager* handles the incoming data center workload. Its *managed systems* consist of the Front-End (FE)

Cluster, Mid-Tier (MT) Cluster and the Back-End (BE) Cluster. It uses the *data center Appflow* to profile and characterize the incoming workload in a manner that can be intelligently utilized by the lower-level *autonomic managers* to maximize the *performance-per-watt* of their *managed systems*.

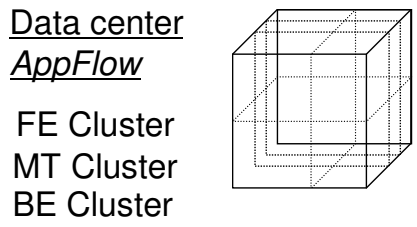

Figure 4-5: Data center *Appflow*

4.2.2.4 Cluster *Autonomic Manager*

The *cluster autonomic manager* uses the *data center Appflow* to make initial configuration decisions for the *managed cluster.* Its *managed system* consists of servers within the cluster. It solves a *performance-per-watt* optimization problem to search for the best configuration (*state*) for the cluster such that it minimizes the cluster power consumption and while maintaining performance. It uses the *cluster Appflow* to determine when to trigger a *state* reconfiguration whenever the *cluster operating point* is predicted to go out the predetermined safe operating zone.

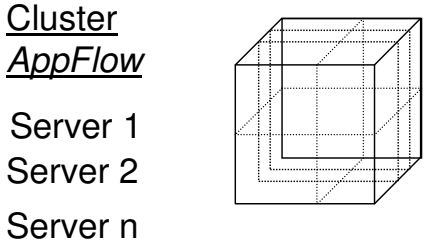

Figure 4-6: Cluster *Appflow*

4.2.2.5 Server *Autonomic Manager*

The *server autonomic manager* functions exactly as the *cluster autonomic manager* but with narrower visibility. Its *managed system* consists of devices within the server – CPU, memory, network, I/O. It refines the global decision received from the *cluster autonomic manager*. It uses its *server Appflow* to trigger the search for the best server configuration (*state*) such that it minimizes the server power consumption and while maintaining performance.

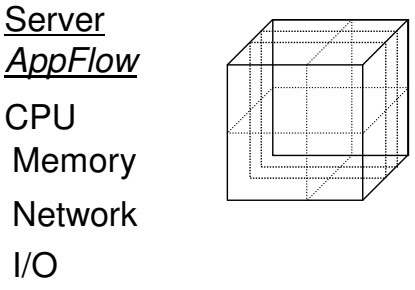

Figure 4-7: Server *Appflow*

4.2.2.6 Device *Autonomic Manager*

The *device autonomic manager* is responsible for power and performance management at the lowest level in the hierarchy such as the processor, memory, network, disk etc within a server. Its *managed system* consists of devices sub-units such as processor cores, memory ranks etc. It functions similar to the upper-level *autonomic managers*. It uses its *device Appflow* to trigger the search for the best device configuration (*state*) such that it minimizes the server power consumption and while maintaining performance.

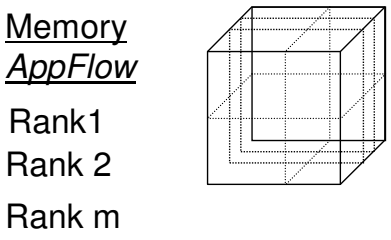

Figure 4-8: Memory *Appflow*

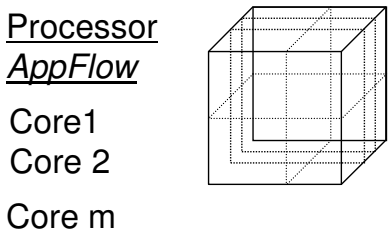

Figure 4-9: Processor *Appflow*

In this work, we validate our approach bottom-up. We first look at autonomic power and performance management for the memory subsystem within a high-performance data center server platform. This is discussed in Chapter 5. We then build on this work to encompass power and performance management of the whole server platform that consists of multi-core processors and multi-rank memory subsystem. This is discussed in Chapter 6.

4.3 Conclusion

In this Chapter, we presented our overall research approach for hierarchical autonomic power and performance management of datacenters. We presented the design of an autonomic datacenter built using autonomic clusters, servers and devices within a server platform. Each autonomic component in the data center is designed based on the archetypal autonomic component of Chapter 3. In Chapter 5 we will focus on power and performance management of the memory subsystem within a server platform and then focus on power and performance management of the whole server platform in Chapter 6.

5 ADAPTIVE INTERLEAVING TECHNIQUE FOR MEMORY POWER & PERFORMANCE MANAGEMENT

5.1 Introduction

In this Chapter we focus on power and performance management at the lowest wrung of the data center hierarchy. We discuss how we apply the autonomic computing paradigm to design an autonomic memory subsystem within a high-performance data center server that jointly maintains its power and performance under varying workload conditions.

The energy consumed by servers' memory subsystem constitutes a major part of the total data center power consumption [Lebeck et. al. 2000]. However given that workloads' dynamic memory requirements display a wide range of variation it is possible to use multi-power *state* memory technologies such as Rambus DRAM (RDRAM) [Rambus1999] and Fully-Buffered DIMM (FBDIMM) [FBDIMM2006] to save energy and maintain performance by allocating just the required amount of memory to applications at runtime and transitioning any additional memory capacity to low-power *state*s.

However, given that server platforms in a data center are often configured at peak performance, the memory subsystem is most often configured at the maximum degree of interleaving. This introduces a challenge for the memory power management problem. Owing to symmetrical distribution of memory accesses across all memory modules interleaving does not offer much opportunity for energy saving and thus provides less opportunity for idleness of the memory modules. For example, an experimental study to measure the idleness of memory modules when executing SPECjbb2005 benchmark on a

server platform (2 Dual-Core Intel™ Xeon processors, 5000P Memory Controller Hub, 8 GB DDR2 FBDIMM memory) with fully-interleaved (16-way) memory showed that memory modules were idle for less than 5% of the total application runtime. Applying existing power management techniques [Lebeck et. al. 2000], [Fan et. al. 2001], [Zhou et. al. 2004] to this memory subsystem would yield only ~ 4.5% total saving. However, conducting the same experiment with a smaller degree of interleaving (12-way) created an imbalance in idleness making some modules more idle and others busier. By power managing the idle memory modules we gained an energy saving of 25% (14.7 kJ) with negligible impact on SPECjbb2005 performance. This demonstrates an opportunity to maximize the memory power and performance by dynamically scaling down the degree of interleaving (16-way to 12-way) to adapt to the application's memory requirements at runtime. With this objective in mind, we design an autonomic memory subsystem that addresses the following research challenges related to memory power and performance management.

1. How do we exploit an application's memory reference behavior to guide our choice of an appropriate degree of interleaving for the memory subsystem? This depends on how physical allocation of application's *working set pages* on memory modules impacts the power consumed by the memory subsystem and the application-level performance.
2. How do we dynamically predict the impact of a specific degree of interleaving on the *performance-per-watt* metric for the platform?
3. How do we design smart interleaving techniques that effectively exploit the platform's memory hierarchy architecture to maximize its *performance-per-watt?*

4. What enabling techniques and hardware design changes are required to implement this paradigm shift from static (boot-time) interleaving to dynamically reconfigurable memory interleaving?
5. How can we design a dynamic interleaving technique that leverages existing fine-grained power management techniques to maximize *performance-per-watt* for platforms with interleaved memory?
6. What are the cost, run-time complexity and reconfiguration overhead associated with our technique and how they can be reduced to attain a greater return-on-investment?
7. What is the ideal *mean-time-between-reconfigurations (MTBR)* for our technique such that it maintains adaptivity to incoming workload without significantly increasing the overhead of reconfiguration?

5.2 Motivational Example

In this Section we explain the problem of power and performance management of interleaved memory subsystems in detail with the aid of the following example.

5.2.1 Power Management for Interleaved Memory

Let us consider a memory subsystem with 8 memory modules (A1, A2, A3, A4 and B1, B2, B3, B4) as shown in Figure 5-1 where each module is individually power-managed. Memory modules in 'block A' can be accessed in parallel to those in 'block B' (similar to our experimental server unit). Let us consider that the memory is fully-interleaved, which is the general configuration for server memory. With full-interleaving

(8-way) cache lines belonging to a single *page* are striped across all memory modules. For the memory subsystem of Figure 5-1, cache line 1 is allocated to module A1,

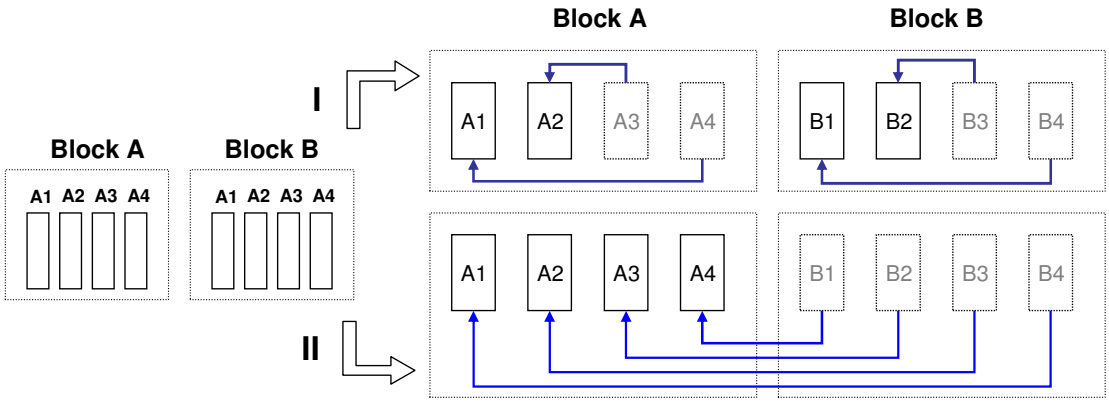

Figure 5-1: Data migration strategies

line 2 to module B1, line 3 to module A2, line 4 to module B2 etc. Hence adjacent cache lines are allocated to separate blocks. A cache line pair is allocated to the same memory module if there are a multiple-of-8 cache lines between them. Now, let us consider two time instants t_i and t_{i+1} during the application execution such that the application requires n_i *pages* at t_i and n_{i+1} *pages* at t_{i+1} to achieve its maximum *hit ratio*. Let us consider a sequential *page* allocation scheme [Lebeck et. al. 2000] where *pages* are allocated to one memory module completely filling it up before going to the next. Hence, if $n_{i+1} < n_i$, we can save power by transitioning the modules that contain the unused (n_i - n_{i+1}) *pages* into a low-power *state*. However because of full-interleaving the n_{i+1} *pages* would occupy all the modules leaving no memory modules to transition to a low-power *state*. We propose to create the opportunity for power saving in fully-interleaved memory by dynamically varying the degree of interleaving with minimal performance impact. For the example shown in Figure 5-1, we reduce the degree of interleaving from 8-way to 4-

way by migrating the data from 8 modules to 4 modules. In this manner we can transition the remaining 4 modules to a low-power *state*.

5.2.2 Exploiting Platform's Memory Architecture

Reducing the degree of interleaving decreases the parallelization in memory accesses which in turn may impact *memory access delay*. One way of reducing the impact on *delay* is to migrate the data in a manner that exploits any unique characteristics of the underlying memory architecture. For example in Figure 5-1 memory modules in 'block A' can be accessed in parallel to those in 'block B'. We want to devise migration strategies that effectively exploit this characteristic. For example, Figure 5-1 shows two different migration strategies. In strategy I, data is migrated from memory modules A4 and A3 to A1 and A2 respectively within block A and from B4 and B3 to B1 and B2 respectively within block B. In strategy II on the other hand, data is migrated from memory modules B4, B3, B2, B1 to A1, A2, A3, A4 respectively all in the same block A. Given that the blocks can be accessed in parallel, strategy II would have a higher impact on *delay* compared to strategy I because it did not exercise both blocks. Since cache lines with very high spatial reference affinity would lie within the same block, they would experience an increase in *delay* owing to sequentialization of accesses as compared to strategy I where they can be accessed in parallel. Since most programs demonstrate a high spatial locality of reference, this would lead to a significant reduction in the parallelism of accesses for strategy II and thereby severely impacting *delay*. We have experimentally verified this observation where we noticed a 5.72% drop in SPECjbb2005 performance for migration strategy II.

5.3 Autonomic Memory Subsystem

Figure 5-2 Figure 5-2 shows an *autonomic* memory subsystem based on our archetypal autonomic component from Chapter 3. It implements a hierarchical management architecture. The top-level *autonomic manager*, which we call the *Data Migration Manager*, manages the entire memory subsystem. This is denoted by the global control loop *G*. The bottom-level *autonomic manager,* which we call the *Power Manager*, manages an individual memory module within the entire memory subsystem. This is denoted by the local control loop *L*. Essentially, an individual memory module is the smallest physical memory unit that can be independently power managed. In what follows, we describe each of the components in detail and explain how they work together to maintain the power and performance for the memory subsystem.

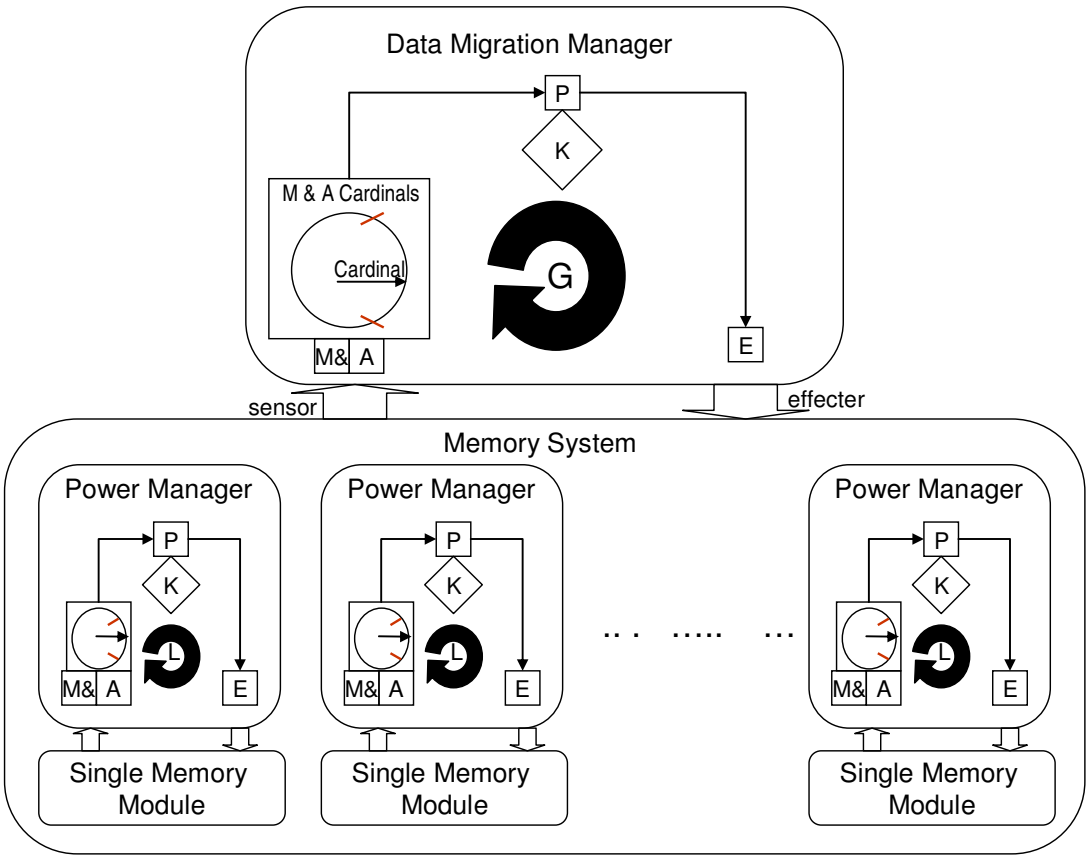

Figure 5-2: Autonomic memory subsystem

5.3.1 Modeling the Memory *Managed System*

In this Section we discuss the power and performance model for the memory subsystem. The memory subsystem is an aggregation of several individual memory modules that can be independently power and performance managed. In our hierarchical management architecture, the top-level *managed system* is the whole memory subsystem and it is power and performance managed by the *autonomic manager* we call the *Data Migration Manager* and the bottom-level *managed system* is a collection of individual memory modules and each is power and performance managed by a corresponding *Power Manager.*

In this work we consider an FBDIMM memory subsystem as our *managed system* which is popular in high-performance servers because of its reliability, speed and density features. As shown in Figure 5-3, an FBDIMM packages multiple DDR DRAM devices and an Active Memory Buffer (AMB) in a single module. The AMB is responsible for buffering and serially transferring data between DRAM devices on the FBDIMM and the Memory Controller (memory controller). Figure 5-5 shows the model of our memory subsystem. It is based on the memory subsystem architecture found in Intel™ S5000PAL/S5000XAL server boards with 5000 series chipsets. The memory subsystem consists of multiple branch*es;* each consists of multiple channel*s*. Each channel in turn contains multiple FBDIMMs and each FBDIMM contains multiple rank*s*. *Ranks* on separate branch*es* are accessed in parallel. However, rank*s* on separate channel*s* within the same branch are accessed in lock-step. *Ranks* within the same channel are accessed sequentially. The number of DRAM devices accessed simultaneously to service a single memory request defines a memory rank [Huang et. al. 2005]. In our memory subsystem,

we consider a rank as the smallest unit for memory power management. This is depicted as a single memory module in Figure 5-2 and defines the *managed system* at the lower-level of the hierarchy that is power and performance managed by the *Power Manager*.

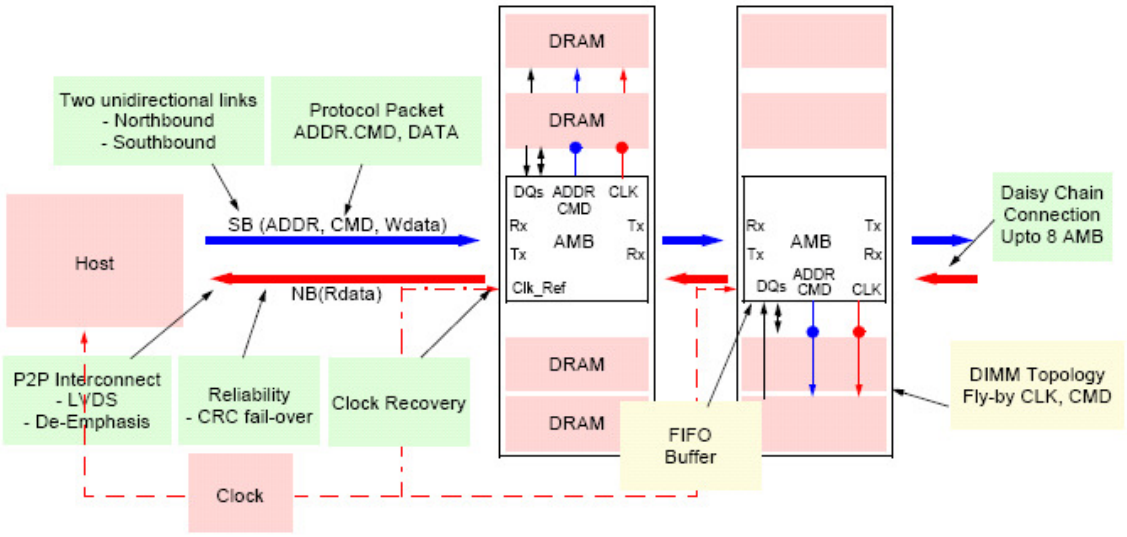

Figure 5-3: FBDIMM Memory Model

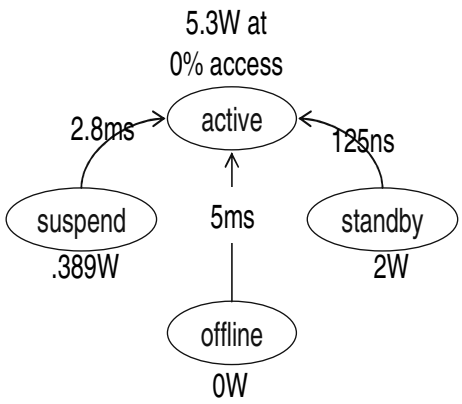

Figure 5-4: Power *state*s and transitions

An FBDIMM supports four power *state*s – *active* (channel on, DRAM on, AMB on), *standby* (DRAM off but power is not removed, channel on, AMB on), *suspend* (DRAM self-refresh, channel Off, AMB core-off) and *offline* (DRAM off, channel Off, AMB off)

as shown in Figure 5-4. It can service memory requests only when *active*. A power *state* that consumes the least power has the highest reactivation time.

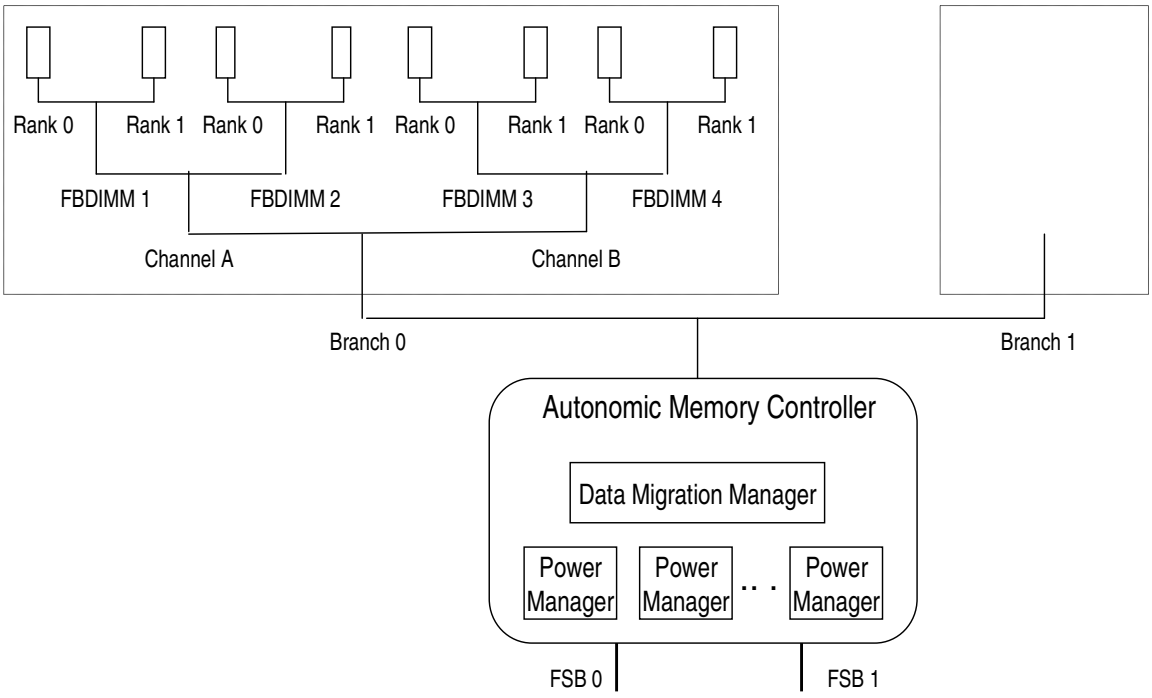

Figure 5-5: Memory subsystem model

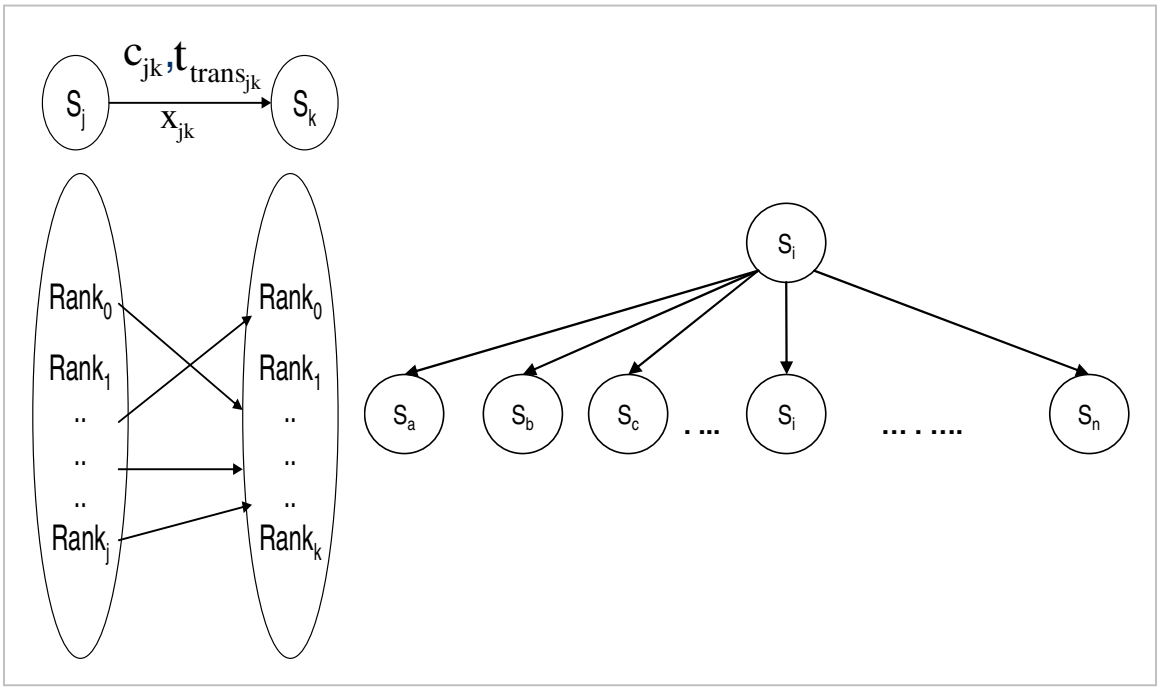

Figure 5-6: Memory subsystem *state* space

The AMB brings in another challenge for memory power management as it consumes power (even when the FBDIMM is idle) to maintain the data transfer link between the memory controller and neighboring FBDIMMs on the channel (Figure 5-3).

