Study of Application-Aware Techniques for System and Runtime Power Management

By

Sharat Chandra Doddaghatta Shashidhar

A dissertation submitted to the

Graduate School – New Brunswick

Rutgers, The State University of New Jersey

In partial fulfillment of the requirements

For the degree

Master of Science

Graduate program in Electrical and Computer Engineering

Written under the direction of

Professor Manish Parashar

And approved by

_____

_____

_____

_____

New Brunswick, New Jersey

OCTOBER 2012

ABSTRACT OF THE THESIS

Study of Application-Aware Techniques for System and Runtime Power Management

By Sharat Chandra Doddaghatta Shashidhar

Thesis Director

Professor Manish Parashar

Abstract

Energy efficiency of large-scale data centers is becoming a major concern not only for reasons of energy conservation, failures, and cost reduction, but also because such systems are soon reaching the limits of power available to them. Like High Performance Computing (HPC) systems, large-scale cluster-based data centers can consume power in megawatts, and of all the power consumed by such a system, only a fraction is used for actual computations. In this paper, we study the potential of application-centric aggressive power management of data center's resources for HPC workloads. Specifically, we consider power management mechanisms and controls (currently or soon to be) available at different levels and for different subsystems, and leverage several innovative approaches that have been taken to tackle this problem in the last few years, can be effectively used in an application-aware manner for HPC workloads.

To do this, we first profile standard HPC benchmarks with respect to behaviors, resource usage and power impact on individual computing nodes. Based on a power and latency model and the workload profiles, we develop an algorithm that can improve energy efficiency with little or no performance loss. We then

evaluate our proposed algorithm through simulations using empirical power characterization and quantification. Finally, we validate the simulation results with actual executions on real hardware. The obtained results show that by using application aware power management, we can reduce the average energy consumption without significant penalty in performance. This motivates us to investigate autonomic approaches for application-aware aggressive power management and cross layer and cross function predictive subsystem level power management for large-scale data centers.

Table of Contents

1 Introduction

1.1 Motivation

Power consumption of high performance computing (HPC) platforms are becoming a major concern for a number of reasons including cost, reliability, energy conservation, and environmental impact. High-end HPC systems today consume several megawatts of power, enough to power small towns, and are in fact, soon approaching the limits of the power available to them. For example, the Cray XT5 Jaguar supercomputer at Oak Ridge National Laboratory (ORNL) with 182,000 processing cores consumes about 7 MW. The cost of power for this and similar HPC systems runs into millions per year.

To further add to the concerns due to power and cooling requirements and associated costs, empirical data show that every 10 degree Celsius increase in temperature results in a doubling of the system failure rate, which reduces the reliability of these expensive system. As supercomputers, large-scale data centers are meant to be clusters composed by hundreds of thousands or even millions processing cores (1) with similar power consumption concerns (2).

In addition, high-performance systems are inefficient in their energy consumption. Ge et al (3) studied five supercomputers and observed that the average performance of these systems is only 54–71% of the peak performance on the optimized benchmark package. That inefficiency is mainly caused by an unequal distribution between the nodes in the cluster of the various computing, communication and I/O activities. During idle or slack times, faster components

waste their energy by waiting for slower components. They could be slowed down or even shut down to save energy. Existing and ongoing research in power efficiency and power management has addressed the problem at different levels, including, for example, data center design, resource allocation, workload layer strategies, cooling techniques, etc. At the platform level (individual node or server), current power management research broadly falls into the following categories - processor and other subsystems (e.g. memory, disk, etc.) level, Operating System (OS) level and application level. At the processor level, unlike the earlier generations of servers and HPC systems that supported only a reduced set of sleep states, current generation systems support advanced power management solutions in hardware. For example, the Intel Nehalem processor has an integrated micro-controller called Power Control Unit or PCU. It has its own embedded firmware and dynamic sensors to monitor current temperature and power in real time, and has an integrated power gate, which eliminates the problem of power leakage.

Although the processor is the most power consuming component, other subsystems have incorporated energy management functionalities such as memory, storage and network interfaces (NIC). Within the OS, there are fewer power management techniques available, and include OS control of processor C-states, P-states and device power states or sleep states. At the application level several approaches have been also proposed such as those based on exploiting communication bottlenecks in MPI programs.