We model the performance of the memory subsystem in terms of end-to-end *delay* d. It is defined as the time from the instant a request arrives at the memory controller to the time when the data becomes available at the memory controller and consists of two delays – the *processing delay* τ_p and the *queuing delay* τ_q. τ_p consists of the transmission delay over the channel and the rank access time delay. τ_q is the time spent by a request in a memory controller buffer queue before it is sent to a memory rank for processing. This *delay* can be due to contention either in the channel or the channel and the rank.

The memory subsystem's *state* space is shown in Figure 5-6. Each *state* is defined by the number and the physical location of the rank*s* in the '*active'* power *state*. The remaining rank*s* can be in any of the low-power *state*s – *suspend, standby and offline*. A transition takes the system from one *state* to another. Each *state* s_k is defined by a fixed base power consumption p_k and a variable end-to-end *delay* d_k. The *state* is defined as follows.

$$s_k = \{r_i \mid i : 1 \rightarrow n_k, r_i \in N\}$$

$$p_k = n_k * p_a$$

$$d_k = \tau_{p_k} + \tau_{q_k}$$

where, N : Total rank*s* in the system

n_k : Total *'active'* rank*s* in *state* s_k

p_a : Base power consumption in the *active* power *state* (5.3 W from Figure 5-4)

p_k : Total power consumed by the system in *state* s_k

d_k : End-to-end memory access *delay* in *state* s_k

τ_{q_k} : Queuing delay in *state* s_k

τ_{p_k} : Processing delay in *state* s_k

5.3.2 Modeling the *Data Migration Manager*: Adaptive Interleaving for Memory Power & Performance Management

In this Section we describe how the *Data Migration Manager* works with the individual *Power Managers* to collectively manage the power and performance of the memory subsystem. Every observation epoch, the *Data Migration Manager* predicts the dynamic memory requirements of the application. It then identifies an optimal memory configuration/*state* (number and physical location of rank*s*) that meets the application memory requirements and at the same time maintains the end-to-end memory access *delay* for the given workload. It uses this memory configuration to determine the degree of interleaving that would maximize the *performance-per-watt* for the given workload.

The *Data Migration Manager* uses the memory *Appflow*, to determine when such a *state* transition/reconfiguration is required for the memory subsystem in order to maintain its power and performance. It determines the target *state* to transition to by solving a *performance-per-watt* optimization problem. It readjusts the degree of interleaving by dynamically migrating cache lines on to selected memory rank*s* transitioning the remaining rank*s* to appropriate low-power *state*s. It then updates a hardware indirection data structure to handle address mapping of migrated cache lines.

In what follows we first discuss how the *Data Migration Manager* predicts the application memory requirements and then discuss our formulation of the *performance-per-watt* optimization problem. We then construct the memory *Appflow* and finally bring it all together in the form of our memory *performance-per-watt* management algorithm called **PERFWATT**. The dynamic *Data Migration Manager* implements the **PERFWATT** algorithm for adaptive interleaving. The individual *Power Managers* are responsible for transitioning memory rank*s* to the appropriate low-power *state* based on the length of the idle duration. The *Data Migration Manager* can be coupled with any suitable fine-grain power management techniques at the lower-level implemented in the *Power Manager*.

5.3.2.1 Dynamic Tracking of Application's Memory Requirement

The *Data Migration Manager* uses the **memAlloc** algorithm to determine at runtime the appropriate memory size (N_{ws} *pages*) that can reduce memory misses and over-provisioning simultaneously. It uses the MRC metric [Zhou et. al. 2004] to predict the dynamic memory requirements of the application. Let us consider that during epoch t_i, memory of size *n* (*pages*) is in an *active* power *state*. Let us assume that we measured the number of hits going to each memory *page* and that the *pages* are maintained in a strict LRU order [Bovet et. al. 2002]. Now, the contents of a memory of size *n pages* are a subset of the contents of a memory of size *n + 1 pages* or larger due to the *inclusion property* of the LRU algorithm. Using this property we can calculate the MRC (line 3) for any memory size of *m pages* where *m* <= *n* during epoch t_i as follows

$$MRC(m)_{t_i} = 1 - \frac{\sum_{i=1}^{m} p_{hits[i]_{t_i}}}{\sum_{i=1}^{n} p_{hits[i]_{t_i}} + p_{miss_{t_i}}} \quad \ldots (1)$$

where $\sum_{i=1}^{n} p_{hits[i]_{t_i}}$ and $\sum_{i=1}^{m} p_{hits[i]_{t_i}}$ are memory hits to the *n* and *m pages* respectively and $p_{miss_{t_i}}$ is the measured *page misses* during t_i. The numerator in Equation (1) is the number of memory misses for a memory of size *m (pages)*, where *m*<*n*.

memAlloc: Getting ideal memory size for minimum memory misses

1. $N_{ws} = -1;\ MRC_{\min} = MRC(N_{ws}) = \infty;\ p_{buf} = \beta$
2. **For** $(m = 0; m \le n; m++)$
3. $MR(m) = 1 - \frac{\sum_{i=1}^{m} p_{hits[i]}}{\sum_{i=1}^{n} p_{hits[i]} + p_{miss}}$
4. **If** ($MRC(m) < MRC_{min}$)
5. $MRC_{min} = MRC(m)$
6. $N_{ws} = m$
7. **Endif**
8. **Else if** ($MRC(m) = MRC_{min}$)
9. **Break**
10. **Endelse**
11. **Endfor**
12. **If** $(N_{ws} == n)$
13. $N_{ws} += \beta$
14. **Endif**
15. **Return** N_{ws}

As we increase the memory size, the corresponding *MRC* reduces (lines 4-7). However, it would stay the same for memory of size *m* (*pages*) or m+1 *(pages)* if the misses stay the same for both cases (line 8). This would mean that the (m+1)th *page* was over-

provisioned and the ideal memory size is *m (pages)*. If however the *MRC* keeps reducing until we have covered all the *n pages* we increase the size of the memory by *bufPages* in anticipation that it would bring down the *pages misses* further. *bufPages* is determined at the end of each epoch t_i based on the measured *pages misses* $p_{miss_{t_i}}$, the higher the *page misses* the higher the *bufPages*

In fully-interleaved memory, N_{ws} *pages* are striped across all memory rank*s* mainly to improve performance-efficiency. In non-interleaved memory, N_{ws} *pages* are consolidated on one memory rank completely filling it up before going to the next rank which improves power-efficiency. In the following Section we discuss how at runtime the *Data Migration Manager* determines a degree of interleaving that stripes the N_{ws} *pages* on memory rank*s* to improve both power and performance efficiency.

5.3.2.2 Interleaving of Application *Working Set* Pages for Power and Performance Management

In this Section, we first discuss data placement in a fully-interleaved memory. We then discuss how the *Data Migration Manager* adjusts the degree of interleaving to adapt to incoming workload by using a temporal affinity prediction technique that effectively exploits the internal memory architecture.

I. Data Placement in Fully-interleaved Memory

Let us revisit the memory architecture shown in Figure 5-5 that interleaves data at the cache line granularity. Cache lines are allocated to memory rank*s* such that spatially

adjacent cache lines reside on memory rank*s* that are furthest away from one another. This increases the parallelization in memory accesses. For example, in Figure 5-5 cache line 0 is allocated to rank 0 on branch 0 while cache line 1 is allocated to rank 0 on branch 1 etc.

Since the temporal affinity in accesses is determined to a large extent by placement of cache lines on the underlying architecture that consists of branch*es*, channel*s* etc we abstract the notion of distance between cache line pairs based on their relative placement in the memory subsystem. We express this in the form of Spatial Reference Affinity (*SRA*) and Spatial Location Affinity (*SLA*) metrics and use them to analyze the impact of a memory configuration on the access *delay*.

II. Predicting Temporal Affinity

We predict the temporal affinity between memory accesses with the aid of the *SRA* and *SLA* metrics. Let us consider cache lines that belong to a single *page*. Adjacent cache line pairs have a higher probability of temporal affinity in accesses compared to those that are further apart in the same *page*. We use the ***SRA Metric*** to capture the temporal affinity arising out of this closeness in space. The *SRA* between a cache line pair *i* and *j* is measured by the number of cache lines that separate them. Hence *SRA* is zero for adjacent cache line pair and $(\frac{s_P}{s_{CL}} - 1)$ for the first and last cache lines on the *page* where s_P denotes the *page* size and s_{CL} denotes the cache line size.

Temporal affinity in accesses may have very different impact on the *delay* depending on the relative position of the cache line pairs within the memory hierarchy of Figure 5-5. For example, adjacent cache line pairs (*SRA* = 0) with high temporal affinity may have

negligible impact on the *delay* if they are placed on separate memory branch*es*. The reason is because even when they are accessed immediately after one another each access is serviced by a separate branch which can be accessed in parallel. We use the ***SLA Metric*** to capture the temporal affinity that arises out of this closeness in physical location of cache line pairs. Unlike *SRA*, there is no simple way of computing the *SLA* metric. So we manually configured the memory architecture on a server to multiple different configurations (e.g., different branch*es*, same branch different channel*s* etc). We then ran SPECjbb2005 and recorded its performance (throughput) on each of these configurations. We repeated the experiment with the memory simulator application DRAMsim [Wang et. al. 2005] and measured the execution time for the application on each configuration. We also repeated this experiment by running ***gcc*** traces on different memory configurations (16 rank*s* total) configured in the DRAMsim simulator - such as (4 branch*es*, 1 channel per branch and 4 rank*s* per channel), (2 branch*es*, 2 channel*s* per branch and 4 rank*s* per channel), (1 branch, 2 channel*s* per branch and 8 rank*s* per channel) and so on. From these results we derived the following empirical function to compute SLA

$$SLA_{[B_i,D_i,R_i][B_j,D_j,R_j]} = (w_B * | B_i - B_j | + w_D * | D_i - D_j | + w_R | R_i - R_j | + w_z) \quad \ldots (2)$$

where w_B : weight of cache line pair across branch*es*, w_D : weight across FBDIMMs, w_R : weight across rank*s* and w_Z : weight on the same rank. These are empirically derived weights where $w_B > w_D > w_R > w_Z$, $w_B > (w_D + w_R)$ etc. Hence, a cache line pair across separate branch*es* B_i and B_j gives a higher *SLA* compared to a cache line pair across separate rank*s* ($w_B > w_R$). The impact of the *SLA* on the *delay* itself is discussed in the following paragraph.

Clearly, there exist two dimensions of variability when we study the impact of placement of cache lines on the *delay* – their *SRA* and *SLA*. We now combine these two metrics in order to weigh the impact of one placement strategy over another in terms of their combined impact on the *delay* using the ***conflict metric*** $\Delta\psi$.

$$\Delta\psi(s) = \frac{1}{SLA_{\min}(s) * SRA_{\min}(s)} \qquad \text{... (3)}$$

Let us consider two placement strategies that use different memory sizes. We compute the minimum *SRA* given the minimum *SLA* for each strategy. From equation (3) the strategy that has the smallest *SLA* and smallest *SRA* for that *SLA* has a higher *conflict metric* indicating a higher impact on the *delay*. Similarly, for two placement strategies that use the same memory size (number of rank*s*) but different memory configuration (physical location of rank*s*) we compute minimum *SLA* for adjacent cache line pair (minimum *SRA*) for each strategy. Since the minimum *SRA* is fixed for both strategies, in this case the strategy that gives the smaller *SLA* has a higher *conflict metric* indicating a higher impact on the *delay*. For example, the migration strategies of Figure 5-1 have the same memory size but different memory configuration. For strategy II, the *SLA* for minimum *SRA* is w_D and for strategy I the value is w_B. Since $w_B > w_D$ $\Delta\psi$ is higher for strategy II. This explains the drop in performance for SPECjbb2005 for strategy II. Hence, a migration strategy with a smaller $\Delta\psi(s)$ gives better performance.

5.3.2.3 Formulating the Optimization Problem for *Performance-per-Watt* Management

We formulate our adaptive interleaving technique as a *performance-per-watt* optimization (maximization) problem with respect to the memory subsystem *state* space discussed earlier.

$$\text{Maximize } ppw_{t_i} = \frac{1}{d_k * e_k * x_{jk}} \text{such that}$$
$$1. n_k * s_r >= N_{ws} * s_p$$
$$2. d_{\min} \le d_k \le d_{\max} \qquad \text{... (4)}$$
$$3. \sum_{k:1}^{N_s} x_{jk} = 1$$
$$4. \vee x_{jk} = 0 | 1$$

where, ppw_{t_i} is the *performance-per-watt* during interval t_i, d_k is the *delay* and e_k is the energy consumed in target *state* s_k where, e_k is given by $e_k = \sum_{k:1}^{N_s} (c_{jk} * \tau_{trans_{jk}} + p_a * n_k * t_{obs}) * x_{jk}$ is the sum of the energy consumed during *state* transition and ($c_{jk} * \tau_{trans_{jk}}$) is the energy consumed in the target *state* ($p_a * n_k * t_{obs}$), N_s: total number of system *states*, c_{jk}: power consumed in *state* transition and $\tau_{trans_{jk}}$: time taken for *state* transition, $[d_{\min}, d_{\max}]$: the threshold delay range, x_{jk}: decision variable for transition from *state* s_j to s_k, s_r: size per rank, s_p: size per *page*

Constraint 1 of the optimization equation (4) states that the target *state* should have enough memory to hold all the N_{ws} *pages*. Constraint 2 states that in the target *state*, the *delay* should stay within the threshold range. Constraint 3 states that the optimization

problem leads to only one decision. The decision variable corresponding to that is 1 and the rest are 0. Constraint 4 states that the decision variable is a 0-1 integer.

Analysis of Transition Overhead: The transition overhead $c * \tau_{trans}$ is the energy spent during *state* transition. We factor this overhead into the objective function to identify *state* transitions that would give the smallest overhead among all possible transitions. We also include the transition time τ_{trans} in calculating the *delay*. From constraint 2 of the optimization equation this prevents *state* transitions when τ_{trans} is too high. Hence this reduces the frequency of *state* transitions and thereby maintains the algorithm sensitivity to workload changes, within acceptable bounds.

Before we analyze the migration overhead let us first briefly discuss the migration mechanism itself.

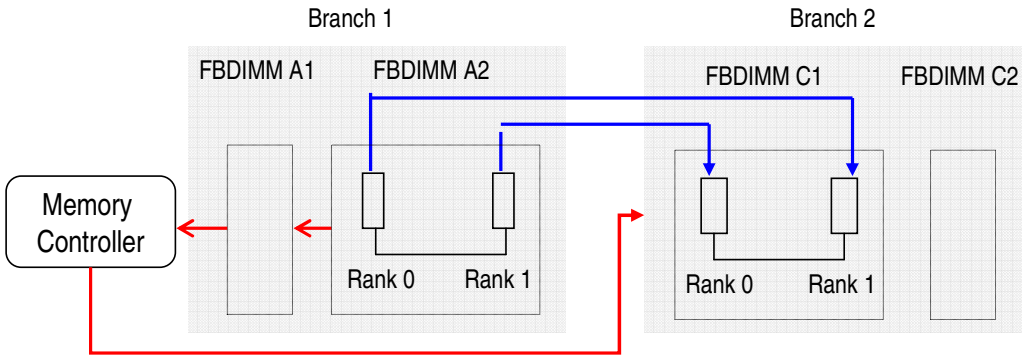

Figure 5-7: Data Migration Path

We perform the migration in hardware under the cover of the OS. For example, consider the migration shown in Figure 5-7 (in blue). We migrate data from FBDIMM A2 (rank*s* 1, 2) on branch 1 to FBDIMM C1 (rank*s* 1, 2) on branch 2. Data is first copied from FBDIMM A2 to A1 using the link between (A1, A2) on branch 1 and then from A1 to the

memory controller. It is then copied from the memory controller to C1 on branch 2. During any data copy, read/write in one branch can overlap with that in the other branch. However, overlapping reads/writes within a branch would lead to transfer conflict. After the migration is complete, the memory controller updates a hardware indirection data structure that then re-routes future accesses addressed to rank*s* in A2 to their current location in C1. This introduces an additional address translation step in the memory controller after it has generated the physical address from the linear address. However, since this is done in hardware we believe this translation can be done very fast. Designing an efficient hardware data structure is part of our future work. After this structure is updated, the local Power Managers responsible for the rank*s* on FBDIMM A2 would transition them to the appropriate low-power *state*.

Given that the *active* rank*s* in *state* $s_j = \{r_j \mid r_j \in n_j\}$ and *state* $s_k = \{r_k \mid r_k \in n_k\}$, where n_j and n_k are the number of *active* rank*s* in the respective *states*, we use the difference between *states* s_j and s_k to define the source rank set Δr_s and destination rank set Δr_d where data is always migrated from Δr_s to Δr_d during a *state* transition. *State* transition gives us two cases to deal with, ***Case I:*** *Shrinking memory size* $n_j > n_k$ $\Delta R_s = s_j - s_k$, $\Delta R_d = s_k$ and $\Delta R_d \subset \Delta R_s$, ***Case II:*** *Expanding memory size* $n_j < n_k$, $\Delta R_s = s_j$, $\Delta R_d = s_k - s_j$ and $\Delta R_s \subset \Delta R_d$. When transitioning the system from one *state* to another, the *Data Migration Manager* maintains the memory fully-interleaved in all system *states.*

Transition Time $\tau_{trans_{jk}}$ is the time taken to transition from *state* s_j to s_k. It is expressed as $\tau_{trans_{jk}} = \tau_{m_{jk}} + \tau_{p_{jk}}$, where $\tau_{p_{jk}}$ denotes the rank power *state* transition time (Figure 5-4) and $\tau_{m_{jk}}$ denotes the data migration time. We consider negligible transition

time from an *active* to a low-power *state*. Hence $\tau_{p_{jk}}$ is computed based on the reactivation time τ_{ra} only. Since rank*s* can transition in parallel , $\tau_{p_{jk}} = \max[\tau_{ra}(r)]$. $\tau_{m_{jk}}$ denotes the data migration time from *state* s_j to s_k. Let us consider the case when we shrink memory size during a *state* transition. *Ranks* in Δr_s would transition to one of the low-power *state*s and hence the data contained in those rank*s* have to be either migrated to the *active* rank*s* in *state* s_k or the *dirty pages* have to be written back to disk if rank*s* in Δr_d transition to the *offline* low-power *state* where the FBDIMM does not contain any valid content. Let us consider the case where the data is entirely transferred to the *active* rank*s* in *state* s_k. We define this as the migration data m_r (per rank). In this case a target *state* is selected such that the memory size is big-enough to hold the migrated data. The migration time per memory access $\tau^a_{m_{jk}}$ can be expressed as $\tau^a_{m_{jk}} = (\tau_{p_r} + \tau_{c_r}) + (\tau_{p_w} + \tau_{c_w}) + \tau_o$, where τ_{p_r}: source rank read time, τ_{c_r}: channel transfer time from source rank, τ_{c_w}: channel transfer time to destination rank, τ_{p_w}:destination rank write time and τ_o: time taken to update a hardware indirection data structure that re-directs future requests to the migrated data to current location. Hence for m_r data (per rank), the total migration time per rank is $\tau^r_{m_{jk}} = [(\tau_{p_r} + \tau_{c_r}) + (\tau_{p_w} + \tau_{c_w}) + \tau_o] * m_r$. Assuming $\tau_{p_r} + \tau_{c_r} = \tau_{p_w} + \tau_{c_w} = \tau_p$ where τ_p is the processing time per request we have $\tau^r_{m_{jk}} = [2\tau_p + \tau_o] * m_r$. Since data migration can happen in parallel between multiple rank pairs, the total data migration time $\tau_{m_{jk}} = \max \tau^r_{m_{jk}}$ for all rank pairs. Here we have assumed a simple migration scenario such that data from one

rank is written to one and only one rank and vice-versa. Exploring the other overlap scenarios would involve a queuing delay. We can similarly compute the transition time for the case of expanding memory size during a *state* transition.

Clearly, the migration overhead is directly proportional to the size of migration data m_r. For example, in our test bed, scaling down the memory subsystem from 16 *Ranks* to 15 requires migrating 12.5% per *page* but scaling down to 14 rank*s* requires migrating 25% of a *page* thus doubling the migration overhead.

Migration Data m_r is the amount of data (cache lines) per rank that needs to be migrated during a *state* transition. Given a total number of *pages* (p) in memory the total data distributed across all rank*s* is given by $D_t = p * \frac{s_P}{s_{CL}}$, where s_P: *page* size and s_{CL}: cache line size. Hence, the data per rank in *state* s_j with n_j rank*s* (n_j-way interleaved) is given by $d_{r_{s_j}} = \frac{D_t}{n_j}$. When transitioning the system from *state* s_j to *state* s_k with n_k rank*s* the DMM migrates the data such that it dynamically changes the memory interleaving from n_j-way in *state* s_j to n_k-way in *state* s_k. Hence when shrinking memory size, the migration data m_r is given by $m_r = M_t / n_k$, where total data to be migrated $M_t = d_{r_{s_j}} * (n_j - n_k)$. Hence, data per rank after migration is $d_{r_{s_k}} = d_{r_{s_j}} + m_r = D_t / n_k$. Similarly when expanding memory size the migration data m_r is given by $m_r = d_{r_{s_j}} - d_{r_{s_k}}$, where total data to be migrated $M_t = (d_{r_{s_j}} - d_{r_{s_k}}) * n_k$ and data per rank after migration $d_{r_{s_k}} = d_{r_{s_j}} - m_r = D_t / n_k$.

In our case, the migration data is the predicted application working set (N_{ws}) obtained from the **memAlloc** algorithm. Hence, we want to make the point here that maintaining the application's working set in memory not only reduces memory over-provisioning but also reduces the migration overhead.

Migration Energy: The energy consumed during migration from *state* s_j to *state* s_k, $c_{jk} * \tau_{trans_{jk}}$ can be expressed as

$$c_{jk} * \tau_{trans_{jk}} = n_k * p_{t_{jk}} * \tau_{p_{jk}} + p_{m_{jk}} * \tau_{m_{jk}}$$

where $p_{t_{jk}}$ is the transition power consumed by a rank and $p_{m_{jk}}$ is the power consumed in buffers during data migration. This second term is a sum of the buffer power, base FBDIMM power, DRAM refresh power, read power, link power and write power. We assume a close-page policy which is energy-efficient for interleaved memory. Hence we do not account for the energy spent in accessing open pages during migration.

5.3.2.4 Memory *Appflow*

Table 5-1 shows the SCSF, DCSF and ORSF for the memory subsystem. The *Data Migration Manager* monitors the appropriate *SCSF* and *DCSF* at runtime and computes the predicted application memory requirements using the **memAlloc** algorithm. This is given by (N_{ws} * s_p) where N_{ws} is the predicted application working set size in *pages* and s_p is the size per *page*. In addition, the *Data Migration Manager* also monitors the end-to-end memory access *delay*, *d* and the energy consumed by the memory subsystem in the current *state, e*. As shown in Table 5-1, these three features constitute the *ORSF* for the

memory subsystem. As shown in Figure 5-8, at runtime, they constitute the memory subsystem *operating point* in a 3-dimensional space. The *operating point* changes in response to the incoming workload. The *Data Migration Manager* triggers a *state* transition whenever the *operating point* goes outside the safe operating zone. This is depicted by the *decision,* d_Z, in the Figure. The *decision* is computed using the *optimization* approach discussed earlier.

SCSF	DCSF	ORSF
MemTotal	nr_dirty	Energy
Buffers	nr_writeback	Delay
HighTotal	nr_unstable	Predicted Application Working Set Size, (Nws*sp)
LowTotal	nr_page_table_pages	
SwapTotal	nr_mapped	
VmallocTotal	nr_slab	
HugePages_Total	pgpgin	
Hugepagesize	pgpgout	
VmallocChunk	pswpin	
	pswpout	
	pgalloc_high	
	pgalloc_normal	
	pgalloc_dma	
	pgfree	
	pgactivate	
	pgdeactivate	
	pgfault	
	MRC (Miss Ratio Curve)	
	HugePages_Free	
	VmallocUsed	
	Committed_AS	
	SwapFree	
	Dirty	
	Writeback	
	Mapped	
	Slab	
	MemFree	
	Cached	
	SwapCached	
	Active	
	InActive	
	HighFree	
	LowFree	

Table 5-1: Memory *Appflow* Features

Note that the safe operating zone is defined by threshold bounds along the *delay* (x) axis

given by [d_{min}, d_{max}], threshold bounds along the *energy* (z) axis given by [e_{min}, e_{max}] and the bounds along the application memory requirements (y) axis given by [m_{min}, m_{max}]. For our work, the threshold *delay* bounds are predetermined and kept set at that value, the threshold *energy* bounds is defined by [$e_{min} = 0$, $e_{max} = N*p_a$], where e_{max} represents the maximum power that is consumed by the memory subsystem when it is configured at its maximum capacity. The third threshold bound defined by the application memory requirements however varies. Let us understand this with the aid of an example.

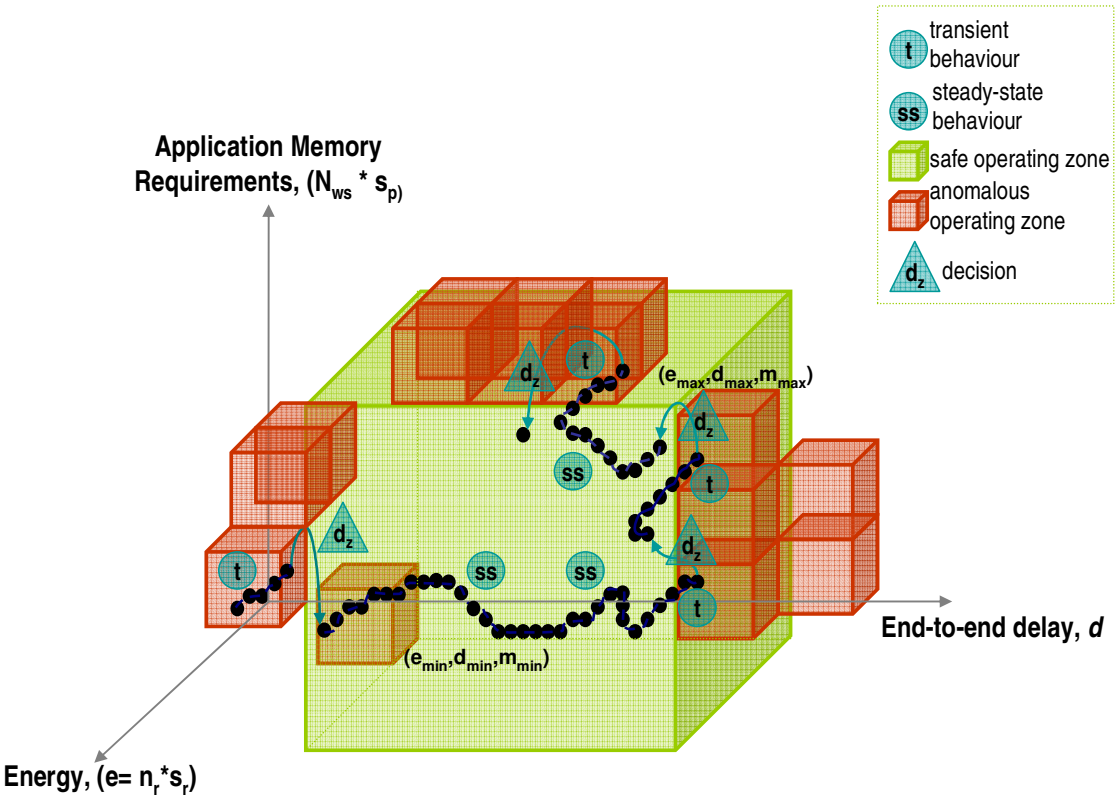

Figure 5-8: Trajectory traced by the memory subsystem operating point

Figure 5-9 depicts three different directions of movement of the *operating point* keeping variability only along the y-axis, keeping the x and the z axis constant. Figure 5-9 shows that given the initial position of the system *operating point*, the memory allocated to satisfy the application is given by (n_{r+1}*s_r), where n_{r+1} is the number of rank*s* in the *active*

state and s_r is the size of a rank. Hence the threshold bounds for this position of the *operating point* is defined by [(n_r*s_r), (n_{r+1}*s_r)]. This means that if the system *operating point* varies while staying within this threshold bound, no reconfiguration or *state* transition is required. This is shown by the *operating point* movement along the green arrow in the Figure 5-9. However, for the *operating point* movement along the blue arrow, the threshold changes to [(n_{r+1}*s_r), (n_{r+2}*s_r)]. Similarly, for the movement along the pink arrow, the threshold changes to [(n_{r-1}*s_r), (n_r*s_r)]

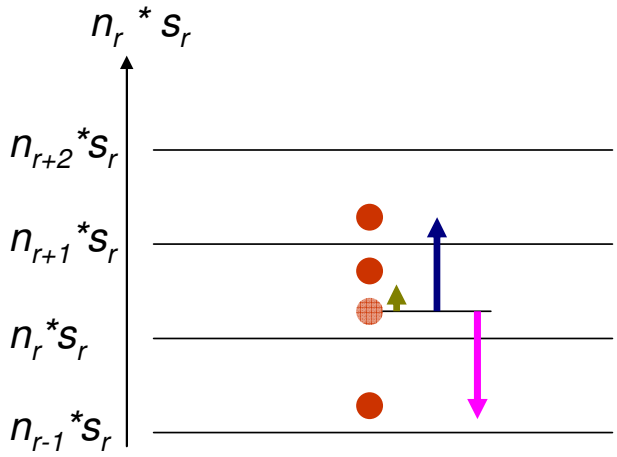

Figure 5-9: Threshold bounds for application memory allocation

In our current work, there is no reconfiguration owing to changes in the *operating point* along the *energy* axis. We achieve this by setting the *energy* threshold bounds to the minimum and maximum offered by the platform. The reason is because we opportunistically search for a *state* with minimum energy consumption that maintains the other performance requirements. There is no bound on a minimum or maximum value for this *energy* as long as it is within the limits of what the platform can support. The *energy* axis has been incorporated into the *ORSF* to support future work that would require working with fixed power budgets.

5.3.2.5 PERFWATT: Memory *Performance-per-Watt* Management Algorithm

The main steps of the algorithm, which is executed at the end of each observation epoch, is shown below. Given the memory subsystem is in a current *state* s_c, the *Data Migration Manager* initializes the maximum *performance-per-watt* $ppw_{\max}$, the target *state* set for exploration S_T and the set of feasible *states* S_F that satisfy all constraints. The *Data Migration Manager* monitors the system *operating point* to determine if a reconfiguration may be required. If the application's memory requirement does not change and the measured *delay* in the current *state* d_{s_c} lies within the threshold delay range, the *Data Migration Manager* maintains the same *state* (lines 2-4). Otherwise it looks for a *state* that gives the maximum *performance-per-watt* and also satisfies all the constraints. It first updates the target *state* set S_T by removing all the *states* from the set that have a memory size smaller than that required by the application as predicted by **memAlloc** (lines 6-10)**.** The *Data Migration Manager* then visits each *state* in S_T and computes the *delay* in that target *state*. If the *delay* is within the acceptable *delay* range it becomes a feasible solution and is added to the feasible set S_F (lines 13-14). It then computes the *performance-per-watt* (line 15). If it is greater than the maximum *performance-per-watt* $ppw_{\max}$, the value of $ppw_{\max}$ is updated with the new value and the target *state* is set to the feasible *state* (line 17-18). This is repeated until all the *states* in S_T have been evaluated. Finally the target *state* that has the maximum *performance-per-watt* is returned (line 23).

The complexity of our algorithm can increase exponentially with the size of the *state* space.

PERFWATT: Getting the optimal memory configuration for maximum *performance-per-watt* with miss ratio considerations

1: current state = s_c, $ppw_{\max} = 0$, $S_T = \{s \mid s \in S\} - \{s_c\}, S_F = \phi$
2: **If** $((d_{s_c} \geq d_{\min}) \,\&\&\, (d_{s_c} \leq d_{\max}) \,\&\&\, (n_{s_c} == N_{ws}))$
3: target state = s_c
4: **End if**
5: **Else**
6: **For** $(i = 0; i \leq N_s; i++)$
7: **If** $((n_{s_i} * s_r) < (N_{ws} * s_p))$
8. $S_T \leftarrow S_T - \{s_i\}$
9. **Endif**
10. **Endfor**
11: **While** ($S_T \neq \phi$))
12: $S_T \leftarrow S_T - \{s_t\}$
13: **If** $((d_{s_t} \geq d_{\min}) \,\&\&\, (d_{s_t} \leq d_{\max}))$
14: $S_F \leftarrow S_F \cup \{s_t\}$
15: $ppw_{s_t} = 1/(d_{s_t} * e_{s_t})$
16: **If** $(ppw_{\max} < ppw_{s_t})$
17: $ppw_{\max} = ppw_{s_t}$
18: target state = s_t
19: **Endif**
20: **End if**
21: **End while**
22: **End else**
23: **Return** target state

However, the *state* space is small for memory sizes generally found in servers. Hence the complexity of our algorithm is not high. However, we are improving it by using data-mining and rule-learning techniques that can be implemented in hardware. Note that our algorithm searches for a *state* that has the optimal *performance-per-watt* among all possible system *states*. It is therefore "optimal" for that *system*.

5.4 Experimental Evaluation

We validated our approach both on a real server as-well-as on a trace-driven memory system simulator. We configured the memory simulator to closely simulate the memory subsystem in our server unit. Let us first discuss the results on hardware followed by that on the simulator.

5.4.1 Evaluation of Technique in Hardware

Our test-bed server includes 2 dual-core Intel™ Xeon processors, 5000P Memory Controller Hub and 8 GB DDR2 FBDIMM memory. Its memory architecture is similar to the one shown in Figure 5-5. It consists of two branch*es,* two channel*s* per branch, two FBDIMMs per channel and two rank*s* per FBDIMM. The server can support a total of 8 FBDIMMs or 16 rank*s* in total.

We studied the *performance-per-watt* management for SPECjbb2005 benchmark on our server unit. SPECjbb2005 emulates a 3-tier client/server system with emphasis on the middle tier business logic engine. Its score is system *throughput* measured in BOPS (business operations per second).

Current servers cannot change the degree of memory interleaving dynamically in hardware. Dynamic interleaving can be emulated from the OS by using the memory hot-add and hot-remove features. However, the hot-remove feature is not yet fully-supported by current OSes. To get around this problem we emulated dynamic interleaving by manually reconfiguring memory as required by **PERFWATT**. Every reconfiguration was followed by a system restart. We also used a chipset specific address translation tool to

translate *pages* into their physical location in the memory subsystem. In this manner we determined the degree of the interleaving itself.

5.4.1.1 Analysis of *Performance-per-watt* Improvement for SPECjbb2005

We started with the maximum memory configuration and ran SPECjbb2005 on this configuration. The **PERFWATT** algorithm running on the system (on top of the OS) monitored the MRC for SPECjbb2005 using **memAlloc** and the average *delay* with the aid of chipset counters per rank. Our algorithm used these parameters to trigger a search for an optimal *state* as discussed in 5.3.2.3. We then manually reconfigured the memory subsystem to the desired configuration and restarted the system. We repeated this process until the application execution was complete. At the end of each phase that required a memory reconfiguration, we recorded the SPECjbb2005 BOPS and the corresponding power consumed by the memory subsystem (based on power consumption data from data-sheet).

Note that sometimes the algorithm returned a memory configuration that could not be configured in hardware. For example, since the channel*s* were configured to work in lock-step we always needed to populate FBDIMMs as a pair, one on each channel. Hence we could only work with even-numbered FBDIMMs. In that case, the algorithm returned a second sub-optimal solution that gave a smaller *performance-per-watt* compared to the optimal solution and we reconfigured the memory subsystem accordingly. Figure 5-10 shows the temporal variation of the optimal and sub-optimal solutions given by **PERFWATT** for SPECjbb2005.

In order to comparatively evaluate the *performance-per-watt* improvement given by our algorithm, at the end of each epoch that required a reconfiguration, we reconfigured the memory not only to that desired by **PERFWATT** but also to all other possible memory configurations allowed by the hardware. We ran SPECjbb2005 on each of these configurations and recorded the BOPS as well as the power consumed for each such configuration. We plotted the temporal variation of *performance-per-watt* (BOPS/Joules) for each such configuration as shown in Figure 5-11. We observed that our algorithm always determined the memory configuration (configuration IV in Figure 5-11) that gave the maximum *performance-per-watt* among all possible memory configurations. It gave a maximum *performance-per-watt* increase of 88.48% among all points where it was recorded during the application execution.

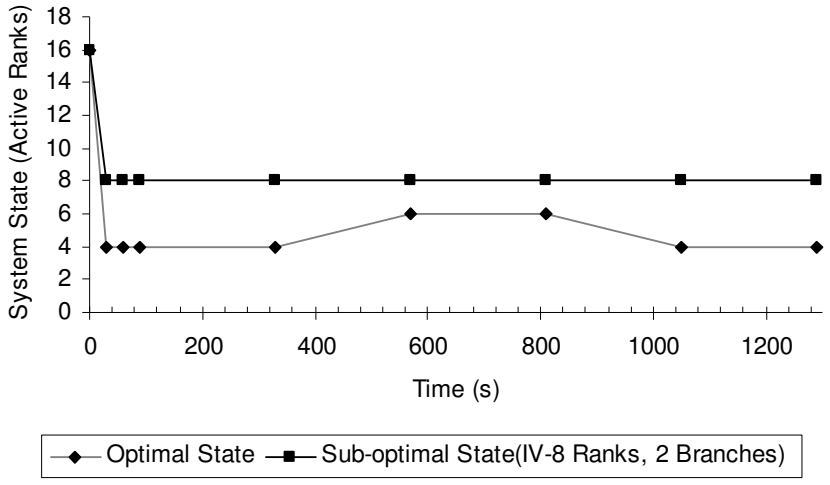

Figure 5-10: Optimal and sub-optimal *state*s

On the same server we ran SPECjbb2005 and measured the idle durations between memory accesses to each rank by using chipset counters. We got an energy savings of 4.47% (189.6 J) for an immediately '*suspend*' algorithm when a memory rank is

transitioned to a low-power '*suspend*' *state* as soon as it becomes idle. This compares to about 48.8% (26.7 kJ) power savings with our technique on the same server running the SPECjbb2005.

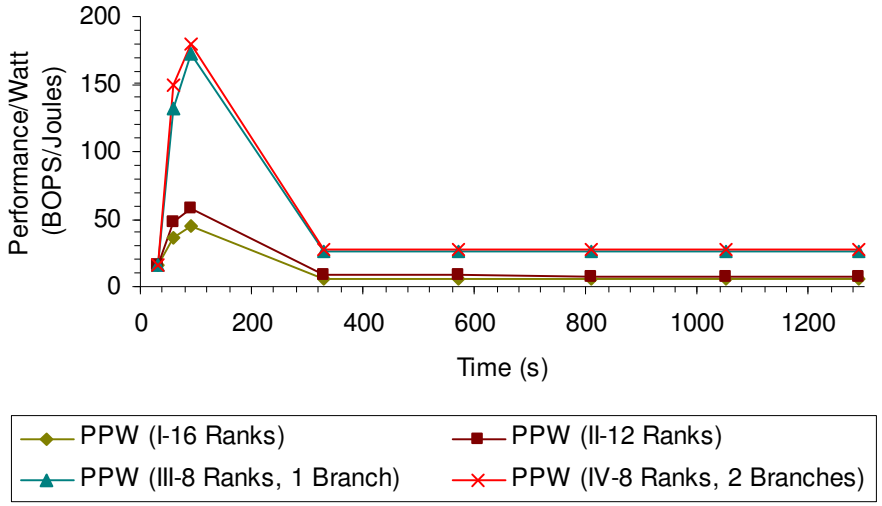

Figure 5-11: Performance-per-watt comparison

5.4.1.2 Algorithm Adaptivity to SPECjbb2005

SPECjbb2005 launches an additional warehouse at the end of each observation epoch that executes randomly selected business operations from an in-memory database of operations. Instead of computing the MRC, the **memAlloc** algorithm used the benchmark's heap usage at the end of each warehouse to predict the memory requirements of SPECjbb2005. This is because it was not possible to measure the number of hits per *page* accurately from the OS to compute the MRC. However as can be seen from Figure 5-11, this approximate approach still gave the memory configurations with the maximum improvement in *performance-per-watt* among all possible configurations. We also instrumented the Linux kernel to index the memory *pages* starting at the head of the LRU active list until it fills the heap used. This is the *working set* for SPECjbb2005 and comprises the migration data M that is to be dynamically interleaved on the memory

configuration given by **PERFWATT.** Figure 5-14 plots these *pages* as a percentage of the total *pages* in the LRU *active* list. As expected, this graph varies inversely as the percentage of over-provisioned memory as shown in Figure 5-12. Figure 5-12 shows the temporal variation of the percentage of the total allocated heap that remains unused by SPECjbb2005. Figure 5-13 plots the memory size (in ranks) that is predicted to be required by SPECjbb2005 by using **memAlloc**. The 'actual ranks' plotted in Figure 5-13 is the ceiling value of the 'calculated ranks'. By comparing Figure 5-12 and Figure 5-13 we see that the memory size varies inversely with over-provisioned heap as expected. Also notice that the optimal and sub-optimal ranks computed by **PERFWATT** (Figure 5-10) are always higher than the 'calculated rank*s*' of Figure 5-13. This is because consolidating the *working set* on these 'calculated ranks' maintains the application memory requirements but it significantly increases the *delay* and hence violates the *delay* constraint. Hence these solutions remain infeasible.

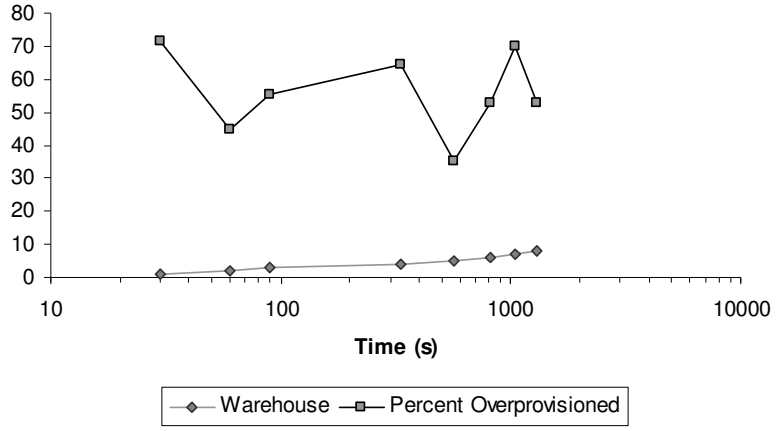

Figure 5-12: Dynamic heap usage of SPECjbb2005

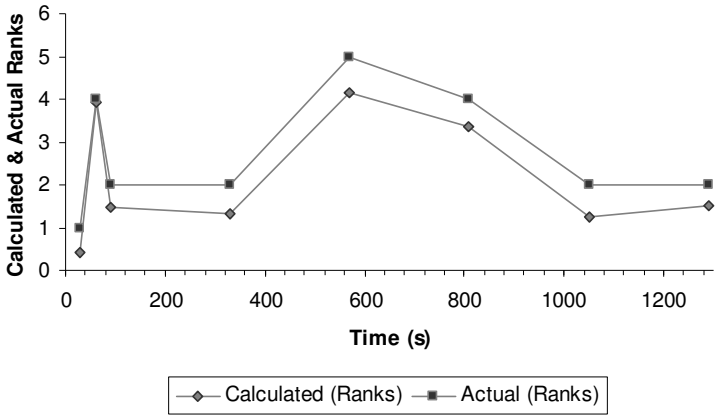

Figure 5-13: Variation of SPECjbb2005 working set

Also note from Figure 5-12 that around 600 sec into the application's execution, the over-provisioning reduces close to 30%. As can be seen from Figure 5-10, it is around this time that the algorithm increases the memory size from 4 to 6 rank*s* in anticipation of a heavy workload arrival phase. Similarly we notice from Figure 5-12 that the over-provisioning increases around 800 sec. However the algorithm maintains the same memory size (6 rank*s*). At about 1000 sec, when the over-provisioning further increases, the algorithm reduces the memory size from 6 rank*s* back to 4 rank*s*. The algorithm has a tendency to latch on to previous memory configurations and initiate reconfigurations only when significant over-provisioning is detected. It works conservatively because it accounts for the overhead involved in *state* transitions. This is discussed in the following section.

5.4.1.3 Analysis of Migration Overhead

Figure 5-14 plots the migration overhead associated with the migration data M for SPECjbb2005. This migration overhead has been computed for the solution that gives the

maximum *performance-per-watt* (solution IV shown in Figure 5-11). Notice that the migration overhead (16 to 8 rank*s*) is very small at the end of the first SPECjbb2005 warehouse launch, mainly due to small-sized migration data. Consequently, the algorithm allows this *state* transition from 16 rank*s* to 8 rank*s* (Figure 5-10). However, at the other warehouses the migration overhead (8 to 4 rank*s*) increases considerably with the increase in the size of the migration data. Hence the algorithm did not allow this *state* transition from 8 to 4 rank*s*. Instead it paid the migration overhead one time and maintained the memory configuration at a steady *state* with 8 rank*s* distributed across two branches.

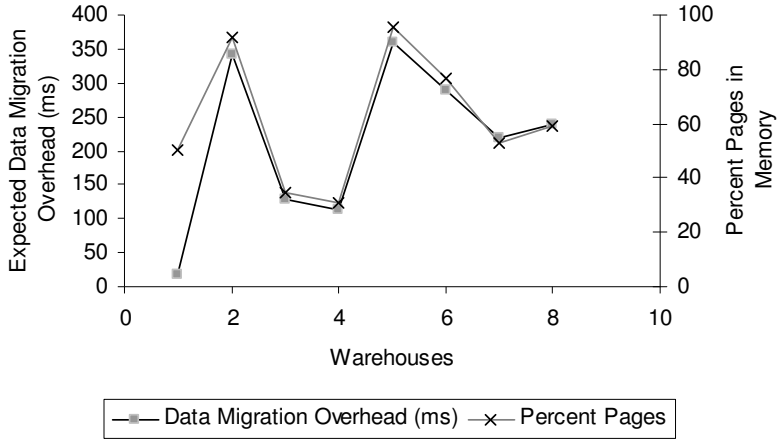

Figure 5-14: Migration overhead

Figure 5-15: Comparison of migration strategies

5.4.1.4 Impact of Migration Strategies on SPECjbb2005 Performance

Figure 5-11 shows the *performance-per-watt* obtained for two solutions (III-8 ranks, 1 branch & IV-8 ranks, 2 branch*es*). Note that these two solutions have the same number of rank*s* but different physical location in the memory hierarchy. However solution IV provides a higher *performance-per-watt* compared to solution III. Figure 5-15 plots the SPECjbb2005 BOPS measured at the end of each warehouse for both solutions III and IV. Solution III gives a performance drop of 5.72% for SPECjbb2005 when compared to IV. Our algorithm was able to effectively identify this difference with the aid of the temporal affinity prediction technique discussed in Section 5.3.2.2II and chose solution IV over III thus giving the maximum *performance-per-watt* for SPECjbb2005.

5.4.2 Evaluation of Technique in Simulation

We used ***gcc*** traces to evaluate our algorithm in the DRAMsim [Wang et. al. 2005] simulator. We simulated DDR FBDIMM memory and used a close-page row buffer management policy to emulate a fully-interleaved memory similar to that used in our server unit. We configured the memory with 4 branch*es* and 4 rank*s* per branch as the maximum configuration supported by the platform.

5.4.2.1 Workload Prediction Models

We use an efficient workload prediction model that is used by our algorithm to reconfigure the degree of interleaving for *performance-per-watt* management. We used a standard linear extrapolation technique (linear regression [Hastie et. al. 2001]) for future load prediction based on past history of load evolution. Since linear predictors are

sensitive to the observation/prediction window [Ghanbari et. al. 2007] we trained the predictor model on training data sets with a number of fixed window and variable window sizes.

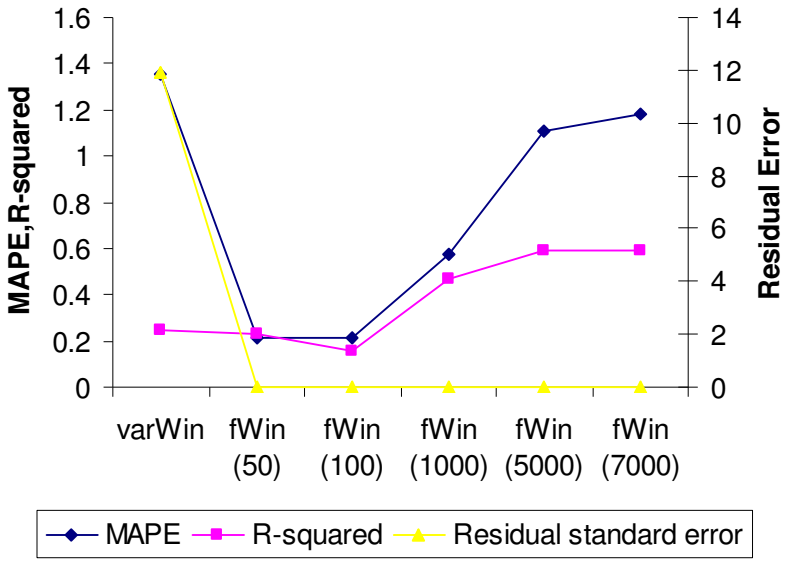

Figure 5-16: Comparison of different predictors

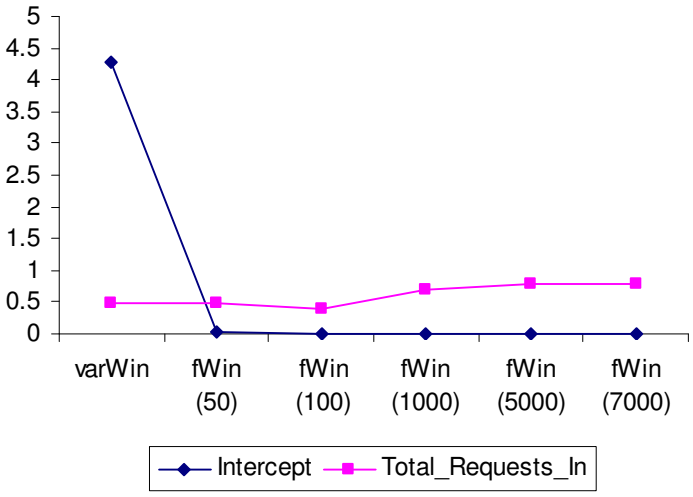

Figure 5-17: Workload prediction model

We divided the original data-set into training and test data sets by using the k-means technique [Cluster2007]. Figure 5-16 shows the Mean Absolute Percentage Error (MAPE) value for each model which is a measure of the accuracy of prediction

performance of the models on the test data. Figure 5-16 also shows the coefficient of determination R^2 [r^{2}2002] for each model where R^2 is the proportion of variability in a data set that is accounted for by the model. Figure 5-17 shows the prediction model that we used.

5.4.2.2 *Performance-per-Watt* Comparison with Existing Techniques

We compared the *performance-per-watt* obtained by running 6 different algorithms on the DRAMsim simulator using ***gcc*** memory traces. Figure 5-18 plots the *performance-per-watt* (PPW) given by each algorithm for ***gcc*** traces. The first four algorithms in Figure 5-18 are based on traditional power management techniques. Algorithms 1, 2 and 3 transition the memory rank*s* to the corresponding low-power *state* immediately on detecting idleness. Algorithm 4 transitions the memory rank to an appropriate low-power *state* by comparing the length of the idle duration to pre-determined threshold values.