1.2 Problem Description

In this thesis, we study the potential of application-centric aggressive power management of data center's resources for HPC workloads. Specifically, we consider how power management mechanisms and controls (currently or soon to be) available at different levels and for different subsystems, and leverage several innovative approaches that have been taken to tackle this problem in the last few years, can be effectively used in an application-aware manner for HPC workloads.

To use these mechanisms effectively we require cross-layer solutions that are driven by the application and which adapt themselves according to application demands in terms of physical resources. The goal is to investigate power management strategies and their impact on the overall energy consumption in order to define an upper bound of possible energy savings and gain sufficient experience to develop an autonomic runtime to improve the energy efficiency of data centers' resources.

To do this, we firstly profile standard HPC benchmarks with respect to behaviors, resource usage and power consumption. Specifically, we profile the HPC benchmarks in terms of processor, memory, storage subsystem and NIC usage. From the profiles we observe that across different workloads, the utilization of these subsystems varies significantly and there are significant periods of time in which one or more of these subsystems are idle for substantial time-intervals, but still require a large amount of power. It is worth noting that our approach is complimentary to existing solutions at different levels such as those implemented within the OS or those that consider applications' load imbalance and

communication slacks. Based on the empirical power characterization and quantification of the HPC benchmarks, we develop an analytical model that incorporates the applications resource usage patterns, where subsystems in a computing node support different power states, each with specific entry and exit latency, and energy consumption. Furthermore, we use our analysis of HPC workloads to estimate which subsystems are essential for the workload in question, and which subsystems can dynamically enter and exit lower power states (not necessarily idle). We then use the model along with simulations to investigate the potential energy saving of deterministic, application-aware, power management strategies. We then evaluate our proposed algorithm through simulations using empirical power characterization and quantification. Finally, we validate the simulation results with actual executions in real hardware.

The obtained results show that by using application-aware power management, we can reduce the average energy consumption without significant penalty in performance. The results earlier obtained motivate us to investigate autonomic approaches for application-aware aggressive power management and cross layer and cross function predictive subsystem level power management for large-scale data centers. We study this approach for PGAS applications, specifically the UPC implementation of PGAS.

We also look into applications of Partitioned Global Address Space implementations (PGAS), such as Unified Parallel C (UPC), which are an emerging alternative that allows shared memory-like programming on distributed memory systems. It sparks a new area of interest from the point of

view of power management. We study it's power behavior and analyze the potential it offers for runtime power management. From the analysis performed, we derive a mechanism to perform runtime power management. We test the mechanism under different policies and compare the results to come up with optimum policies for PGAS applications - specifically UPC-NAS Benchmarks.

1.3 Contribution

The main contributions of this is summarized as follows: (i) that different existing techniques for energy management can be combined to improve energy efficiency of data center's servers by configuring them dynamically depending on the workloads' resource requirements, (ii) profile HPC benchmarks with respect to behaviors, resource usage and power impact on individual computing nodes and determine empirically (rather than with estimations) possible ways to save energy, (iii) formulate an energy consumption model and propose an algorithm attempting to improve energy efficiency with little or no performance loss, and (iv) quantify possible energy savings of application-centric aggressive power management through simulations and actual executions, (v) profile PGAS implementations (UPC) of HPC applications, (vi) define suitable policies to implement Power management mechanisms from the knowledge gained of the power profiles of PGAS applications, (vii) Implement the policies and observe the trade-offs with energy and runtime.

1.4 Thesis Organization

The rest of the thesis is organized as follows. In Section 2, we describe background and related work. In Section 4.1, we quantify possible power savings and profile HPC workloads using standard benchmarks. In section 4.6, we develop a power model and an algorithm for predictive and aggressive power management based on workload profiles. In Section 4.2, we discuss our evaluation and present the obtained results. In section 5, we explore the opportunities of saving energy dynamically in PGAS applications. We demonstrate PGAS applications showing the potential to be candidates for DVFS, mainly due to load imbalance. In section 6Power Profiling, we profile PGAS applications with respect to power, perform runtime and energy analysis. Then we describe the runtime system that performs application-aware DVFS on PGAS applications in 9.1. We perform analysis on the experimental results and find optimum policies that suit different benchmarks in 9.3. Finally, in Section 10, we conclude the thesis and outline directions for future work.