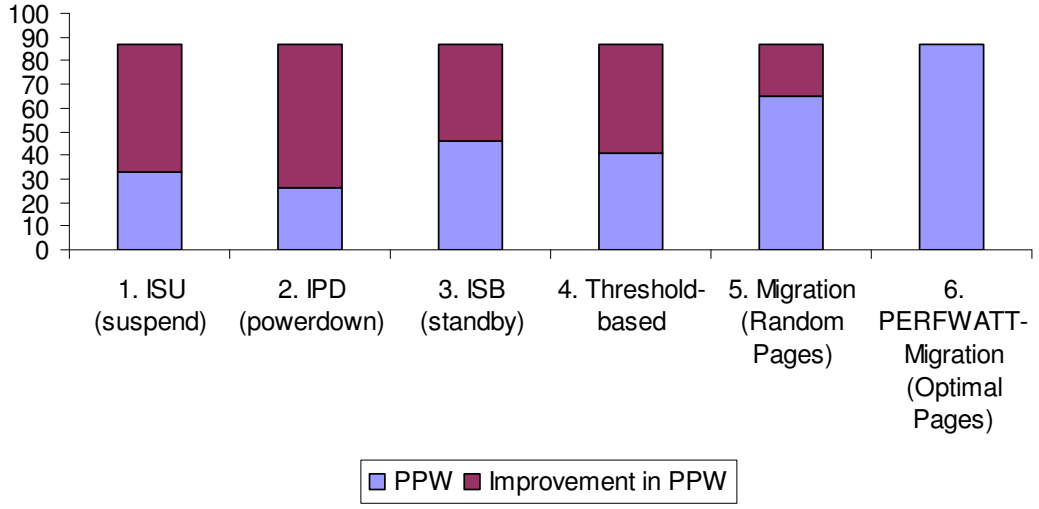

Figure 5-18: Comparison of *performance-per-watt*

Algorithm 6 in Figure 5-18 is our **PERFWATT** algorithm while algorithm 5 is a variant of **PERFWATT** that involves dynamic data migration of random number of *pages* as

opposed to N_{ws} *pages* used by **PERFWATT**. As can be seen from the Figure, **PERFWATT** gives the maximum *performance-per-watt* among all the other algorithms. Figure 5-18 also plots the improvement in *performance-per-watt* given by **PERFWATT** as compared to each other algorithm. It gives a *performance-per-watt* improvement of 89.7% over the best performing algorithm based on traditional technique (ISB) and gives an improvement of 34.81% compared to its variant algorithm 5 that migrates random *pages*.

5.4.2.3 Algorithm Adaptivity

Figure 5-19 plots the total number of active rank*s* that the memory was configured to during the entire run of ***gcc*****.** Our algorithm adapts well to the bursty nature of ***gcc*** memory traffic. For example, the memory traffic suddenly increases around 350,000 and lasts until 450,000 cycles followed by a lull-period that lasts until 600,000 cycles. As can

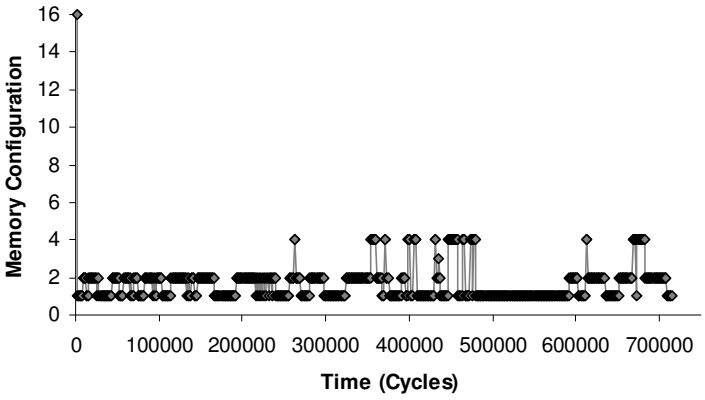

Figure 5-19: Optimal *state*s and *state* transition

be seen from Figure 5-19, our algorithm scales up the memory configuration to 4 rank*s* (2 rank*s* per branch) to adapt to the increase in traffic and scales it down to 1 rank during the lull-period. However, there are frequent reconfigurations during the initial period when

the ***gcc*** traffic fluctuates a lot. This goes to show that the algorithm is sensitive to changes in the incoming traffic. We are working to bring the sensitivity to an acceptable level by experimentally determining the appropriate observation epochs for the global DMM.

5.4.2.4 Algorithm Complexity Analysis

The complexity of our algorithm can be expressed as $O(n-1)$, where the total number of *state*s n is an exponential function of the number of rank*s* per branch and the total number branch*es*. It is expressed as follows

$$n \sim= [(k^b + (1-k)^r]/8 \qquad \ldots (5)$$

where, n is the number of *state*s, b is the number of branch*es*, r is the number of rank*s* per branch and $k \sim= 3.846$ is an experimentally determined constant. Figure 5-20 shows

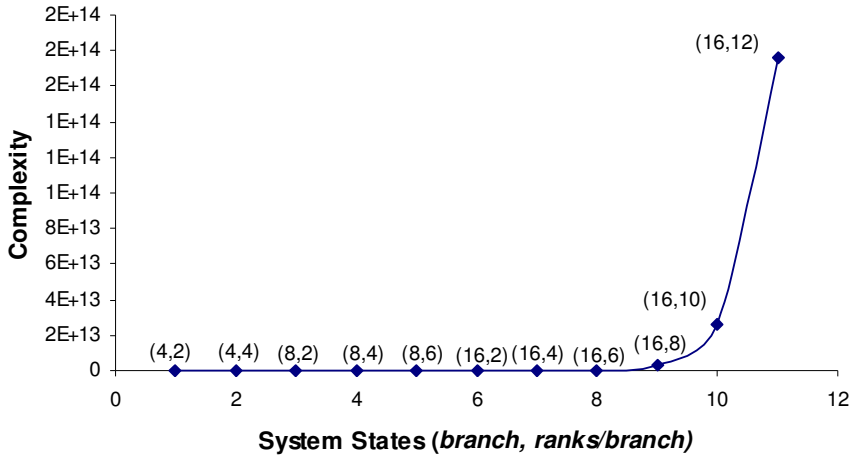

Figure 5-20: Algorithm complexity analysis

the complexity of our algorithm plotted against the number of system *states*. As expected, the complexity increases exponentially with the total number of *states*. However it starts increasing exponentially around a value of (16, 8) which stands for 16 branch*es* and 8 rank*s* per branch *(*128 rank*s)*. Hence, the complexity of the algorithm is low-enough for

smaller memory capacity. However, for huge memory capacity (which may very soon become a reality in servers with multi-core processors), we have to devise effective runtime strategies with low computation complexity. We are currently applying data mining and rule learning techniques to enhance our algorithm for runtime performance

5.5 Conclusion

In this Chapter, we presented a technique to optimize the *performance-per-watt* of fully-interleaved memory subsystem and analyzed its performance. Our approach yields an energy saving of about 48.8 % (26.7 kJ) compared to traditional power management techniques measured at 4.5%. It gives a transition overhead of about 18.6ms leading to energy saving of 1.44kJ per millisecond of transition overhead time and a maximum *performance-per-watt* improvement of 88.48%.

We are currently validating our results on a number of memory traces and studying the algorithm scalability, sensitivity to threshold values and performance with multiple observation windows for DMM and PMs. We are applying data mining and rule learning techniques to implement an efficient real-time version of the **PERFWATT** algorithm that significantly reduces the runtime complexity of the algorithm. We are also investigating non-traditional BIOS interleaving techniques that can be exploited by the OS and the memory controller in a co-operative manner to achieve low–overhead efficient dynamic power management techniques. We are also extending our technique for servers running multiple applications.

6 AUTONOMIC POWER AND PERFORMANCE MANAGEMENT OF HIGH-PERFORMANCE SERVER PLATFORMS

6.1 Introduction

In this Chapter we focus on power and performance management of a high-performance server platform within the data center hierarchy. We discuss how we apply the autonomic computing paradigm to design a server platform that continuously maintains its power and performance under varying workload conditions. In this Chapter, we build on our work related to memory power management as discussed in Chapter 5 and extend that to encompass power and performance management of the whole platform. The central idea behind this work is to proactively detect and reduce resource over-provisioning in server platforms such that it is just right-sized to handle the requirements of the application. In this manner we can save power by transitioning the over-provisioned resources to low-power *state*s and simultaneously maintain performance by satisfying the application resource requirements. The challenge is to be able to do that for each component within a server platform – processor, memory, network, I/O such that they co-operate and co-ordinate to reduce the energy savings for the whole platform while maintaining platform performance. For the purpose of this work we consider a server platform that consists of multi-core processors and multi-rank memory subsystem. Both the processor and/or the memory subsystem are dynamically reconfigured (expanded or contracted) to suit the application resource requirements. The reconfigured platform creates the opportunity for power savings by transitioning any unused platform capacity (processor/memory) into low-power *state*s for as long as the platform performance remains within given acceptable thresholds. The platform power

expenditure is minimized subject to platform performance parameters, which is formulated as an optimization problem.

This work addresses the following research challenges.

1. Given the heterogeneity and dynamism of resources (processor, memory, network card, I/O) within a data center server platform, how do we capture the power and performance management objective across platform resources in a coherent and consistent manner such that the resources co-ordinate and collaborate to work towards the common objective of the platform's power and performance management. If this challenge can be addressed at the platform-level it can be applied across all hierarchies of a data center.
2. How do we characterize dynamic behavior and resource requirements of data center server workloads? What are the necessary and sufficient features for any platform resource that can accurately help capture this information?
3. Given that we have effectively characterized the workloads, how do we identify appropriate reconfigurations of the underlying platform resources that entail minimum reconfiguration overhead while delivering the maximum platform *performance-per-watt*?
4. Can we leverage existing fine-grained power management techniques for platform resources (e.g. processor, memory, I/O etc) to work with our platform *performance-per-watt* management technique?
5. What are the cost, run-time complexity and reconfiguration overhead associated with our techniques and how they can be reduced to attain a greater return-on-investment?

6.2 Autonomic Server Platform

Figure 6-1 shows the architecture of our server platform that is to be power and performance managed. It consists of multi-core processors and a multi-rank memory subsystem. The architecture of the memory subsystem is similar to that used in Chapter 5. For purposes of this work we consider a single unified (L2) cache between the processors and the memory subsystem. The memory subsystem is connected to the processor subsystem through the front side bus depicted as FSB in the Figure. In our experimental server unit, the number of FSBs matches the number of memory branch*es* with an FSB per processor. We show a similar architecture in Figure 6-1.

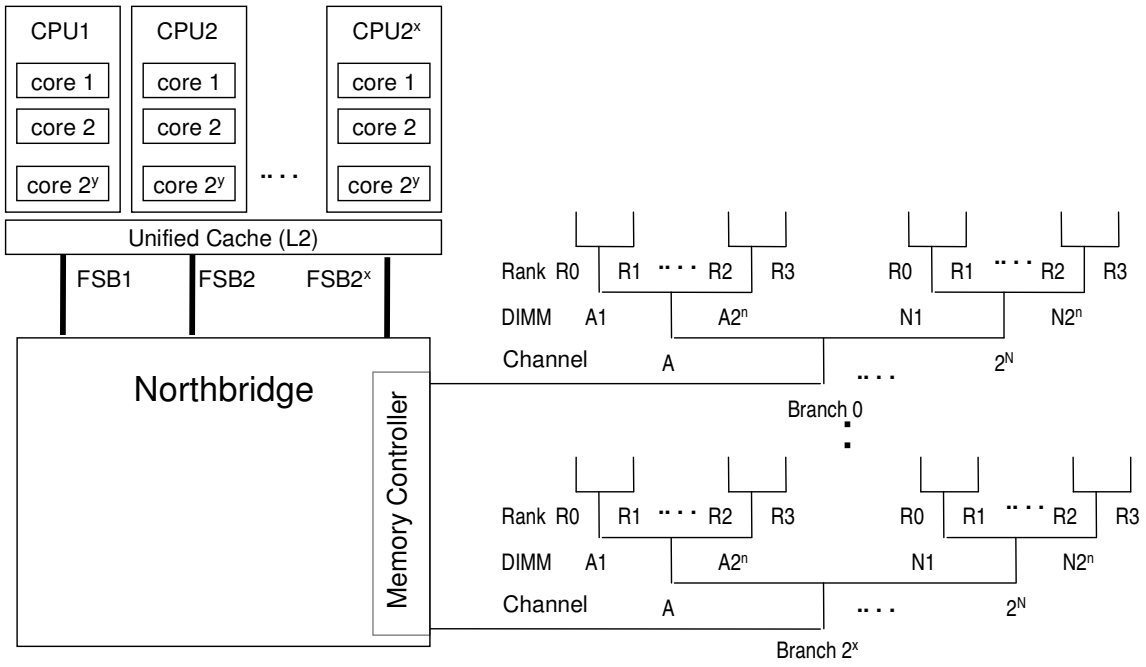

Figure 6-1: Server platform with multi-core processors and multi-rank memory

All memory accesses (cache misses in L2 cache) are serviced over the FSBs. The memory controller servicing memory accesses by routing memory accesses to the right memory rank*s* within the memory subsystem. We take this server platform and transform

it into an *Autonomic Platform* that can self-manage its power and performance. This is shown in Figure 6-2.

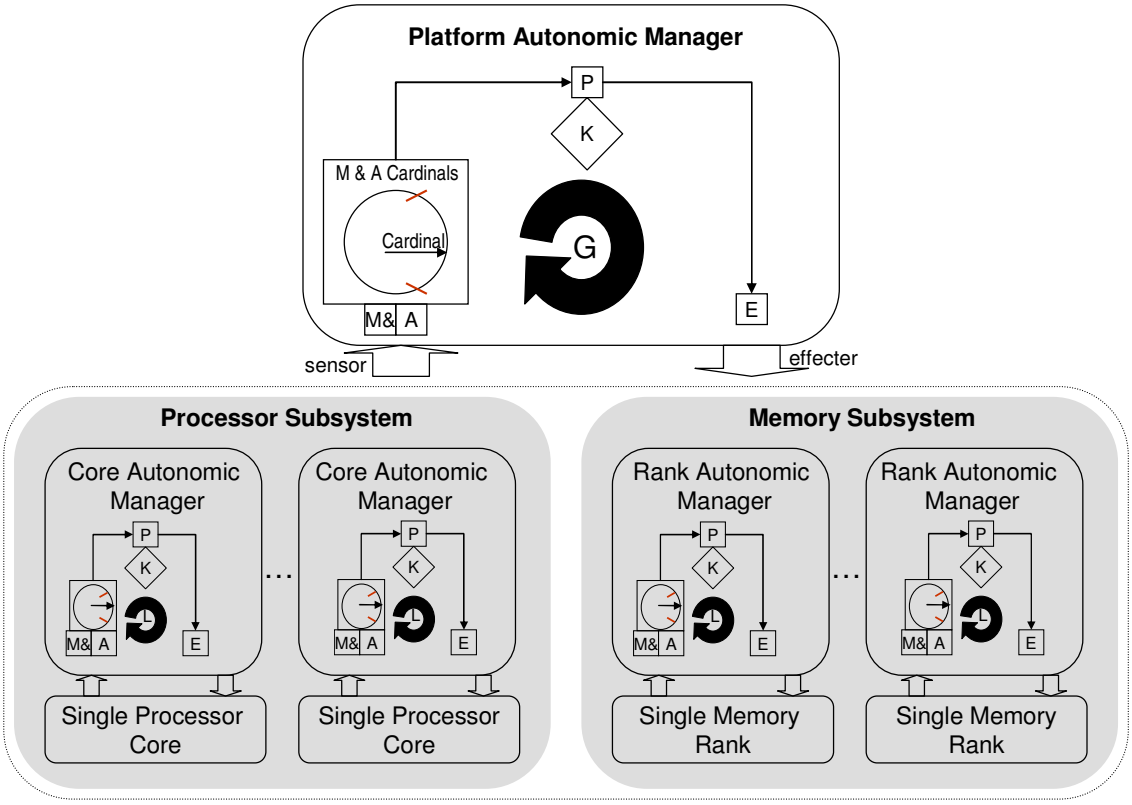

Figure 6-2: Autonomic power and performance managed platform

This *Autonomic Platform* is derived from our archetypal autonomic component of Chapter 3. The *Autonomic Platform* consists of two hierarchies. At the upper-level the *Managed System* is the whole platform that consists of all the multi-core processors and multi-rank memory. It is managed by the *Platform Autonomic Manager*. This is denoted by the global control loop G. At the lower-level, there are two *Managed Systems* - the processor core managed by the *Core Autonomic Manager* and memory rank managed by the *Rank Autonomic Manager*. They are denoted by the local control loop L.

6.2.1 Modeling the Platform Managed System

At any hierarchy of our *Autonomic Platform*, the *Managed System* is modeled as a set of *states* and *transitions*. Figure 6-3, Figure 6-4 and Figure 6-5 show the *state* space for the platform, the memory rank [FBDIMM2006] and the processor *core* [Isci et. al. 2006] respectively.

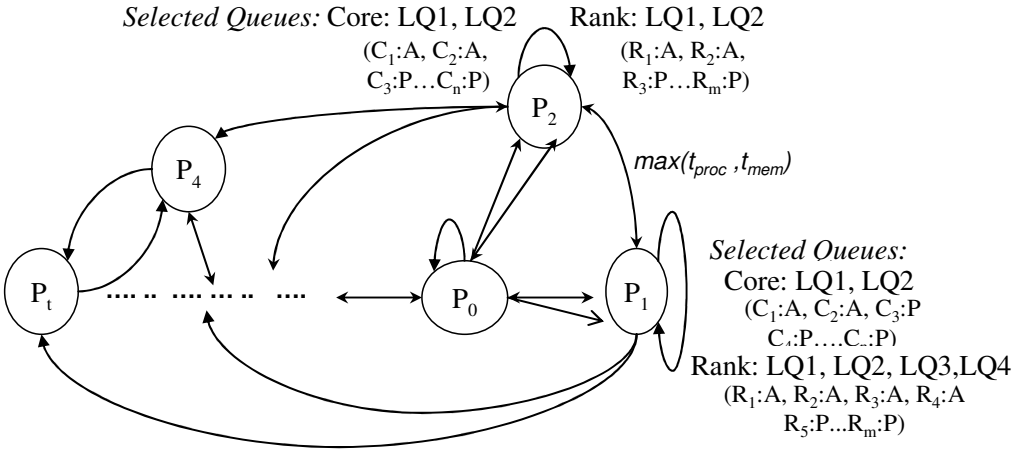

Figure 6-3: Platform *state* space

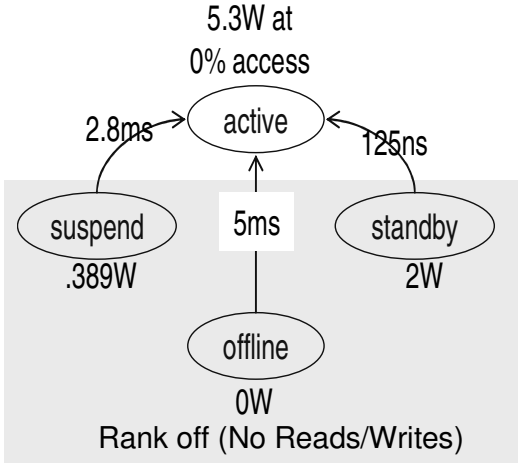

Figure 6-4: Rank *state* space

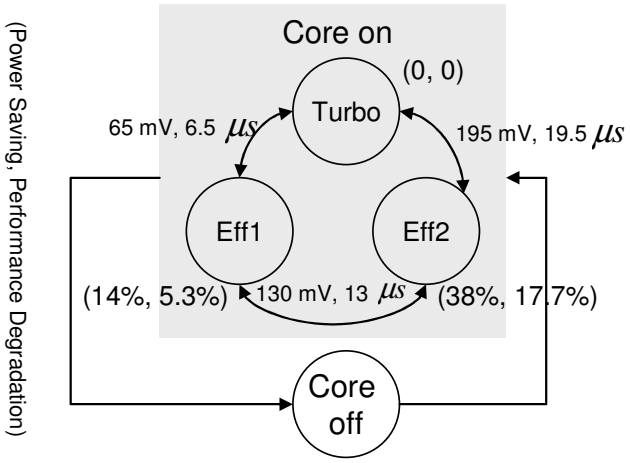

Figure 6-5: Core *state* space

Each *state* is associated with power consumption and performance values. For example, the processor core has three DVFS [Isci et. al. 2006] power and performance *states* when the core is on. The power and performance values are shown in the Figure 6-5. It also shows the power consumption and transition time for the core to transition from one *state* to another. Similarly, the values are shown for the memory rank. The memory rank *state* space has been discussed in Chapter 5. The platform *state* is defined by the number of processor cores in the *turbo state*, the number of memory rank*s* in the *active state* and the physical location of these rank*s* within the memory hierarchy. Hence the power consumed by a platform *state* is a sum of the power consumed by its constituent parts (cores, ranks). The platform *state* transition power is a sum of the *state* transition of the constituent parts (i.e. processor and memory). However the *state* transition time is a maximum of the transition time of its constituent parts.

The performance of the platform *state* depends on the physical configuration of the platform (number of cores, number of rank*s*, physical location of rank*s*) in that *state* and

the rate of arrival and nature of the incoming workload (processor –intensive, memory-intensive). In Chapter 5 we have quantified the impact of the number and physical location of rank*s* within the memory hierarchy on the overall memory performance (end-to-end memory access *delay*). In this Chapter we incorporate those findings into our platform power and performance model. We use the basic platform performance model as shown in Figure 6-6 [Hennessy et. al. 2000].

The performance parameter that we manage is the platform *response time*. Clearly, the platform *response time* depends on the *state* of the platform. For example, the *CPI* (cycles per instruction) parameter determines the amount of time (number of CPU cycles) taken to process an instruction/job and depends on the configuration of the processor subsystem, the higher the number of processor cores in *turbo state* the smaller the CPI and hence the smaller the *response time* and vice-versa.

Performance Parameter: *Response time/Execution Time = **IC**(CPI*
+ Cache References/Instruction
** Cache Misses/Cache Reference*
** Cache Miss Penalty)*

Where Cache Miss Penalty = Average Memory Access Time
*= Hit Time * (1- Miss Rate) + Miss Rate * Miss Penalty*

IC: This is the reflective of the incoming workload

Figure 6-6: Platform Performance Model

The *cache miss penalty* parameter determines the amount of time taken to process a cache miss or in other words a memory access. It is nothing but the average memory access time also defined as the end-to-end memory access *delay* in Chapter 5. We have already shown how the *delay* is dependent on the memory configuration. As can be seen from the performance model, the smaller the end-to-end memory access *delay,* the smaller is the platform *response time* and vice-versa. In addition, the platform performance model also

depends on the nature and rate of the incoming workload. The impact of the rate factor is captured by the *instruction count (IC)* parameter and the nature of the incoming workload is captured by the *cache references/instruction* and the *cache misses/cache reference* parameters in the performance model. For example, for a highly memory intensive workload, the *cache misses/cache reference* would be very high. This may lead to a higher platform *response time*.

6.2.2 Modeling the Platform Autonomic Manager

In this Section we describe how the *Platform Autonomic Manager* works with the individual *Core* and *Rank Autonomic Managers* to collectively manage the power and performance of the whole platform. Every observation epoch, the *Platform Autonomic Manager* predicts the platform resource requirements for the application. It then identifies an optimal platform configuration/*state* that meets the application resource requirements and at the same time maintains the platform *response-time*.

The *Platform Autonomic Manager* uses the platform *Appflow*, to determine when such a *state* transition/reconfiguration is required for the platform in order to maintain its power and performance. It determines the target *state* to transition to by solving a *performance-per-watt* optimization problem.

6.2.2.1 Platform *Appflow*

The objective of the *Platform Autonomic Manager* is to ensure that the platform resources (processor/memory) are configured to meet the dynamic application resource requirements such that any additional platform capacity can be transitioned to low-power

*state*s. In this manner the *Platform Autonomic Manager* saves platform power without hurting application performance. The *Platform Autonomic Manager* uses the *Platform Appflow* to help determine when to trigger platform reconfiguration and whether the reconfiguration should scale-up or scale down the platform capacity. The *SCSF, DCSF, ORSF features* for the *Platform-level Appflow* are listed in Table 6-1.

SCSF	DCSF	ORSF
Total Number of Processor Cores	Current Number of Processor Cores in Power State TURBO	Request Loss
Total Number of Memory Ranks	Current Number of Memory Ranks in Power State ACTIVE	Response Time
Total Branches, Channels per Branch	Current (Branch Id, Channel Id) of ACTIVE Memory Ranks	MRC (Miss Ratio Curve)
, FBDIMMs per Channel, Ranks per FBDIMM	Number of Misses	End-to-end Delay
Core voltage and frequency states	Number of Hits per Page	Platform Power
Rank power states	Size of LRU Active List	
Rank Size (MB)		

Table 6-1: Platform *Appflow* Features

The *Platform Autonomic Manager* monitors the *SCSF* and *DCSF* features to compute the *ORSF*. These features determine the platform *operating point* in an n-dimensional space at any instant of time during the lifetime of the application. The *Platform Autonomic Manager* manages the platform power and performance by maintaining the platform *operating point* within a predetermined safe operating zone. This is depicted in Figure 6-7. It predicts the trajectory of the *operating point* as it changes dynamically in response to changes in the nature and arrival rate of the incoming workload and triggers a platform reconfiguration whenever the *operating point* drifts outside of the safe operating zone. For example, a sudden increase in the rate of arrival of processor-intensive jobs would increase the average platform *response time* because the processor is not configured (too few cores in a high power processing *state*) to handle this increase in traffic. This leads to a reduction in the rate of processing jobs and thereby increasing the average *response time* for the platform jobs. This causes the platform *operating point* to drift out of the safe

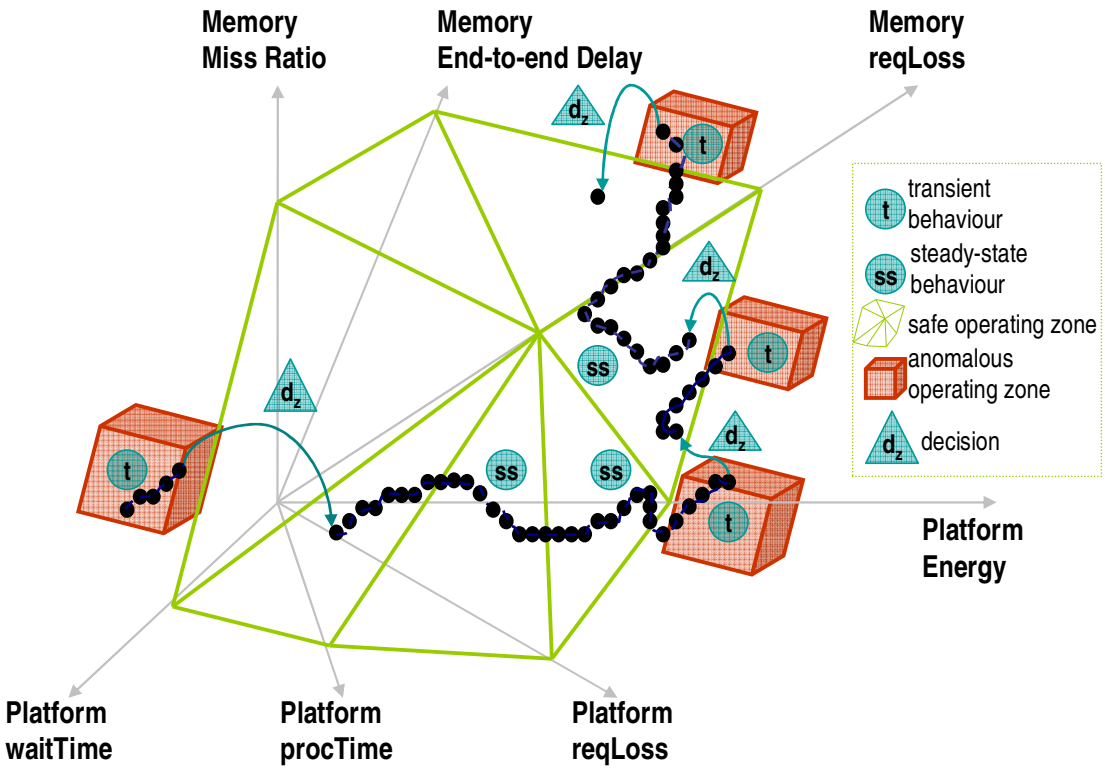

Figure 6-7: Trajectory of the platform operating point owing to fluctuations in the ORSF

operating zone. The *Platform Autonomic Manager* monitors the rate of change of the *response time* parameter to predict the nature of the incoming workload and reconfigure the platform to a *state* such that the *response time* is maintained within the safe operating zone while consuming the minimum platform power possible. In this scenario the platform reconfiguration would entail an increase in the number of processor cores in the *turbo state*. The memory subsystem is not reconfigured. In a similar manner, a sudden increase in the arrival rate of memory intensive jobs may also increase the platform *response time* requiring a platform *state* transition where the memory subsystem is reconfigured without touching the processor subsystem. In order to determine the individual impact of the processor subsystem and the memory subsystem on the platform *response time*, the *Platform Autonomic Manager* monitors additional parameters such as

memory miss ratio, memory end-to-end delay and *memory request loss* and uses this information to arrive at a proper decision in terms of platform reconfiguration. We discuss this in details when we discuss our modeling approach of an *Autonomic Platform* in the following Section.

6.2.2.2 The Platform Power and Performance Optimization Problem

Whenever the *Platform Autonomic Manager* triggers a reconfiguration of the platform, it causes the platform to transition from the current *state* to a target *state* that would bring the platform *operating point* back into the safe operating zone. The search for this ideal target *state* is formulated as an optimization problem and discussed in 6.3.1.9 This reconfiguration decision of the upper-level *Platform Autonomic Manager* is further refined by the lower-level *Autonomic Managers*. For example, if a certain target platform *state* is required to have a specific core in *turbo state*, the local *Core Autonomic Manager* would find the best target *state* for the core among the three possible *states – turbo, eff1* and *eff2* that would maintain the core *operating point* within the safe operating zone. Similarly, this applies to the platform memory subsystem. In that sense, the global and the local control loops of Figure 6-2 work in tandem, with the decision of the upper-level *Autonomic Manager* being refined by the lower-level *Autonomic Managers* and the lower-level *Autonomic Managers* giving feedback back to the upper-level *Autonomic Manager*. In that sense the *Platform Autonomic Manager* shares a hierarchical relationship with the lower-level *Autonomic Managers*. The lower-level *Autonomic Managers* however work independent of one another.

6.3 Modeling and Simulation Approach

In this Section, we discuss how we model our autonomic power and performance managed platform model. Our modeling approach is motivated by the modeling and simulation of hierarchical adaptive computing architecture presented in [Wang1986]. We first introduce the modeling and simulation framework used followed by detailed discussion of each component of our model.

6.3.1 Discrete Event System (DEVS) Modeling and Simulation Framework

For this work, we use the DEVS modeling and simulation framework. DEVS provides a sound modeling and simulation framework and is derived from mathematical dynamical system theory. It supports hierarchical, modular composition and reuse and can express discrete time, continuous, and hybrid models. DEVS allows for the construction of hierarchical simulation models composed of atomic and coupled models. Each atomic model is assigned to an atomic simulator, and atomic models as components within coupled models are assigned to a coupled simulator. Coupled models are assigned to coordinators, while coupled models as components within larger models are assigned to coupled-coordinator simulators. The simulators keep track of the events and execute the simulation model-defined methods based on the events list. Each component model of this case-study is formulated as a DEVS Atomic Model. A DEVS Atomic Model is a finite state machine and is defined as [Zeigler2000],

$$M_{Atomic} =< X, S, Y, \delta_{\text{int}}, \delta_{ext}, \lambda, t_a >$$

where,

X is the set of inputs accepted by the model

S is the set of states of the model

Y is the output set generated by the model

$\delta_{\text{int}} : S \to S,$ captures internal state transitions for the model

$\delta_{ext} : Q \times X \to S$, captures state transitions for the model in response to external inputs

$\lambda : S \to Y$, is the output function that maps a state to an output from the output set.

t_a : is the time–advance function for remaining in a state before an internal state transition occurs

$Q = \{(s,e) \mid s \in S, 0 \le e \le t_a(s)\}$ is the total state set, e is the elapsed time since last transition.

The internal transition function δ_{int} dictates the system's new state when no events occur since the last transition. The external transition function δ_{ext} dictates the system's new state when an external event occurs and this state is determined by the input, *x*, the current state *s* and how long the system has been in this state *e*. Figure 6-8 shows the model of our autonomic power and performance managed platform. It models the server platform (Figure 6-1) with multi-core processors and multi-rank memory subsystem. The *service requester* generates jobs to be processed by the platform. The job is processed by a processor *core*. If the *core* requires access to the *cache* to complete the processing, it forwards the job to the *cache* and blocks until the data arrives from the *cache.* Now, a *cache* access may lead to a cache hit or a cache miss.

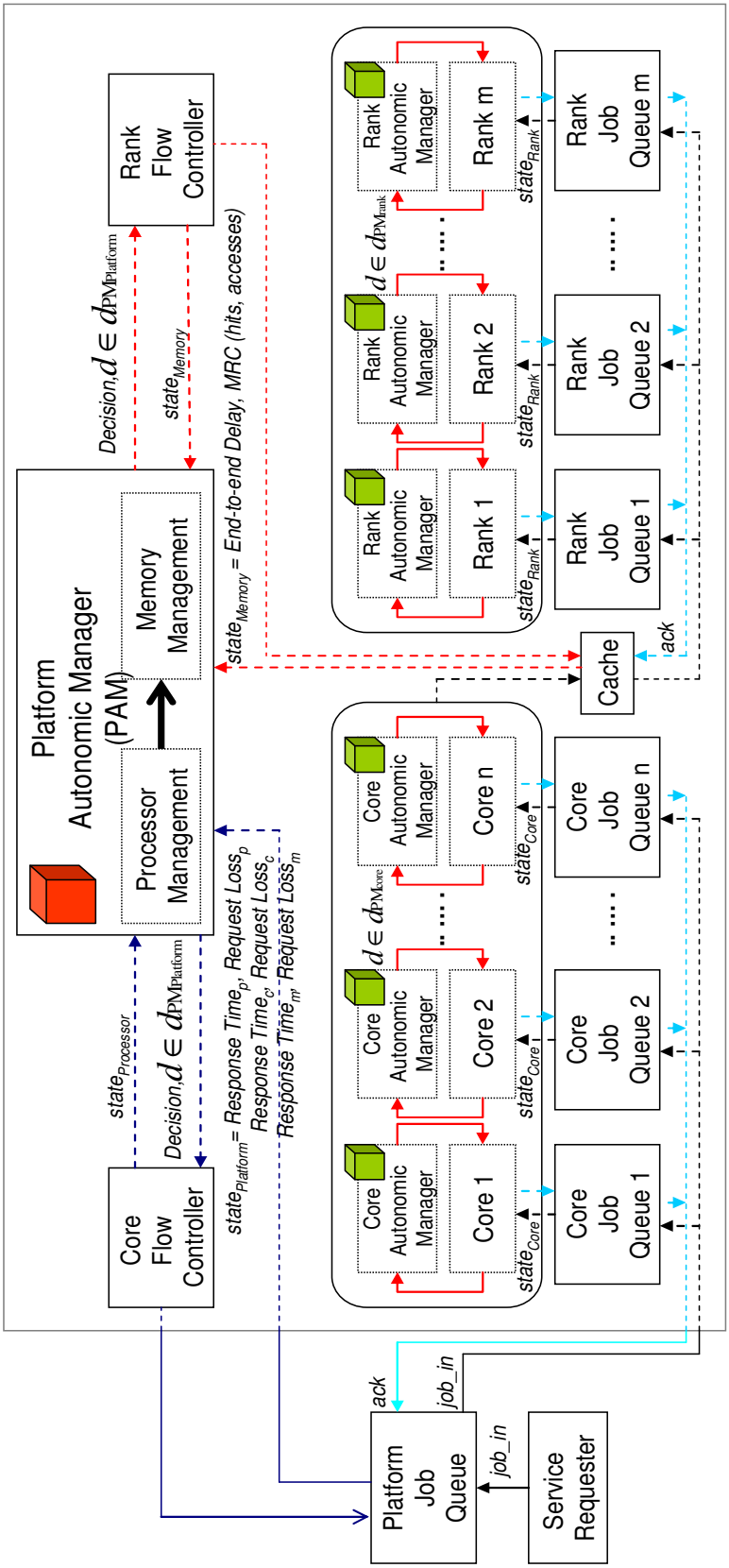

Figure 6-8: Autonomic Power and Performance Managed Platform Model

If it is a cache hit, the data is forwarded from the *cache* to the *core*; the *core* unblocks itself and finishes processing of the job. If however, it leads to a cache miss, the data needs to be brought from the platform memory. In this case, the *cache* forwards the job to an appropriate memory rank for further processing. In what follows, we discuss each module in details and introduce the DEVS formalism for these different components constituting the model.

6.3.1.1 Service Requester

The *service requester* models the client that generates jobs to be processed by the server platform. These jobs are stored in the *platform job queue*. The *service requester* remains in the '*active*' phase for the duration of job inter-arrival time. At the end of this time, it sends out a job to the *platform job queue*. The inter-arrival time can be set to simulate any job arrival behavior ranging from random, uniform, normal or any other traffic pattern obtained from real workload generators. The *service requester* generates three different types of jobs – those that are processed by a processor *core*, those that are

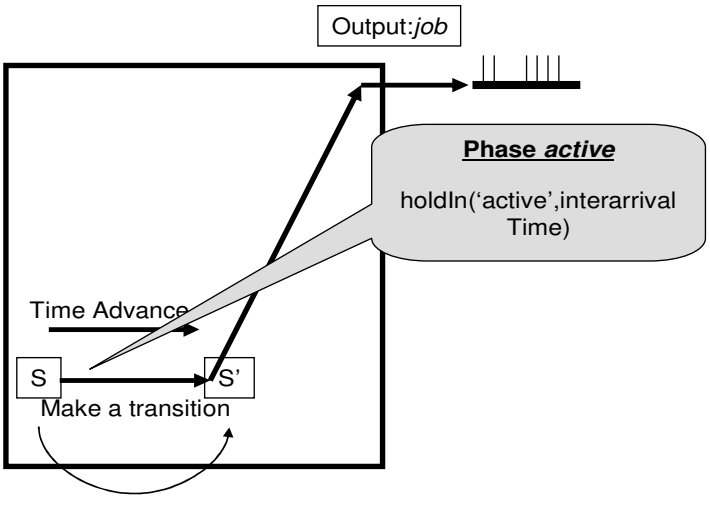

Figure 6-9: Internal transition function

processed by the *cache* and those that are processed by a memory rank. Figure 6-9 gives the internal transition function for the *service requester*. The Service Requester can be formally defined as a DEVS atomic model as follows:

$$M_{service\,\mathrm{Re}\,quester} =< X, S, Y, \delta_{\mathrm{int}}, \delta_{ext}, \lambda, t_a >$$

S = {"*passive*","*active*"} x R^+

X = { }

Y = job ∈ Job, where Job is a class defined by attributes that signify the nature of the job such as the number of cache references, number of cache misses etc. It is identified by a unique job identifier.

$\delta_{\mathrm{int}} : S \rightarrow S$

Transitions from '*active*' to '*active*' after time t_a of 'jobInterArrivalTime'

t_a = 'jobInterArrivalTime' in phase "active"

∞ in phase "passive"

$\lambda("active", \sigma)$ = jobId ∈ I , the set or positive integers

6.3.1.2 Platform Job Queue

Jobs generated by the *service requester* are queued in the *platform job queue* before they are sent to the platform for processing. The jobs are then routed to an appropriate *core job queue* for processing. The *platform job queue* measures three important platform performance parameters namely *requestLoss, waitTime* and *procTime*. The amount of time spent by a job in the *platform job queue* before it is sent for processing is measured by the *waitTime* metric. The *procTime* metric is defined as the actual processing time for the job. The sum of *waitTime* and *procTime* is given by the *responseTime* metric for the

platform. This is the same as the platform *response time* performance parameter discussed earlier. We consider a finite job queue length for the *platform job queue*. A job is deleted from the queue once the acknowledgement (ack_p) arrives for the job. This creates space for incoming jobs in the queue. If however the job queue is full any incoming job is treated as a lost request and measured by the *requestLoss* metric. The rate of change of the above mentioned performance parameters are good predictors of the dynamic workload resource requirements. For example, if the average platform *responseTime* is very high, the platform is taking a longer time to process jobs which calls for scaling out the platform capacity by either increasing the number of processor cores in *turbo* state and/or increasing the number of memory ranks in the *active* power state. This information is used by the *platform autonomic manager* to reconfigure the platform to the desired *state* such that it meets the workload resource requirements.

The *platform job queue* also aids the *platform autonomic manager* in reconfiguring the platform as discussed below. The *core flow controller* is responsible for maintaining current *state* information for the platform. Once the *platform autonomic manager* determines the desired platform *state*, it is sent to the *core flow controller* which in turn sends it to the *platform job queue*. Based on the core-specific configuration of that *state,* (number of cores in *turbo state*) the *platform job queue* forwards jobs to the respective *core job queue* for processing. For example if the platform is in *state* p_i (*x* cores *turbo*, *y* ranks *active*), the *platform job queue* forwards jobs to *core job queue (1, 2 ,…, x)* only. Note that we do not differentiate between processor cores in this model making the assumption that a job has no perceivable advantage in being processed by one core as opposed to another within the same processor or within different processors.

The *platform job queue* passivates in phase '*wait*' until it receives a job from the *service requester*. This is shown in the external transition in Figure 6-10. It then transitions into phase '*out*' where it forwards the job to an appropriate *core* for processing. It however does not delete the job from the job queue until it receives an acknowledgement for the job from the *core*. The acknowledgment for jobs can arrive in any order depending on when and by whom (core, cache, memory) the jobs were processed. Thus a job sent later can be acknowledged before a job that was sent earlier in time. Hence, the *platform job queue* continuously forwards jobs to the *cores* whether they are new arrivals or waiting for acknowledgement from the *core*. Hence the *platform job queue* passivates in '*wait*' only when the queue is completely empty. This is demonstrated in the internal transition function in Figure 6-11. Occasionally, the *platform autonomic manager* may request for the *requestLoss, waitTime* and *procTime* of jobs from the *platform job queue*.

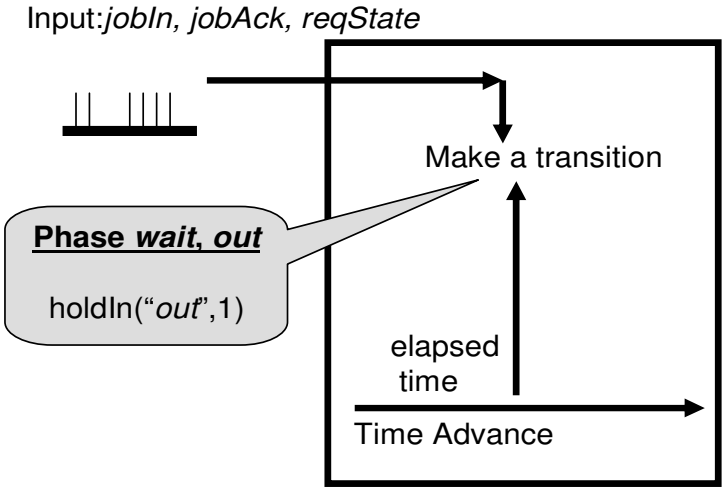

Figure 6-10: External transition function

The *platform job queue* transitions to phase '*out*' as shown in the external transition function in Figure 6-10 and outputs the values at the end of that phase. Here we assume

that the *platform job queue* takes the same time to output a job to the *core* for processing as to output the state consisting of the *requestLoss, waitTime* and *procTime* values to the *platform autonomic manager*. It transitions back to '*out*' if the queue is not empty else it passivates in '*wait*'. The *core flow controller* also communicates with the *platform job queue* to inform any reconfiguration decision taken by the *platform autonomic manager*. The *platform job queue* uses this information to update its routing information when routing a job for processing to the processor *cores*.

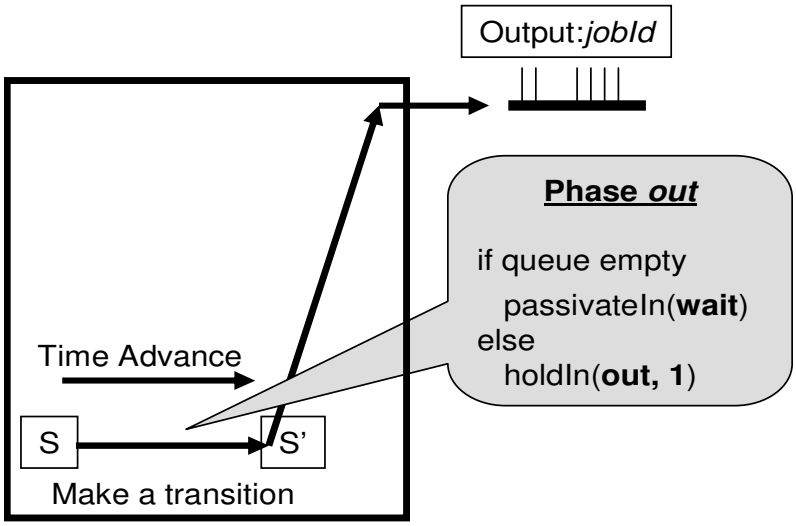

Figure 6-11: Internal transition function

The *platform job queue* can be formally defined as a DEVS atomic model as follows:

$$M_{platformJobQueue} = < X, S, Y, \delta_{\text{int}}, \delta_{ext}, \lambda, t_a >$$

S = {*wait*, *out*} x R^+

X: {jobIn, jobAck, reqState }

Y: {jobId, (*requestLoss, waitTime* and *procTime*)}

$\delta_{ext} : S \times X \rightarrow S$,

- In response to input 'jobId' it stores the jobId in the queue if the queue is not full

- In response to input ‘jobAck’ it deletes the job from the queue.

- In response to input ‘reqState’ it sends an entity that consists of (*requestLoss, avWaitTime, avJobProcTime*) and few other attributes.

$\delta_{\text{int}} : S \rightarrow S$,

- is the transition from *out* to *out* after time t_a which is the time taken to send the output $y \in Y$

- is the immediate transition from *out* to *wait* where time t_a is zero

t_a: is the time taken to send an output while in phase ‘*out*’, or time taken to transition to phase ‘*wait*’ which is zero.

$\lambda(\text{"}out\text{"}, \sigma) =$ jobId and/or (*requestLoss, avWaitTime, avJobProcTime*)}

6.3.1.3 Cache

A job processed by the processor *core* may require access to the *cache* and if it is a cache miss it would in addition require access to the memory subsystem. In our platform model the *cache* serves two tasks –of processing an incoming job, of serving as a job queue for the memory subsystem similar to what the *platform job queue* does for the whole platform. As part of this role the *cache* acts as the memory controller from Figure 6-1. If the job leads to a cache hit, the *cache* processes the job and sends an acknowledgement (ack_m) back to the *core* that forwarded the job. If however it leads to a cache miss, it forwards the job to the appropriate rank *job queue* for processing by the associated rank.

The *cache* also monitors important performance parameters that aid the *platform autonomic manager* in its decision process. As we have already seen, the platform *state*

consists of a processor-specific configuration and a memory-specific configuration. In determining a desired platform *state* simply based on the platform performance parameters, the *platform autonomic manager* does not have sufficient information to study the impact of the processor subsystem and the memory subsystem on the overall platform performance parameters. Hence the *platform autonomic manager* monitors additional performance-related information that is specific to the memory subsystem to determine for example if the recent increase in average platform *responseTime* is owing to an under-provisioned processor or an under-provisioned memory or both. This additional information is provided by the *cache* subsystem. It monitors three important performance metrics for the memory subsystem – the *miss ratio curve*, the *average end-to-end memory access delay* and the *memory request loss*.

Now let us understand how the *cache* subsystem measures this information. We have already mentioned that in case of a cache miss, the job gets forwarded to the appropriate rank *job queue* for processing by the associated rank. In addition, the job is added to the end of the internal *LRU active list* maintained by the *cache* that mimics the operating system *LRU active list*. Note that the incoming job emulates a memory access to a specific memory *page frame number* (PFN). The *cache* maintains the *LRU list* of PFNs based on which *PFN* the specific job translates to. In addition it also keeps a count of the number of hits per *PFN* as part of the *page* data structure. The *cache* uses this information to compute the *miss ratio curve* for the memory subsystem once every t_p cycles. *Miss ratio curve* computation is discussed in Chapter 5 Section 5.3.2.1 in the *memAlloc* algorithm. Now, after a job is forwarded to a rank *job queue* for processing, it forwards the job to the associated rank. Upon completion, the rank sends an

acknowledgement (ack_r) to the respective rank and to the *cache*. The *cache* measures the *end-to-end memory access delay* as the difference between the time the job was sent out to the memory subsystem for processing and the time the corresponding ack_r arrives. The memory *request loss* is measured similar to the *requestLoss* parameter measured by the *platform job queue.* Periodically, the *platform autonomic manager* queries the *cache* to collect the *miss ratio curve, end-to-end delay* and *memory request loss* performance metrics. It uses these memory related performance parameters (*miss ratio curve (MRC), end-to-end delay, memory request loss*) combined with the platform related performance parameters (*waitTime, procTime, requestLoss)* to arrive at a decision regarding the new platform *state.*

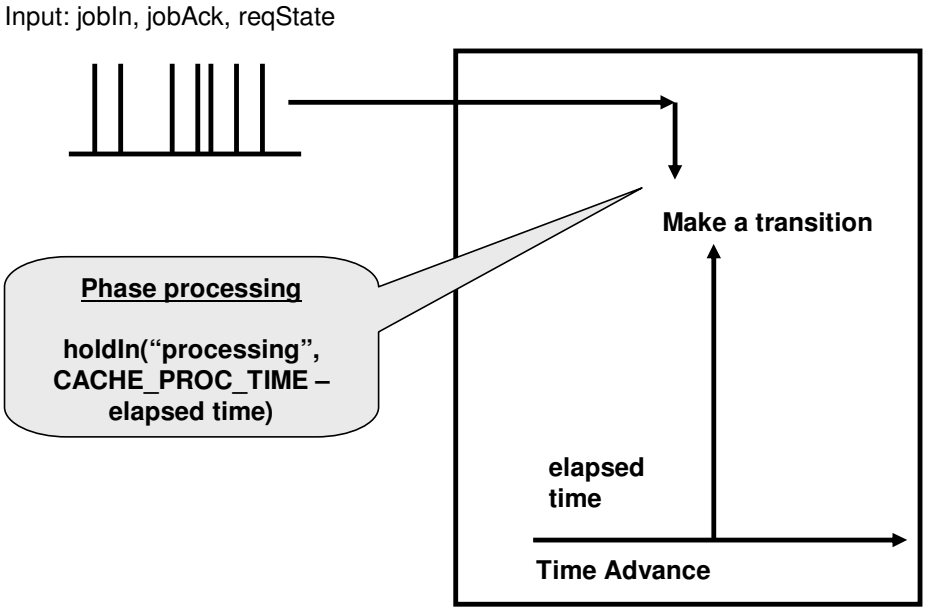

Figure 6-12: External transition function

The *cache* passivates in *empty* if its internal job queue is empty. When a job arrives that can be processed by the *cache* it transitions to *processing* and holds in *processing* for job processing time. At the end of the job processing time, it sends an acknowledgement back

to the *core*. This is shown in Figure 6-13. If the job cannot be processed by the *cache* and needs to be forwarded to a memory rank, it transitions into phase '*out*' and forwards the job to an appropriate rank *job queue* for processing. If however, the *cache* receives a job while it was in *processing*, it simply queues up the job in its internal queue to look at it after it has finished processing the current job. This is shown in Figure 6-12. Once done, it looks at the next job in the queue and transitions again to *processing* or *out* based on the nature of the job. It responds to requests from the *platform autonomic manager* by forwarding the *miss ratio curve, memory request loss* and the *average end-to-end memory access delay* parameters.

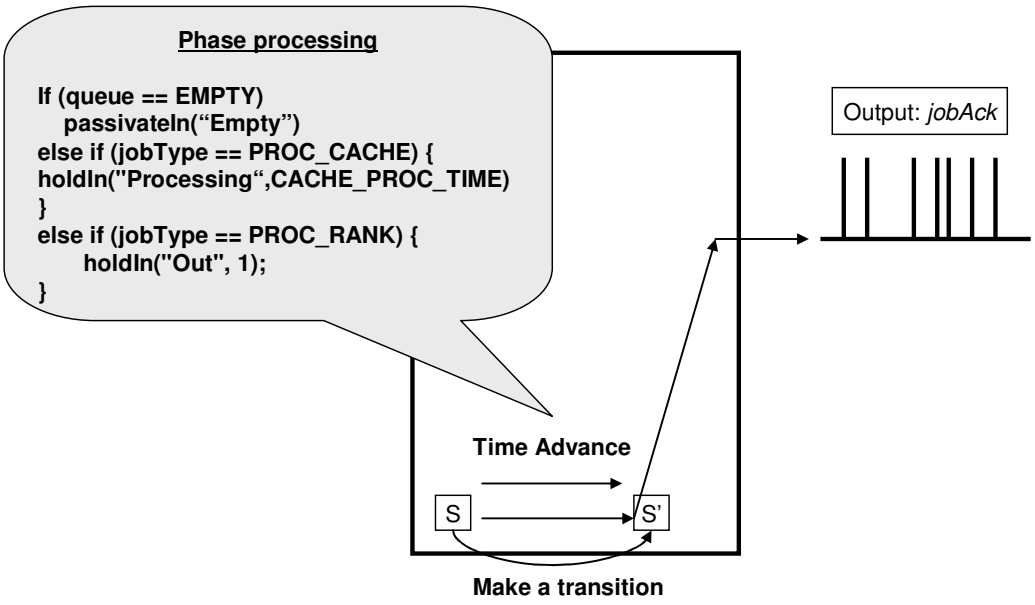

Figure 6-13: Internal transition function

The *cache* can be formally defined as a DEVS atomic model as follows:

$$M_{cache} = < X, S, Y, \delta_{\text{int}}, \delta_{ext}, \lambda, t_a >$$

S = {*empty*, *processing, out*} x R^+

X: {jobIn, jobAck, reqState}

Y: {jobId, (*MRC, delay, reqLoss*), jobAck}

$\delta_{ext} : S \times X \rightarrow S$,

- In response to input 'jobId' it stores the jobId in the queue.

- In response to input 'jobAck' it deletes the job from the queue.

- In response to input 'reqState' it sends an entity that consists of (*MRC, delay*).

$\delta_{\text{int}} : S \rightarrow S$,

- is the transition from *out* to *out* after time t_a which is the time taken to send the output $y \in Y$

- is the immediate transition from *out* to *empty* where time t_a is zero

- is the transition from *processing* to *out* where time t_a which is the time taken to send the output $y \in Y$

- is the transition from *out* to *processing* where time t_a is the time taken to begin processing.

- is the transition from *processing* to *processing* where time t_a is the time taken to begin processing

t_a: is the time taken to send an output while in phase '*out*', the time taken to begin processing a job or time taken to transition to phase '*wait*' which is zero.

$\lambda(\text{"}out\text{"}, \sigma) =$ jobId and/or (*MRC, delay, reqLoss*)}

$\lambda(\text{"}process\text{"}, \sigma) =$ jobAck and/or (*MRC, delay, reqLoss*)}

6.3.1.4 Core Flow Controller and Rank Flow Controller

The *core flow controller* and the rank *flow controller* are responsible for maintaining the current processor and the current memory configuration in the current platform *state*. The *platform autonomic manager queries* these components to learn about the current *state* that the platform is in.

6.3.1.5 Rank Atomic Model

The rank processes memory requests. It is also the smallest unit of the memory subsystem that can be independently power managed. It *passivates* in the initial phase of *'offline'* as long as its local rank *job queue* is empty. Once the rank *job queue* forwards jobs to the rank for processing, it holds in phase '*to_active*' for the power *state* transition time to move from *state 'offline'* to *state 'active'*. It then holds in '*processing*' for the processing time of the incoming job and finally passivates in phase '*active*'. However, if it is already in the '*active*' phase and receives incoming jobs, it directly goes to phase '*processing*' and finally passivates in phase '*active*'. This phase change captures the behavior that the rank processes requests only in the power *state* 'active'. The power *state*s and transitions for the rank are shown in Figure 6-4. After processing a request, the rank outputs an acknowledgement for the processed job by sending the job id out to the rank *job queue*, the *cache* and the *core*. If the rank receives an input command from the rank *autonomic manager*, and it is in any of the '*active*', '*standby*', '*suspend*' or '*offline*' *state*s it will hold in any of the *'to_active', 'to_standby', 'to_suspend'* or *'to_offline'* phases depending on the old power *state* and the new power *state*s required by the command. The transition time is different for different source destination pair. It then

passivates in the target phase ('*active*', '*standby*', '*suspend*' or '*offline*'). The rank does not accept any inputs if it is either in phase '*processing*' or phases *'to_active', 'to_standby', 'to_suspend'*, *'to_offline'*. Some of the internal transitions and external transitions are shown in Figure 6-14, Figure 6-15 and Figure 6-16. Occasionally, the rank *autonomic manager* can query the rank its current power *state*. The rank transitions to the '*computeState*' phase, holds there for certain time and then outputs its current power *state* to the rank *autonomic manager*.

The rank can be formally defined as a DEVS atomic model as follows:

$$M_{RDRAM} =< X, S, Y, \delta_{\text{int}}, \delta_{ext}, \lambda, t_a >$$

S: {*active, standby, suspend, offline, computeState, to_active, to_standby, to_suspend, to_offline, processing*} x R^+

X :{ $\{d \mid d \in d_{PM_{RANK}}\}$,*job_id, reqState*$_{RDRAM}$} , where *d* is a set of decisions output by the rank *autonomic manager*.

where, $d_{PM_{RDRAM}} = \{$go_active, go_standby, go_suspend, go_offline$\}$

job_id is the id of the job sent from the rank *job queue*

reqState$_{RANK}$ is a request from the rank *autonomic manager* to get the current *state*

Y: {sendState$_{\text{RANK}}$, jobAck}

$\delta_{ext} : S \times X \rightarrow S$,

1. In response to input $d \in d_{PM_{RDRAM}}$ it transitions to from the current phase *('active', 'standby', 'suspend'* or *'offline')* to the target phase (to*_active, to_standby, to_suspend, to_offline*) depending on *'d'*

2. In response to $reqState_{RANK}$ it transitions to *computeState* and then sends its current power state to the rank *autonomic manager*
3. In response to *job_id* it transitions to *to_active* (if not in the '*active*' phase) and then *processing*. At the end of *processing* phase it outputs the job id as an acknowledgement to the rank *job queue, cache* and *platform job queue.*

$\delta_{\text{int}} : S \to S,$

1. is the transition from *('to_active', 'to_standby', 'to_suspend'* or *'to_offline')* to *processing* after the transition time
2. is the transition from *processing* to *active* after time t_a which is the job processing time
3. is the transition from *computeState* to *active* after time t_a which is the time taken to compute the current power *state*.

t_a : transition time (depending on source and destination power *state*s), *computeState* time or execution time for job currently processed.

$\lambda(\text{"}computeSta\ te\text{"}, \sigma) = \text{sendState}_{\text{RANK}}$

$\lambda(\text{"}proces\,\text{sin}\ g\text{"}, \sigma) = \text{jobAck}$

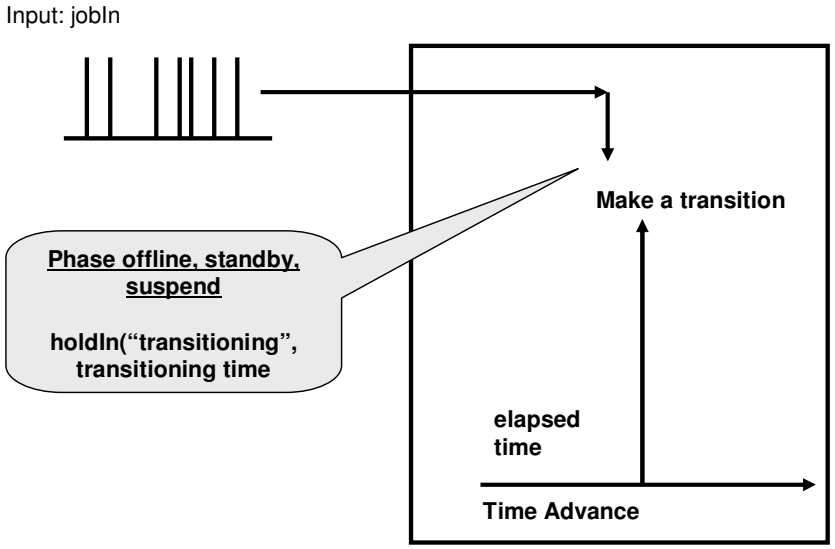

Figure 6-14: External transition

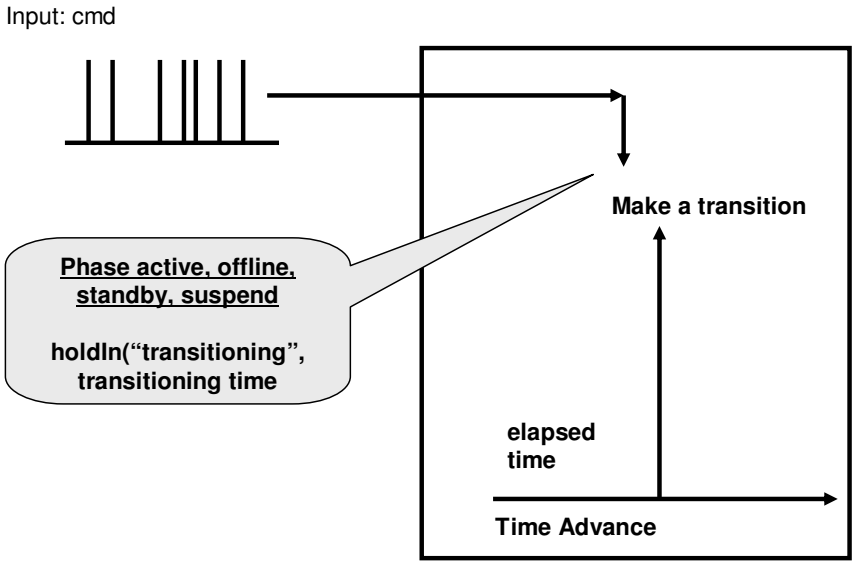

Figure 6-15: External transition

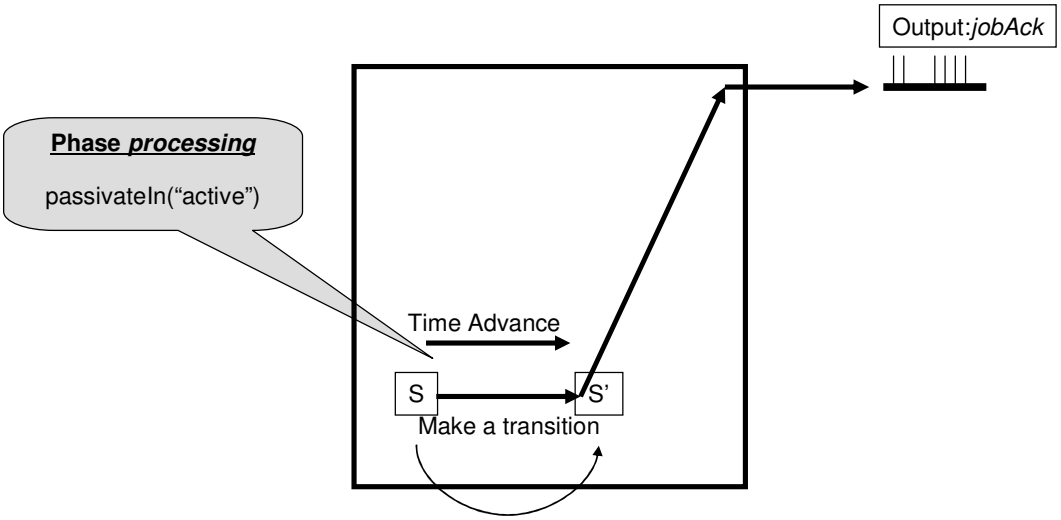

Figure 6-16: Internal Transition

6.3.1.6 Rank Job Queue Atomic Model

The rank *job queue* queues jobs to be processed by the rank. The rank *job queue* receives jobs from the *cache* and keeps forwarding them to the corresponding rank for processing. It works exactly similar to the *platform job queue*. The only difference being that it monitors the performance parameters (*requestLoss, waitTime* and *procTime*) for the rank while the *platform job queue* monitors that for the whole platform.

6.3.1.7 Core Atomic Model

The *core* is modeled similar to the rank described earlier. The main difference between the two is that while the rank successfully processes an incoming job, a *core* may require forwarding the job to the *cache* for further processing. The job may eventually get processed in the *cache* or in the rank. The *core* blocks without processing any new jobs when awaiting the arrival of acknowledgement of the forwarded job either from the *cache* or from the rank. Once the acknowledgement has arrived, the *core*

transitions from the *processing* state to the *block_on_cache* state. This scenario is shown with the aid of Figure 6-17 and Figure 6-18. Note that there are multiple ways to model the *core* when it is waiting for a cache access.

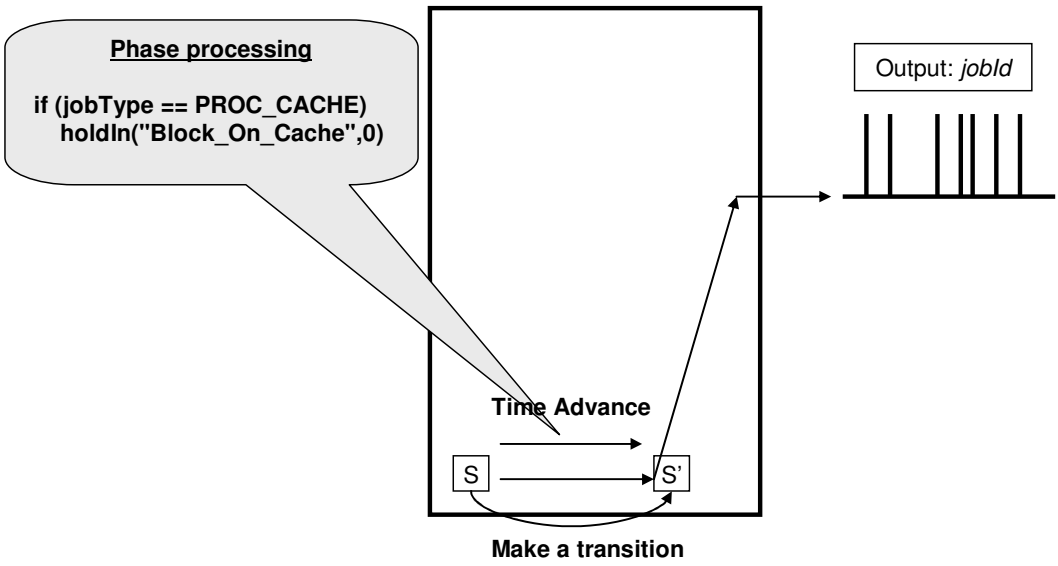

Figure 6-17: Internal transition function

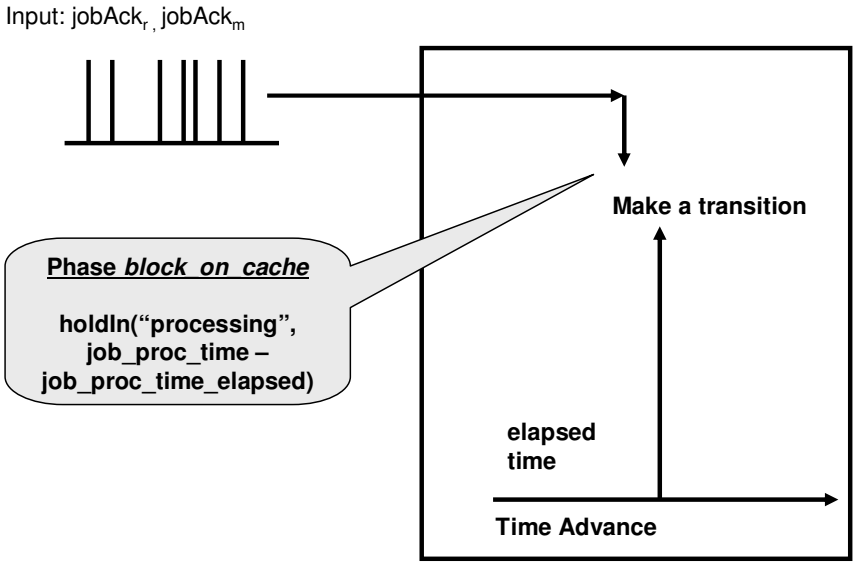

Figure 6-18: External transition function

Out of order [Hennessy et. al. 2000] *core* executions models for example, move ahead with processing new jobs instead of blocking on cache if it does not affect the results of

the computation. We have considered a simple model for ease of implementation as our focus is on power and performance management algorithms. The *core* also has different power *state*s from that of a rank and different transition time between those *state*s.

6.3.1.8 Core Job Queue

The *core job queue* behaves exactly similar to a rank *job queue*. The only difference being that the rank *job queue* receives jobs from the *cache* and forwards jobs for processing to the corresponding rank whereas the *core job queue* receives jobs from the *platform job queue* and forwards jobs for processing to the corresponding rank.

6.3.1.9 Platform Autonomic Manager Atomic Model

Once every t_p cycles, the *Platform Autonomic Manager* queries the *platform job queue* and *cache* to obtain the performance parameters for the whole platform and the memory subsystem. Simultaneously it also queries the *core flow controller* and the rank *flow controller* to obtain the current *state* that the platform is in. These parameters constitute the *ORSF* for the platform *Appflow* as discussed earlier. It uses this information to access the current platform operating point. If the platform operating point is found to out drift out of the safe operating zone, the *Platform Autonomic Manager* triggers a reconfiguration of the platform *state* from the current *state* to a destination *state* that consumes minimum power while maintaining performance. We formulate the *Platform Autonomic Manager* decision making process as a *performance-per-watt* optimization problem. It reconfigures the platform to this destination *state* with the aid of the *core* and rank *flow controllers* and the *platform job queue* and *cache* as discussed earlier.

The *Platform Autonomic Manager* is formally expressed as a *DEVS* atomic model as follows:

$$M_{PAM} =< X, S, Y, \delta_{\text{int}}, \delta_{ext}, \lambda, t_a >$$

S = {compute, idle}

X= $\{\{S_{PROCESSOR}\},\{S_{MEMORY}\},\{S_{PJQ}\},\{S_{CACHE}\}\}$

where, $S_{PROCESSOR} = \{P_1, P_2 ... P_N\}$, $S_{MEMORY} = \{M_1, M_2 ... M_N\}$, S_{PJQ} = *{reqLoss, waitTime, procTime }*, S_{CACHE} = *{reqLoss, missRatio, end-to-end delay }*

Y = *{*$reqState_{PJQ}$*,* $reqState_{CFC}$*,* $reqState_{RFC}$*,* $reqState_{Cache}$*,* $\{d \mid d \in d_{PAM}\}$*}*,where d_{PAM} belongs to the set of platform *states.*

δ_{ext} : Determines the optimal platform *state* every t_p simulation cycles. The optimal *state* is determined by solving an optimization equation, $PO_{PLATFORM}$ (given below)

$\delta_{\text{int}} : S \rightarrow S$, requests the state of the *PJQ, Cache, Processor* and *Memory* subsystem after time t_p to recompute the optimal decision.

$t_a = t_p$, where t_p is simulation cycles for which the *PAM* computes the optimal decision.

6.3.1.10 Policy Optimization $PO_{PLATFORM}$

The search for the optimal platform *state* as executed by the *Platform Autonomic Manager* is formulated as a *performance-per-watt* optimization (maximization) problem.

$$\text{Maximize } ppw_{t_i} = \frac{1}{rTime_k * e_k * x_{jk}} \text{ such that}$$

$$1.rTime_{\min} \le rTime_k \le rTime_{\max}$$

$$2.rLoss_{\min} \le rLoss_k \le rLoss_{\max}$$

$$3.n_k * s_r >= N_{ws} * s_p \qquad \text{..... (1)}$$

$$4.d_{\min} \le d_k \le d_{\max}$$

$$5.\sum_{k:1}^{N_s} x_{jk} = 1$$

$$6.\vee x_{jk} = 0 | 1$$

where, ppw_{t_i} is the *performance-per-watt* during interval t_i, $rTime_k$ is the platform *responseTime* (expressed as a sum of the *waitTime* and *procTime*) and e_k is the platform energy consumed in target *state* p_k, where, e_k is given by $e_k = \sum_{k:1}^{N_s} (c_{jk} * \tau_{trans_{jk}} + p_r * n_r * t_p + p_c * n_c * t_p) * x_{jk}$ is the sum of the transition energy consumed ($c_{jk} * \tau_{trans_{jk}}$), the energy consumed by the processor subsystem in the target *state* ($p_r * n_r * t_p$) and the energy consumed by the memory subsystem in the target *state* ($p_c * n_c * t_p$), p_r: power consumed by a memory rank in the *active state,* p_c: power consumed by a processor *core* in *turbo state* , n_r: number of memory rank*s* in *active state* , n_c: number of processor cores in *turbo state,* N_s: total number of platform *states*, c_{jk}: power consumed in *state* transition and $\tau_{trans_{jk}}$: time taken for *state* transition, $rLoss_k$ is the platform *requestLoss* in target *state* s_k, $[rTime_{\min}, rTime_{\max}]$, $[rLoss_{\min}, rLoss_{\max}]$ are the threshold *responseTime* and threshold *requestLoss* range for the platform, s_r: size per rank, n_r: number of *active* ranks in target *state*, N_{ws}: working set size (in *pages),* s_p:

size of a *page*, $[d_{\min}, d_{\max}]$ is the threshold *delay* range for the memory subsystem. x_{jk} : decision variable for transition from *state* p_j to p_k,

Constraints 1 and 2 of the optimization equation (1), state that in the target *state* the *responseTime* and the *requestLoss* must stay within the respective threshold ranges. Constraint 3 states that the target *state* should have enough memory to hold all the working set *pages* thus giving the minimum memory *miss ratio*. Constraint 4 states that in the target *state*, the *end-to-end memory access delay* should stay within the threshold range. Constraint 5 states that the optimization problem leads to only one decision. The decision variable corresponding to that is 1 and the rest are 0. Constraint 6 states that the decision variable is a 0-1 integer.

Analysis of Transition Overhead: The transition overhead $c * \tau_{trans}$ is the energy spent during *state* transition. We factor this overhead into the objective function to identify *state* transitions that would give the smallest overhead among all possible transitions. We also include the transition time τ_{trans} in calculating the platform *responseTime*. From constraint 1 of the optimization equation this prevents *state* transitions when τ_{trans} is too high. Hence this reduces the frequency of *state* transitions and thereby maintains the algorithm sensitivity to workload changes, within acceptable bounds. The transition time τ_{trans} is a sum of the rank*/core* power *state* transition time (Figure 6-4, Figure 6-5) and rank data migration time. The rank data migration process has been discussed in Chapter 5. We do not consider core process migration time at this point. Since individual cores/ranks can transition in parallel, τ_{trans} is equal to the maximum transition time among all core transitions and all rank transitions. The power consumed during transition

c is a sum of the transition power consumed by a rank*/core* (Figure 6-4, Figure 6-5) and the power consumed in the memory subsystem during data migration.

6.3.1.11 Core/Rank Autonomic Manager Atomic Model

The *Core/Rank Autonomic Managers* work very similar to the *Platform Autonomic Manager.* Once the *Platform Autonomic Manager* determines a target *state* for the platform, this decision is communicated to the *Core/Rank Autonomic Managers*. The *Core/Rank Autonomic Managers* in turn request for the *response time* and *request loss* parameters from the *core/*rank *job queues* and the current power *state* from the *core/*rank. These constitute the ORSF of the *core* and the rank *Appflows*. The decision taken by the *Platform Autonomic Manager* is further refined by the *Core/Rank Autonomic Manager.* For example if the *Platform Autonomic Manager* determines that a specific *core* is to be in *turbo state* the *Core Autonomic Manager* performs more aggressive *performance-per-watt* management by transitioning the core to an appropriate *state (turbo, eff1 or eff2)* that can still maintain the core-level performance by expending less power. The same also applies to a memory rank. We formulate the *Core/Rank Autonomic Manager* decision making process as an optimization problem similar to the *Platform Autonomic Manager.* However we can also use existing heuristic-based power management techniques at the rank and *core* level that look at the idleness between incoming requests to determine the appropriate power *state* for the *core/*rank. The DEVS atomic model for a *Rank Autonomic Manager* is presented below. The DEVS atomic model for a *Core Autonomic Manager* is similar and not discussed here.

$$M_{RAM} =< X, S, Y, \delta_{\text{int}}, \delta_{ext}, \lambda, t_a >$$

where,

S = {*compute, idle*}

$X = \{\{ S_{Rank} \}, \{ S_{RJQ} \}, d \mid d \in d_{PAM} \}$

$Y = \{reqState_{RJQ}, \quad reqState_{\text{rank}}, \quad \{d \mid d \in d_{RAM}\}\}$, where $d_{RAM} = \{do\_active, do\_nap, do\_standby, do\_pwrdown\}$

$\delta_{ext} : S \times X \rightarrow S$: Determines the optimal power *state* for the rank to transition to after every t_r simulation cycles. The optimal *state* can be determined by solving an optimization equation, PO_{RANK} (similar to $PO_{PLATFORM}$), heuristics or graph-theoretic approaches.

$\delta_{\text{int}} : S \rightarrow S$, requests the *state* of the rank *job queue* and the rank after time $t_a = t_r$ to recomputed the optimal decision.

$t_a = t_r$, where t_r is simulation cycles for which the *Rank Autonomic Manager* computes the optimal decision

6.3.1.12 PMRank Coupled Model

The *PMRank* coupled model consists of the rank, rank *job queue* and rank *autonomic manager* atomic models. For sake of clarity this is not shown in Figure 6-8.

The DEVS formalism for the power and performance managed *PMRank* coupled model is as follows:

$$N_{PMRank} = (X, Y, D, \{M_d \mid d \in D\}, EIC, EOC, IC, Select)$$

Where,

InPorts = {"inJob"}

X_{inJob} = a Pair of (jobId, rankJobQueueId)

X = {(“inJobId”, v)| v ∈ X_{inJob}}

OutPorts = {”jobAck”}

Y_{jobAck} = a Pair of (jobId, rankJobQueueId),

where,

- jobId is the ID of the job being acknowledged

- rankJobQueueId is the ID of the rank *job queue/*rank that sends the acknowledgement

Y = {(“jobAck”, v)| v ∈ Y_{jobAck}}

D = {RankJobQueue, Rank, RankPowerManager}

$M_{JQ = RankJobQueue}$, $M_{RDRAM = Rank}$, $M_{PM = RankPowerManager}$

EIC = {(N,”inJob”),(“JQ”,inJobId)}

EOC = {(Rank,“out”),(N,”jobAck”)}

IC = {((RankJobQueue,”outJobId”) , (Rank,”inJobId”)) ,

((Rank, “out”), (RankJobQueue,”jobAck”)),

((RankJobQueue,”outState”) , (RankPowerManager,“JQState”)) ,

((Rank,”outState”) , (RankPowerManager,”MEState”))

((RankPowerManager,”reqState”) , (Rank,”reqState”)) ,

((RankPowerManager,”reqState”) , (RankJobQueue,”reqState”))}

((RankPowerManager,”decision”) , (Rank,”cmd”))}

6.3.1.13 Pmemory controllerore Coupled Model

The *PMCore* coupled model consists of the *core, core job queue* and *core autonomic manager* atomic models. For sake of clarity this is not shown in Figure 6-8. The behavior and modeling of these atomic models is very similar to their counterparts in the *PMRank* coupled model. Hence we do not go into the details of these models here.

6.3.1.14 PMPlatform Coupled Model

The *PMPlatform* coupled model consists of *platform job queue, pmcore, pm*rank*, cache, core flow controller,* rank *flow controller* and *platform autonomic manager* models. It models the entire server platform.

6.4 Simulation Approach and Results

For our experimental study, we modeled a platform with 4 cores, 2 cores per processor and 4 memory ranks with 2 ranks per branch. We validated our technique with a synthetic memory intensive workload, processor intensive workload and a workload that has a mix of processor-intensive and memory-intensive phases. Since we model a web server platform for this work, we use a random workload generator that is poisson distributed [Garcia1994]. A poisson distribution can characterize most real-life web server workloads where the inter-arrival times between successive requests/jobs are exponentially distributed. We repeated the simulation for five different sample sizes – 5000, 6000, 8000, 9000 and 10,000 simulation cycles, in order to study the variability of the results (platform energy savings) from one case to another. We noticed that the larger the simulation duration, the smaller the variability in the results between repeated simulations. In what follows, we present the results for 10,000 simulation cycles.

6.4.1 Discussions on Workload Generation Process

We used synthetic workloads to validate our technique. This workload is generated by the *Service Requester* model discussed earlier. Each job generated by the *Service Requester* has the following fields

```
this.jobId = int jobId;
this.waitTime = float waitTime;
this.procTime = float procTime;
this.genId = int genId;   /* Coming from */
this.src_coreId = int coreId; /* Coming from – core ID */
this.cache_references = int cache_ref;
this.cache_miss [] = int [] cache_miss;
this.memory_misses = int memory_misses;
this.PFN = int PFN;
```

Each job may lead to a certain number of cache references and each cache reference may in turn lead to a cache hit (no memory reference) or cache miss (memory reference). This is captured by the cache_miss attribute in the *job* class as shown above. It is an integer array of size equal to the number of cache references with a 0/1 entry in corresponding array index for each cache reference. Hence, to generate memory intensive workload, we initialize the array with more '1' than '0' entries leading to more cache misses and hence more memory accesses. This is shown in Figure 6-19.

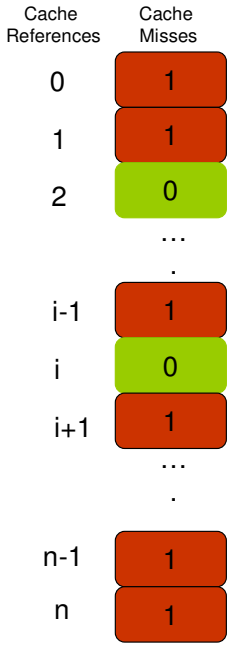

Figure 6-19: Memory-intensive workload

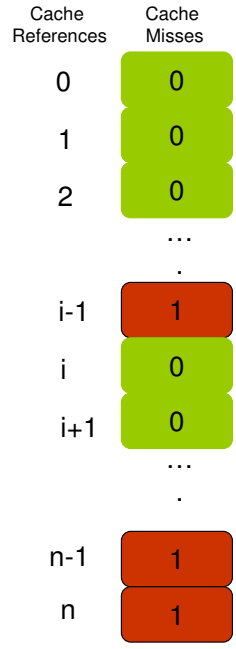

Figure 6-20: Processor-intensive workload

To generate processor intensive workload we initialize the array with more '0' than '1'. This is shown in Figure 6-20. For mixed workload we had a random temporal mix of jobs of both types. We used a random job inter-arrival time for our validations and hence the workload does not follow any perceivable patterns. This is often harder to make predict in terms of when and how much platform reconfiguration is required in order to reduce the platform energy savings while maintaining performance.

6.4.2 Validation of Technique for Memory-intensive Workload

As shown in Figure 6-21, the *average wait time* experienced by a platform job is nearly negligible at the core and cache-level compared to that at the memory-level. While the *average wait time* at the memory level follows the arrival rate of incoming jobs, increasing as the arrival rate increases and vice-versa, around 2000 simulation cycles the *wait time* increases even when the arrival rate drops. This is due to the fact that the platform memory was initially under-configured to handle the incoming traffic.

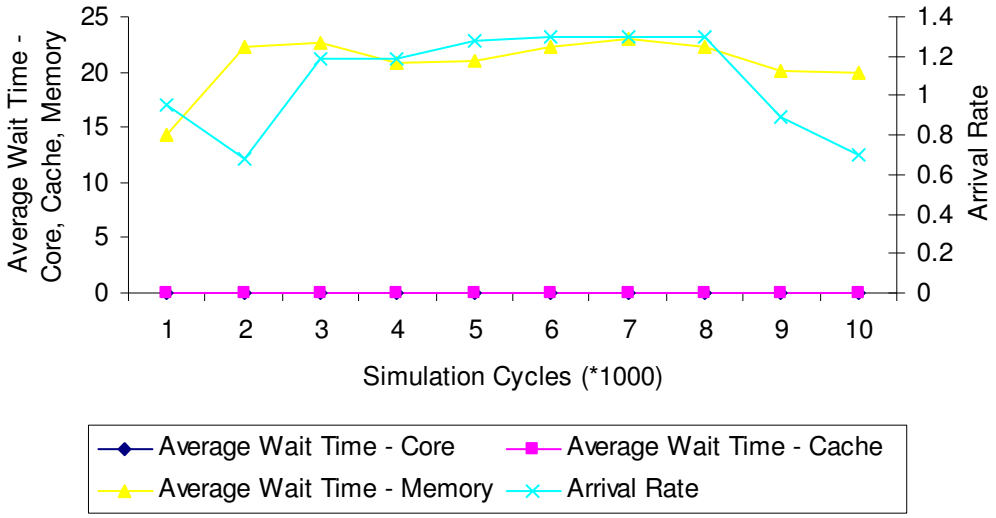

Figure 6-21: Behavior of incoming traffic

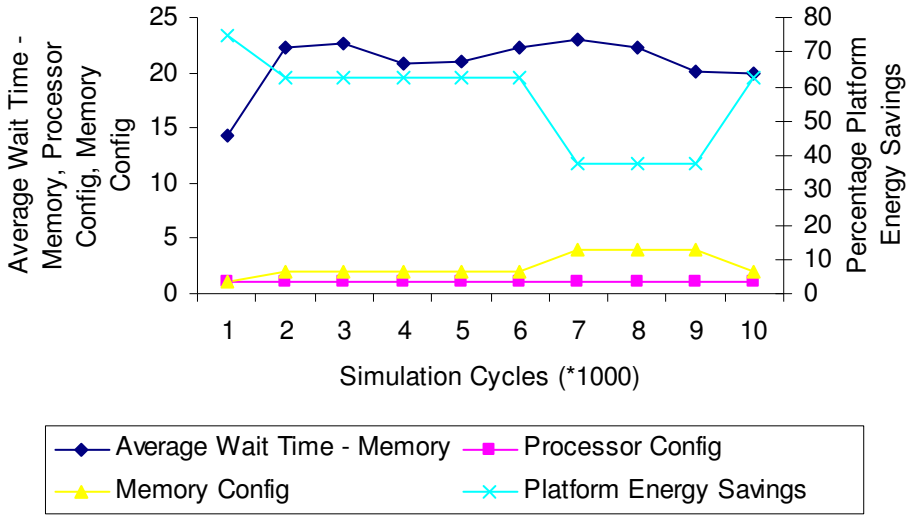

Figure 6-22: Platform energy savings

As can be seen from Figure 6-22, around 2000 simulation cycles, the platform memory configuration is increased from 1 to 2 ranks hence bringing down the *average wait time*. Our scheme gives an average energy savings of 56.25%.

6.4.3 Validation of Technique for Processor-intensive Workload

Initially the platform is configured at 2 processor cores and 2 memory ranks. As can be seen from Figure 6-23, the job arrival rate starts increasing from around 4000 simulation cycles and this increases the *average wait time* experienced by the jobs. From Figure 6-23 it is clear that the *average wait time* experienced by the jobs is significantly higher in the processor subsystem than in the cache or the memory subsystem.

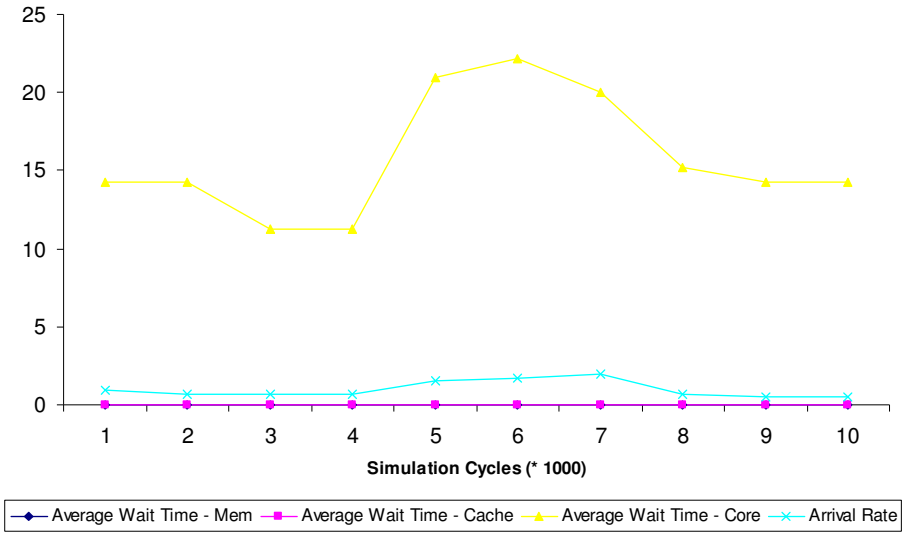

Figure 6-23: Behavior of incoming traffic

In order to avert this increase in the *average wait time*, the *Platform Autonomic Manager*, scales up the processor subsystem from 1 core at 5000 simulation cycles to 2 cores at 6000 simulation cycles and then to 3 cores at 7000 simulation cycles, bringing it down to 1 core again at 9000 simulation cycles when the rate of arrival drops again. This is shown in Figure 6-24. Note that the configuration of the memory subsystem remains more or less static dropping from 2 ranks to 1 rank around 3000 simulation cycles and remains there once the *Platform Autonomic Manager* establishes that the memory subsystem does not have much a role to play in the increase of the *average wait time* as seen by platform jobs. Figure 6-24 also shows the ensuing platform energy savings with an average energy savings of 63.75%. Note that there is a distinctive phase-lag between the monitored *average wait time*-core and the core configuration. This phase lag occurs due to the fact that the *Platform Autonomic Manager* has room for improvement in terms of predicting the arrival rate of the incoming workload. If the *Platform Autonomic Manager* were to accurately predict the arrival rate, it could configure the processor

subsystem proactively even before it starts seeing a real increase in the arrival rate. However in this case, we notice the *Platform Autonomic Manager* is almost one observation cycle behind in terms of predicting and appropriately configuring the processor subsystem. We are investigating this issue further.

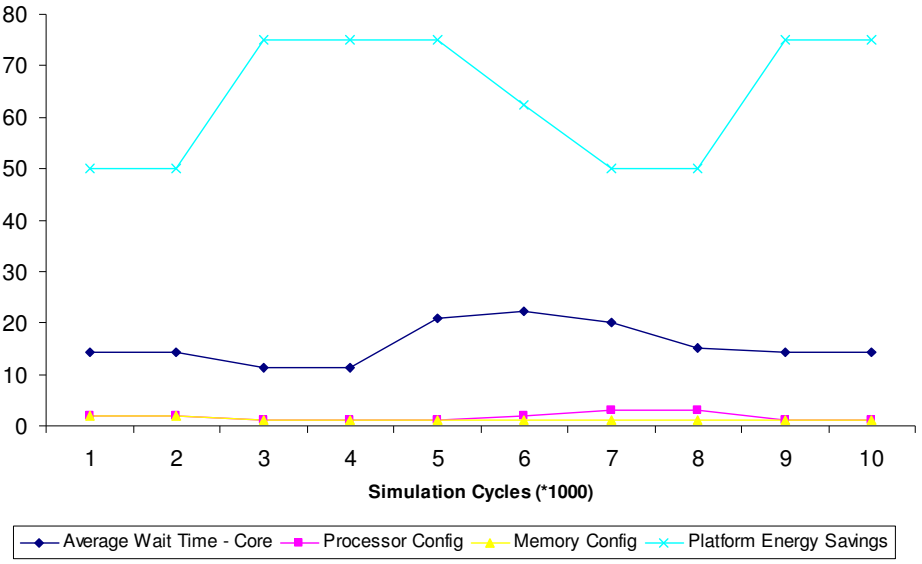

Figure 6-24: Platform energy savings

6.4.4 Validation of Technique for Mixed Workload

Figure 6-25 shows a mixed workload that consists of both processor-intensive and memory-intensive phases. As expected, the workload impacts the *average wait time* experienced by jobs at the processor subsystem different from that experienced by the jobs at the memory subsystem. The request arrival rate increases around 3000 simulation cycles but the *average wait time* – memory increases around 2000 simulation cycles and the *average wait time* – processor increases around 5000 simulation cycles. This is because the memory was initially configured at 1 rank and the processor was configured 2 cores. The initial memory configuration was too small to handle the incoming traffic

causing the *Platform Autonomic Manager* to scale-out the size of the memory subsystem to 2 ranks and eventually to 4 ranks around 7000 simulation cycles. This is shown in Figure 6-26. At that time the memory subsystem was configured to its maximum capacity leading to zero savings in memory energy. This reconfiguration in the memory subsystem is able to being down the *average wait time* experienced by jobs in the memory subsystem.

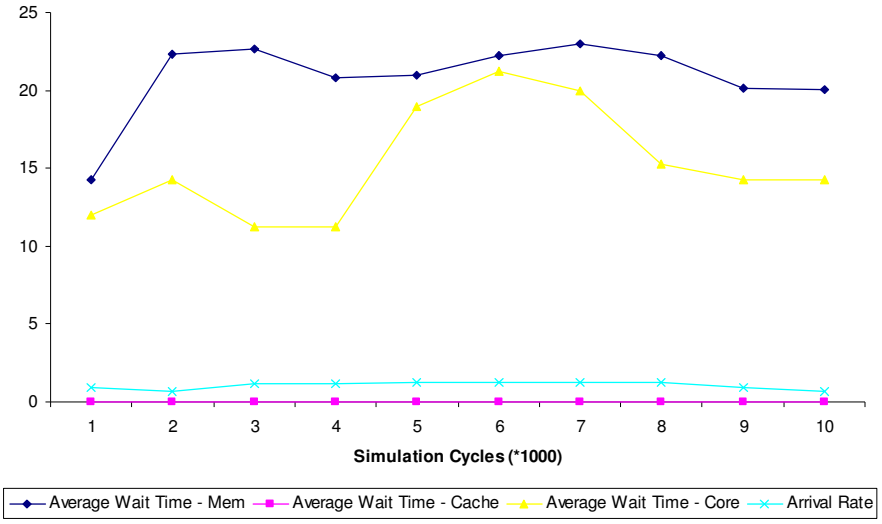

Figure 6-25: Behavior of incoming traffic

For the processor subsystem however, the *average wait time* starts increasing only around 5000 simulation cycles. It is around this time, that the workload changes phase into becoming more processor-intensive and the processor subsystem needs to scale-out to a bigger capacity to handle the workload. Hence the *Platform Autonomic Manager* reconfigures the processor subsystem from 1 core to 2 cores and 3 cores in the subsequent cycle. This is shown in Figure 6-27. This brings down the *average wait time* experienced by jobs in the processor subsystem. In this manner, the *Platform Autonomic Manager* reconfigures that subsection of the platform to an optimal configuration that has a direct impact on the overall platform response time thereby saving platform energy and

maintaining platform performance.

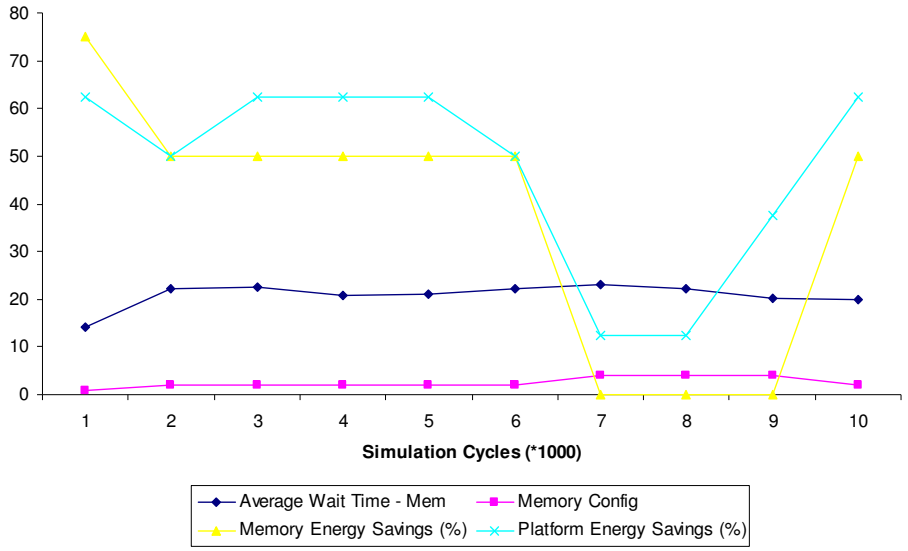

Figure 6-26: Memory energy savings

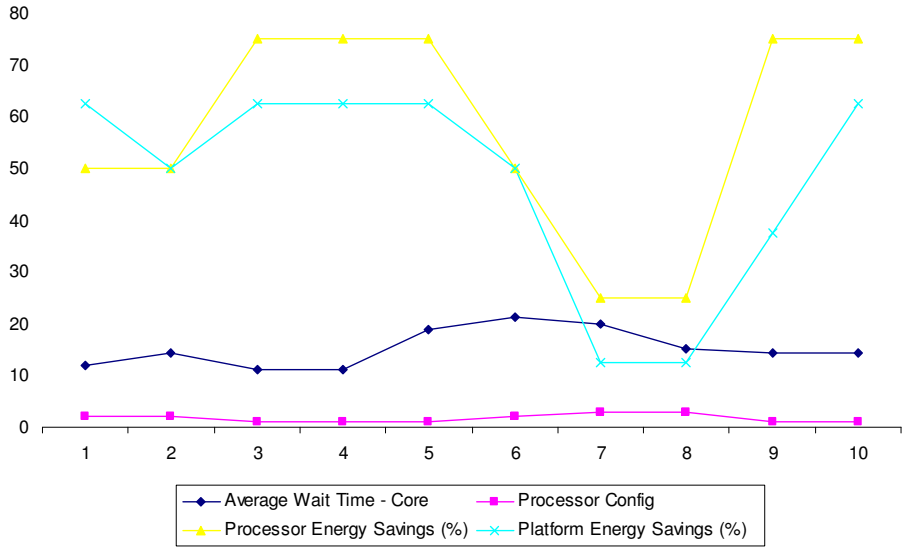

Figure 6-27: Processor energy savings

Figure 6-28 shows a comparison of the energy savings obtained from the processor subsystem and the memory subsystem throughout the length of the simulation. Note that there is zero energy savings in the memory subsystem from simulation cycles 7000 to 9000 because the memory subsystem is operating at its maximum capacity during this phase.

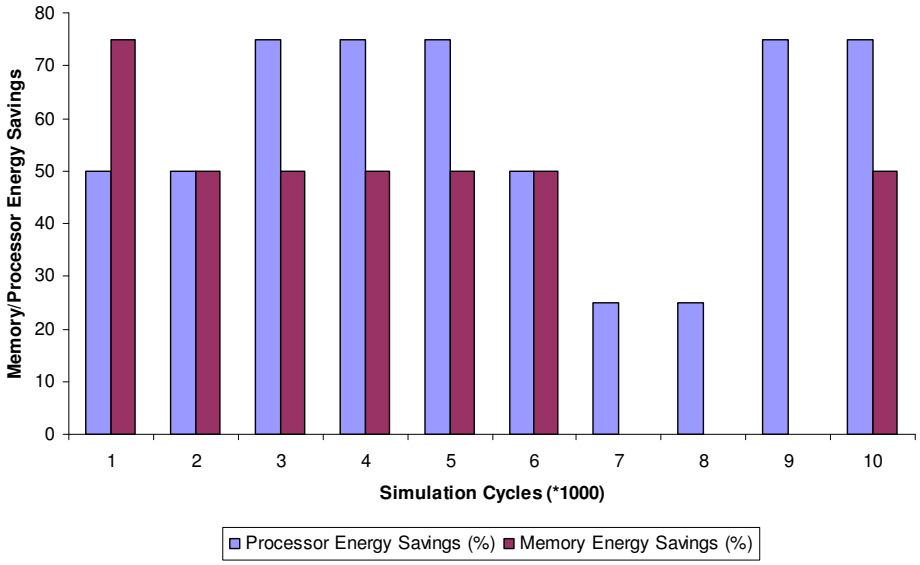

Figure 6-28: Comparison of processor and memory energy savings

Our technique gives an overall average platform energy savings of 47.5% while always maintaining the platform response time within the set threshold value. Next, let us study the impact of changing the threshold value of the platform performance parameter on the platform energy savings.

6.4.5 QoS trade-offs for Platform Energy Savings

We varied the threshold value of the end-to-end *delay* for the memory subsystem and measured the impact of that change on the memory energy savings for the same workload. As expected, for a more stringent threshold value the opportunity for energy savings is very little. For example, an increase in the threshold value by just 33.3% increases the savings by 37.5% and with an increase in threshold by 66.7% the energy savings increases by 62.5%. This is with respect to a threshold *delay* of 15 simulation cycles. This is shown in Figure 6-29. This demonstrates the opportunity of increased power savings by degrading performance within threshold values.

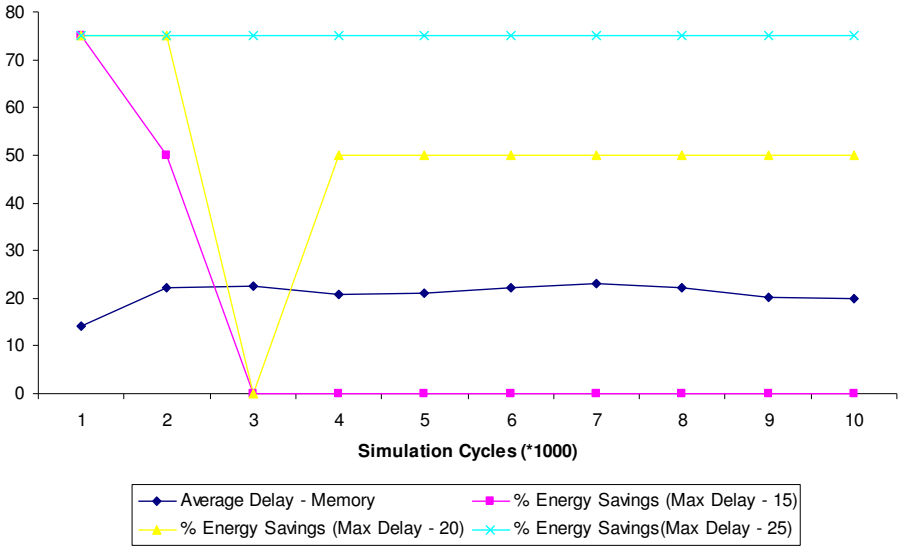

Figure 6-29: QoS trade-offs for energy savings

6.4.6 Analysis of Energy Savings with Hierarchical Power Management

Figure 6-30 shows the results of the global *Platform Autonomic Manager* working in tandem with the local *Rank Autonomic Manager.*

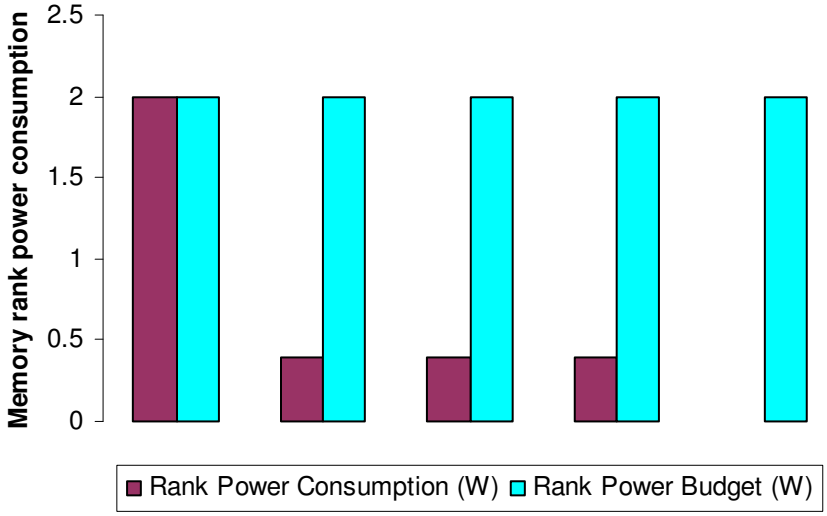

Figure 6-30 Platform memory power budget vS rank power consumption

Based on the incoming workload, the *Platform Autonomic Manager* determines a target power *state* of *standby* (2 Watt) for the memory rank in order to save power while maintaining the platform response time.

This is indicated in the Figure by the rank power consumption. However, the *Rank Autonomic Manager* performs its own optimizations to reduce the actual rank power consumption to 0.3 Wattt (*suspend state*) and finally to 0 Watt (*offline state*). Similarly the *Core Autonomic Manager* performs its local optimizations to perform aggressive power management for the individual core.

6.5 Conclusion

In this paper, we presented a scheme for autonomic power and performance management of high-performance server platform with multi-core processors and multi rank memory subsystem. Our experimental results show around 56.25% platform energy savings for memory-intensive workload, 63.75% platform energy savings for processor-intensive workload and 47.5% platform energy savings for mixed workload.

7 CONCLUSION AND FUTURE RESEARCH DIRECTIONS

7.1 Summary

In this dissertation, we developed a theoretical and experimental framework and general methodology for autonomic power and performance management of high-performance server platforms to achieve (a) online modeling, monitoring, and analysis of power consumption and performance; (b) automatic detection of application execution phases and properties for a wide range of workloads and applications;(c) employing knowledge of application execution phases to dynamically identify strategies that minimize power consumption while maintaining the required quality of service (c) dynamically reconfigure high-performance servers in data centers according to the selected optimization strategies; and (d) effective optimization techniques that can scale well and can perform in a dynamic and continuous fashion, making the system truly autonomic. The main research activities of this thesis can be highlighted as follows:

1. *Modeling, and application, of autonomic computing system based on Ashby's Ultra-stable System* model of the human brain. We first modeled an archetypal autonomic computing system based on Ashby's Ultra-stable System in his study of the human brain. We then applied this archetypal model to lay-out the high-level architectural blue-print of a self-managing data center entity that manages its energy consumption and performance by proactively adapting to changes in the incoming workload.

2. *Hierarchical autonomic power and performance management.* Our autonomic data center was built using smaller and smaller self-managing entities modeled after the

same archetypal autonomic computing system. We called these autonomic entities the building blocks of an autonomic data center. The autonomic data center consisted of multiple autonomic clusters that self-managed their power and performance by adapting to the cluster workload. An autonomic cluster in turn consisted of multiple autonomic servers where each individual server self-managed its power and performance by adapting to the server workload. Similarly, an autonomic server consisted of multiple autonomic subsystems – CPU, memory, I/O each designed as an autonomic entity that adapts to its own workload to self-manage its power and performance. This design is meant to demonstrate that while each individual entity within a data center is self-managing its own power and performance, they also co-operate and co-ordinate amongst themselves and with the upper and lower-level hierarchies to work towards the overall objective of reducing energy consumption for the whole data center while maintaining performance. Power management decisions from the upper-level autonomic entities are further refined by the lower-level entities and feedback from the lower-level entities flow up to the upper-level entities to refine their decisions. This hierarchical management scheme is a manifestation of the local and global control loops of our archetypal autonomic computing system.

3. *Characterize dynamic (spatial and temporal) behaviors and resource requirements of applications*. We devised an Application Flow (*Appflow*) methodology that characterizes the spatial and temporal behaviors and resource requirements of applications at runtime. An autonomic entity at any hierarchy uses this *Appflow* to develop the clairvoyance required to proactively adapt to changes in the incoming workload. Each autonomic entity has its own *Appflow*. We introduced the *Operating Region Spatial Features* within

the *Appflow* that gracefully capture the notion of essential variables in Ashby's Ultra-stable System managing which within bounds ensures the "survivability" of the organism. In our case "survivability" translates to power and performance management. It is the task of the autonomic entity to manage its *Operating Region Spatial Features* within threshold bounds to self-manage its power and performance.

4. *Power and performance management algorithms.* As part of this thesis, we focused on the bottom two hierarchies of a data center to apply our concepts discussed earlier. We devised and validated algorithms for autonomic power and performance management of an interleaved memory subsystem in a high-performance server platform. We then built on that work to encompass power and performance management of a server platform that consisted of multi-core processors and the multi-rank memory subsystem discussed earlier.

   4.1. **PERFWATT***: Adaptive interleaving technique for memory power and performance management.* Our power and performance management technique addressed interleaved memory subsystems where existing power management techniques cannot be applied. We dynamically interleaved data across selected memory modules to adapt to the incoming workload's memory requirements. The goal is to increase the idleness of the remaining modules allowing them to transition to low-power *state*s and remain in those *state*s for as long as the performance remains within given acceptable thresholds. This delivers roughly the same performance, but with considerably less energy consumption consequently maximizing the platform's power and performance.

4.1.1. *The energy consumed by memory is minimized subject to application's memory requirements and end-to-end memory access delay constraints.* We formulate this as a *performance-per-watt* optimization problem and solve it using an analytical memory power and performance model. We model the memory subsystem as a set of *states* and *transitions*. A *state* represents a specific memory configuration supported by the memory subsystem. It is defined by fixed base *power* consumption and variable end-to-end memory access *delay*. Whenever, the application's memory requirements change and/or the *delay* exceeds a threshold value, a Data Migration Manager (DMM) in the Memory Controller (memory controller) searches for a target *state* among all possible *states* that consume the minimum power while satisfying the application's memory requirement and the end-to-end *delay* constraints. This target *state* is obtained as a solution of the *performance-per-watt* optimization problem and is the "optimal" solution for that platform. The Data Migration Manager uses the memory *Appflow* to determined when to trigger a reconfiguration of the memory subsystem in order to reduce power consumption while maintaining performance.

4.1.2. *Application and OS independence.* Our technique is application and OS independent. We reconfigure the degree of interleaving by migrating cache lines that belong to *working set pages* onto selected memory modules in the target *state.* Hence this technique deals with a very fine granularity of migration that does not require OS intervention.

4.1.3. *Exploits the architectural characteristics of the memory subsystem.* Our scheme exploits the architectural characteristics of the memory subsystem in doing the data migrations. This helps maximize the platform's *performance-per-watt* effectively.

4.1.4. *Leverages existing fine-grained power management schemes.* The proposed technique can be coupled with any suitable fine-grain power management technique. Our *Data Migration Manager* that implements the **PERFWATT** algorithm for memory power and performance management works in collaboration with local *Power Managers* one per memory module that can implement any existing fine-grained power management techniques to transition memory modules to low-power *state*s based on their length of idleness.

We validated our approach both on a real server as-well-as on a trace-driven memory system simulator called DRAMSim. We configured the memory simulator to closely simulate the memory subsystem in our server unit. Our test-bed server included 2 dual-core Intel™ Xeon processors, 5000P Memory Controller Hub and 8 GB DDR2 FBDIMM memory. Its memory subsystem consisted of two branch*es,* two channel*s* per branch, two FBDIMMs per channel and two rank*s* per FBDIMM. The server could support a total of 8 FBDIMMs or 16 rank*s* in total. We studied the *performance-per-watt* management for SPECjbb2005 benchmark on our server unit. Our approach gave an energy saving of about 48.8 % (26.7 kJ) compared to traditional power management techniques measured at 4.5%. It gave a transition overhead of about 18.6ms

leading to energy saving of 1.44kJ per millisecond of transition overhead time and a maximum *performance-per-watt* improvement of 88.48%. The complexity of the algorithm was observed to be small for smaller memory sizes; however it increased exponentially when the number of memory ranks reached around 128 rank*s.*

For our experiments in simulation, we used ***gcc*** traces to evaluate our algorithm in the DRAMsim simulator. We simulated DDR FBDIMM memory and used a close-page row buffer management policy to emulate a fully-interleaved memory similar to that used in our server unit. We configured the memory with 4 branch*es* and 4 rank*s* per branch as the maximum configuration supported by the platform. The simulator showed 89.7% improvement in *performance-per-watt* compared to the best performing traditional technique (ISB).

4.2. *Extension of our technique for platform-wide power and performance management.* We introduced a hierarchical management framework where the platform modeled as a collection of networked devices (processors and memory) was managed by an upper-level *autonomic manager* and the individual devices were managed by associated lower-level *autonomic managers.* With the aid of this work, we demonstrated the co-ordination between autonomic entities to work towards a common management objective. At each hierarchy we formulated the problem as a power and performance optimization problem. Solutions at the upper-level feed into the solutions at the

lower-levels and feedback from the lower-levels feed into the solutions at the upper-level. The platform-level *autonomic manager* utilized a platform *Appflow* to determine when to trigger platform reconfigurations.

We modeled the platform (with multi-core processors and multi-rank memory subsystem) using the DEVS modeling and simulation framework. The memory subsystem modeled was very close to the memory subsystem in our experimental server unit that was used to validate our memory power management algorithms. In our experimental study, we modeled a platform with 4 cores, 2 cores per processor and 4 memory ranks with 2 ranks per branch. We validated our technique with a synthetic memory intensive workload, processor intensive workload and a workload that had a mix of processor-intensive and memory-intensive phases. Our experimental results showed around 56.25% platform energy savings for memory-intensive workload, 63.75% platform energy savings for processor-intensive workload and 47.5% platform energy savings for mixed workload.

7.2 Contributions

The main contributions of this work can be summarized as follows:

1. Simultaneous management of multiple objectives in a unified framework in this case demonstrated with the simultaneous management of power & performance.
2. Hierarchical modeling of autonomic power and performance management. This approach ensures scalability of solutions and feedback driven decision-making.

3. Introduction of a novel technique called *Appflow* that characterizes the spatial and temporal behavior and dynamic resource requirements of applications.
4. *Appflow* also provides a coherent and consistent mechanism to capture management objectives while traversing across hierarchies of management without loss of information.
5. Using optimization techniques for runtime identification of appropriate power/performance management strategies that reconfigure the data center server platforms to adapt to the incoming workload.
6. An adaptive interleaving algorithm that is application and OS independent, for power and performance of memory subsystem management.
7. Holistic power management of a server platform by collaborative power management of individual platform subsystems.
8. Power and performance models for fully-buffered DIMMs and multi-core processors.

7.3 Future Research Directions

This work is guided by two research visions. The first is to address the problem of power management in computing systems and the second to develop a unified framework and for management of multiple objectives such as power, performance, fault, security etc. Towards that effect, we have demonstrated the simultaneous management of power & performance for large-scale systems as part of this work. Going forward, we would like to incorporate other management objectives into the framework. This is a highly challenging task given the heterogeneity and dynamism of today's platforms but there is

work already underway that we can leverage on and tie it together in one unified framework.

We would like to expand in breadth and depth into the power management work itself by expanding our work to include other platform components like the NIC card, hard-disks, power supply, fans. We would also extend the work to upper hierarchies within a data center and to geographically dispersed data centers. We would also like to incorporate the thermal element into our power management techniques without which data center power management schemes are quite incomplete. In addition, we would also like to develop accurate and complete power and performance models that can provide a harness for other researchers to experiment their work on.

Our algorithms proposed as part of this work as based on optimization approach. We would however like to experiment with other emerging approaches such as Game Theory and light-weight machine learning techniques that would make our algorithms more suitable for real-time use by keeping the complexity in check even when the sizes scale up.

In this work, we focused on single applications for our validations. In the future we would like to extend our *Appflow* methodology to characterize multiple application flows and experiment with multitude of real data center workloads mimicking more realistic scenarios.

REFERENCES

1. [Egenera2003] Egenera (2003), Improving Datacenter Performance, *White Paper*, Retrieved on August 8, 2007 from http://www.egenera.com/reg/goog/goog_dc_ty.html, Egenera: The Datacenter Virtualization Company.

2. [Mitchell-Jackson2001] Mitchell-Jackson J. (2001), Energy Needs In An Internet Economy: A Closer Look At Data Centers, *MS Thesis*, July 2001.

3. [EPA2007] EPA (2007), EPA Report to Congress on Server and Data Center Energy Efficiency, *Technical Report*, Retrieved on August 29, 2007 from http://www.energystar.gov/ia/partners/prod_development/downloads/EPA_Report_Exec_Summary_Final.pdf.

4. [APC2005] APC (2005), Determining Total Cost of Ownership for Data Center and Network Room Infrastructure, *White Paper CMRP-5T9PQG_R3_EN*, http://www.apcmedia.com/salestools/CMRP-5T9PQG_R3_EN.pdf

5. [Chase et. al. 2001] Chase J. and Doyle R., Balance of Power: Energy Management for Server Clusters, *Proceedings of the 8th Workshop on Hot Topics in Operating Systems* (*HotOS-VIII*), May 2001, pp. 163–165.

6. [White et. al. 2004] White R. and Abels T., "Energy Resource Management in the Virtual Data Center", *IEEE International Symposium on Electronics and the Environment*. May, 2004, pp. 112-116.

7. [Zeigler2000] Zeigler B. P., Praehofer H. and Kim T. G., *Theory of modeling and simulation*. 2nd ed. New York Academic Press, 2000.

8. [SPECjbb2005] SPECjbb2005, SPECjbb2005: A new Java Server Benchmark, Retrieved on Aug 8, 2007 from http://www.spec.org/jbb2005/docs/WhitePaper.html. 2006.

9. [Wang et. al. 2005] Wang D., Ganesh B., Tuaycharoen N., Baynes K., Jaleel A. and Jacob B., "DRAMsim: A memory-system simulator", *SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 100-107, September 2005.

10. [Miyoshi et. al. 2002] Miyoshi A., Lefurgy C., Van Hensbergen E., Rajamony R., Rajkumar R., Critical power slope: understanding the runtime effects of frequency scaling, *Proceedings of the 16th International Conference on Supercomputing*, pp. pp. 35 – 44, June 22-26, 2002, New York, USA.

11. [Shin et. al. 2004] Shin H. and Lee J., Application Specific and Automatic Power Management based on Whole Program Analysis, *Final Report,* http://cslab.snu.ac.kr/~egger/apm/final-report.pdf, August 20, 2004.

12. [Elnozahy et. al. 2002] Elnozahy E. N., Kistler M., Rajamony R., Energy-Efficient Server Clusters. *In Proceedings of the 2nd Workshop on Power-Aware Computing Systems*, February 2002.

13. [Pinheiro et. al. 2001] Pinheiro E., Bianchini R., Carrera E. V. and Heath T., Load Balancing and Unbalancing for Power and Performance in Cluster-Based Systems, *Proceedings of the Workshop on Compilers and Operating Systems for Low Power*, September 2001.

14. [Sharma et. al. 2003] Sharma V., Thomas A., Abdelzaher T., Skadron K. and Lu Z., Power-aware QoS Management in Web Servers, *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, pp. 63, December 03-05, 2003.

15. [Abdelzaher et. al. 2003] Abdelzaher T. and Sharma V., A synthetic utilization bound for a-periodic tasks with resource requirements, *In Euromicro Conference on Real Time Systems*, Porto, Portugal, pp. 141- 150, July 2003.

16. [Abdelzaher et. al. 2001] Abdelzaher T. F. and Lu C., Schedulability analysis and utilization bounds for highly scalable real-time services, *In IEEE Real-Time Technology and Applications Symposium*, Taipei, Taiwan, June 2001.

17. [Lebeck et. al. 2000] Lebeck A. R., Fan X., Zeng H., and Ellis C., Power Aware Page Allocation, *In ASPLOS*, pp. 105-116, 2000.

18. [PC Guide2001] The PC Guide 2001, Hard Disk Power Down Timeout, Retrieved in 2005 from http://www.pcguide.com/ref/mbsys/bios/set/pmDisk-c.html, April 2001.

19. [Douglis et. at. 1994] Douglis F., Krishnan P. and Marsh B., Thwarting the Power Hungry Disk, *In Proceedings of the 1994 Winter USENIX Conference*, San Francisco, pp. 23 - 23, January 1994.

20. [Cai et. al. 2005] Cai, L., Yung L., Joint Power Management of Memory and Disk, *Proceedings of the conference on Design, Automation and Test in Europe*, pp. 86-91, March 07-11, 2005.

21. [Felter et. al. 2005] Felter W., Rajamani K., Keller T. and Rusu C., A Performance-Conserving Approach for Reducing Peak Power Consumption in Server Systems, *ACM International Conference on Supercomputing (ICS)*, Cambridge, MA, pp. 293 - 302, June 2005.

22. [Bohrer et. al. 2002] Bohrer P., Elnozahy E., Keller T., Kistler M., Lefurgy C., McDowell C. and Rajamony R., The case for power management in web servers, *Power Aware Computing*, Chapter 14, 2002. Klewer Academic Publishers.

23. [Elnozahy et. al. 2003] Elnozahy M., Kistler M., and Rajamony R, Energy Conservation Policies for Web Servers. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, March 2003.

24. [Zhu et. al. 2004] Zhu Q., David F. M., Devaraj C., Li Z., Zhou Y. and Cao P., Reducing Energy Consumption of Disk Storage Using Power-Aware Cache Management, *In Proceedings of the International Symposium on High Performance Computer Architecture*, pp. 118-129, 2004.

25. [Tiwari et. al.1994] Tiwari V., Malik S., and Wolfe A., Power Analysis of Embedded Software: A First Step towards Software Minimization, *IEEE Transactions on Very Large Scale Integration*, vol. 2(4) pp. 437-445, December 1994.

26. [Gurumurthi et. al. 2002] Gurumurthi S., Sivasubramaniam A., Irwin M.J., Vijaykrishnan N., Kandemir M., Li T. and John L. K., Using Complete Machine Simulation for Software Power Estimation: The SoftWatt Approach, *In Proceedings of the International Symposium on High Performance Computer Architecture (HPCA-8)*, Cambridge, MA, pp. 141-150, February 2-6, 2002.

27. [Rong et. al. 2005] Rong P. and Pedram M., Hierarchical Power Management with Application to Scheduling, *ISLPED (International Symposium on Low Power Electronics and Design)*, pp. 269 - 274, 2005.

28. [Srivastava et. al.1996] Srivastava M., Chandrakasan A. and Brodersen R., Predictive system shutdown and other architectural techniques for energy efficient programmable computation, *IEEE Trans. On VLSI Systems*, Vol. 4, pp. 42–55, Mar. 1996.

29. [Hwang et. al.1997] Hwang C. H. and Wu A., A predictive system shutdown method for energy saving of event-driven computation, *International Conference on Computer-Aided Design*, pp. 28–32, Nov. 1997.

30. [Hsu et. al. 2003] Hsu C. and Kremer U., The Design, Implementation, and Evaluation of a Compiler Algorithm for CPU Energy Reduction, *ACM SIGPLAN 2003 conference on Programming Language Design and Implementation*, San Diego, CA, pp. 38-48, June 2003.

31. [Weiser et. al. 1994] Weiser M., Welch B., Demers A. and Shenker S., Scheduling for reduced CPU energy. *In First Symposium on Operating Systems Design and Implementation*, pp. 13-23, Monterey, California, U.S, 1994.

32. [Delaluz et. al. 2001] Delaluz V., Kandemir M., Vijaykrishnan N., Sivasubramaniam A. and Irwin M. J., Hardware and Software Techniques for Controlling DRAM Power Modes, *IEEE Trans. Computers*, vol. 50, no. 11, pp. 1154-1173, Nov. 2001.

33. [Fan et. al. 2001] Fan X., Ellis C. and Lebeck A. R., Memory controller policies for dram power management, *Proc. International Symposium on Low Power Electronics and Design*, pp. 129-134, Aug. 2001, Huntington Beach, CA, United States.

34. [Zhou et. al. 2004] Zhou P., Pandey V., Sundaresan J., Raghuraman A., Zhou Y. and Kumar S., Dynamic tracking of page miss ratio curve for memory management, *Proc. Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 177-188, Oct. 2004, Boston, MA, USA.

35. [De La Luz et. al. 2002] De La Luz V., Sivasubramaniam A., Kandemir M., Vijaykrishnan N. and Irwin M. J., Scheduler-based DRAM Energy Management,

Proc. Thirty-ninth Design Automation Conference, pp. 697, June 2002, New Orleans, LA, USA.

36. [Huang et. al. 2003] Huang H., Pillai P and Shin K. G, Design and Implementation of Power-Aware Virtual Memory, *Proc. USENIX Technical Conference*, pp.57-70, Jun., 2003, San Antonio, TX, USA.

37. [Huang et. al. 2005] Huang H., Lefurgy C., Keller T. and Shin K.G., Memory Traffic Reshaping for Energy-Efficient Memory, *Proc. International Symposium on Low Power Electronics and Design*, pp. 393-398, Aug. 2005,San Diego, CA, USA.

38. [Paleologo et. al. 1999] Paleologo, Benini L., Bogliolo A. and Micheli G. De, Policy optimization for dynamic power management, *IEEE Trans. Computer-Aided Design*, Vol. 18, pp. 813–33, June 1999.

39. [Qiu et. al. 2001] Qiu Q., Wu Q and Pedram M., Stochastic modeling of a power-managed system-construction and optimization, *IEEE Trans.Computer-Aided Design,* Vol. 20, pp. 1200-1217, October 2001.

40. [Chung et. al. 2002] Chung E., Benini L., Bogliolo A. and Micheli G., Dynamic Power Management for non-stationary service requests, *IEEE Trans. Computers*, Vol. 51, No. 11, pp. 1345–1361, November 2002.

41. [Simunic2002] Simunic T., Dynamic Management of Power Consumption, Power aware computing, pp. 101 - 125, 2002, ISBN: 0-306-46786-0.

42. [Chen et. al. 2005] Chen Y., Das A., Qin W., Sivasubramaniam A., Wang Q., Gautam N., Managing Server Energy and Operational Costs in Hosting Centers, In *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, Vol. 33 Issue 1, pp. 303-314, June 2005.

43. [Mastroleon et. al. 2005] Mastroleon L., Bambos N., Kozyrakis C., Economou D., Autonomic Power Management Schemes for Internet Servers and Data Centers, *Proceedings of the IEEE Global Telecommunications Conference (GLOBECOM)*, Vol. 2, November 2005.

44. [Lefurgy et. al. 2007] Lefurgy C., Wang X., Ware M., Server-level Power Control. *International Conference on Autonomic Computing (ICAC)*, pp. 4-4, Jacksonville, Florida, 2007.

45. [Li et. al. 2004] Li X, Li Z, David F, Zhou P, Zhou Y, Adve S, Kumar S., Performance-directed energy management for main memory and disks, *Proc. Eleventh International Conference on Architectural support for programming languages and operating systems*, Oct. 2004, Boston, MA, USA.

46. [Diniz et. al. 2007] Diniz B., Guedes D., Meira Jr. W. and Bianchini R., Limiting the power consumption of main memory, *Proc. 34th annual international symposium on Computer architecture*, Jun. 2007, San Diego, CA, USA.

47. [Fan et. al. 2003] X. Fan, Ellis C. and Lebeck A, The synergy between power-aware memory systems and processor voltage, *In Workshop on Power-Aware Computing Systems*, December 2003.

48. [Kimball2007] Kimball's Biology Pages (2007), Organization of the Nervous System, http://users.rcn.com/jkimball.ma.ultranet/BiologyPages/P/PNS.html#autonomic, October 2007.

49. [Ashby1960] Ashby W. R., Design for a brain (Second Edition Revised 1960), published by Chapman & Hall Ltd, London, 1960.

50. [AdSys]Adaptive Systems, http://www.cogs.susx.ac.uk/users/ezequiel/AS/ lectures

51. [Kephart et. al. 2003] Kephart J.O. and Chess D.M., The Vision of Autonomic Computing, *IEEE Computer*, vol. 36(1) pp. 41–503, 2003.

52. [Horn2001] Horn P., Autonomic Computing: IBM's Perspective on the State of Information Technology. http://www.research.ibm.com/autonomic/, October 2001.

53. [NetFlow2007] Introduction to Cisco IOS NetFlow - A Technical Overview, *White Paper,*http://www.cisco.com/en/US/prod/collateral/iosswrel/ps6537/ps6555/ps6601/prod_white_paper0900aecd80406232.html, October 2007.

54. [Rambus1999] Rambus, RDRAM, http://www.rambus.com. 1999.

55. [FBDIMM2006], DDR2 FBDIMM Technical Product Specifications, http://www.samsung.com/Products/Semiconductor/DDR_DDR2/DDR2SDRAM/Module/FBDIMM/M395T2953CZ4/ds_512mb_c_die_based_fbdimm_rev13.pdf. 2006.

56. [Bovet et. al. 2002] Bovet D. and Cesati M., *Understanding the Linux Kernel*, O'Reilly, pp. 294-342, 2002.

57. [Hastie et. al. 2001] Hastie T., Tibshirani R. and Friedman J. H., *The Elements of Statistical Learning*. Springer, pp. 41-75, August 2001.

58. [Ghanbari et. al. 2007] Ghanbari S., Soundararajan G., Chen J., Amza C., "Adaptive Learning of Metric Correlations for Temperature-Aware Database Provisioning", *Proc. 4th International Conference on Autonomic Computing*, Jun. 2007, Jacksonville, Florida, USA.

59. [Cluster2007] A Tutorial on Clustering Algorithms, Retrieved on April 13 2007 from http://home.dei.polimi.it/matteucc/Clustering/tutorial_html/kmeans.html.

60. [$r^2$2002] Co-efficient of Determination –r^2, Retrieved on April 13 2007 from http://www.mis.coventry.ac.uk/~nhunt/regress/good3.html, 2002.

61. [Isci et. al. 2006] Isci C., Buyuktosunoglu A., Cher C-Y., Bose P., Martonosi M., An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget, *39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pp. 347-358, 2006.

62. [Hennessy et. al. 2000] Hennessy J.L. and Patterson, Computer Architecture A Quantitative Approach, 2nd Edition, Morgan Kaufman Publishers, Inc., ISBN 1-55860-372-7(paper), 2000.

63. [Wang1986] Wang I., Simulation of a Modular Hierarchical Adaptive Computer Architecture with Communication Delay, Master's Thesis, The University of Arizona, 1986, 153 pages; AAT 1329505.

64. [Garcia1994] A. Leon-Garcia, Probability and Random Processes for Electrical Engineering, 2nd ed., Addison-Wesley, 1994.