2 Background

There are two types of power management mechanisms available in HPC systems

2.1 Dynamic Speed Scaling (DSS)

In DSS, one can change the performance state of the component to save power, ie performance can be reduced when not needed and thus energy be saved, and vice versa, ie performance increased when needed.

DVFS (4) is one of the mechanisms in processors, where the performance can be controlled by altering either the supply voltage or the dynamic frequency. Since voltage flow to a system has to be kept constant during execution (at runtime), we can only alter the performance by altering the dynamic frequency and thus control the power consumed. In Linux, DVFS is enabled by the *cpufreq* (5) subsystem provided by ACPI specifications. *Cpufreq* subsystem allows user to set processor frequency either statically or dynamically. The *Cpufreq* structure makes use of governors and daemons for setting a static or dynamic power policy for the system. There are different modules in the cpufreq subsystem which can be used to control CPU frequency.

2.1.1 Performance Governor: Highest Frequency

The CPUfreq governor "performance" sets the CPU statically to the highest frequency within the range ***scaling_min_freq*** and ***scaling_max_freq*.**

2.1.2 Powersave Governor: Lowest Frequency

The CPUfreq governor "powersave" sets the CPU statically to the lowest frequency within the range of ***scaling_min_freq*** and ***scaling_max_freq***.

2.1.3 Userspace Governor

The CPUfreq governor “userspace” allows the user, or any userspace program running with UID “root”, to set the CPU to specific frequency by making the sysfs file “***scaling_setspeed***” available in the CPU-device directory. The user can determine the frequency of CPU by writing the ***scaling_setspeed*** file in the CPU-device directory.

2.1.4 Ondemand Governor

The CPUfreq governor "ondemand" sets the CPU frequency depending on the current usage. If CPU utilization rises above the threshold value set in the ***up_threshold*** parameter, the ondemand governor increases the CPU frequency to ***scaling_max_freq***. When CPU utilization falls below this threshold, the governor decreases the frequency in steps; that is, it sets the CPU to run at the next lowest frequency. The lowest frequency that the CPU can go is bounded by scaling_min_freq. After each ***sampling_rate*** milliseconds, the current CPU utilization is reexamined and the same algorithm is applied to dynamically adjust the CPU frequency to current process load.

2.1.5 Conservative Governor

The *conservative* governor (introduced in Linux kernel version 2.6.12) is based on the ondemand governor. It functions like the ondemand governor by dynamically adjusting frequencies based on processor utilization. However, the conservative governor increases and decreases CPU speed more gradually. If CPU utilization is above ***up_threshold***, this governor will step up the frequency to the next highest frequency below or equal to ***scaling_max_freq***. If CPU utilization is below ***down_threshold***, this governor will step down the frequency to the next lowest frequency until it reaches ***scaling_min_freq***. After each ***sampling_rate*** milliseconds. The current CPU utilization will be reexamined and the same algorithm will be applied to dynamically adjust the CPU frequency to current utilization.

2.2 Dynamic Resource Sleeping

In Dynamic Resource Sleeping, a processor can be put to low power sleep state, if the CPU has been idle. ***Cpuidle*** (6) is a processor-idle management framework in the Linux kernel. It provides an interface for any processor hardware to make use of different processor idle states (C0-C7) which it enters when it is not retiring any instructions. The states here differ in amount of power the processor consumed while being in that state and also the latency to enter-exit this low-power idle state. There may also be other differences like preserving the processor state across these idle states, etc based on a specific processor. For example, a processor may only flush L1 cache in one idle state, but may flush L1 and L2 caches in another idle state. There can also be differences around when an idle state can be entered and what its impact will be on other logical or physical processors in the system. There are C-state governors – menu governor and ladder governor which are responsible for placing the processor in the appropriate C state. The ***ladder governor*** takes a step-wise approach to selecting an idle state. Although this works fine with periodic tick-based kernels, this step-wise model will not work very well with tickless kernels. The kernel can go idle for a long time without a periodic timer tick and it may not get a chance to step-down the ladder to the deep idle state whenever it goes idle. A new idle governor to handle this, called the menu governor, is being worked on. The ***menu governor*** looks at different parameters like what the expected sleep time is (as seen by dyntick), latency requirements, previous C-state residency and then

picks the deepest possible idle state straight away. This governor aims at getting maximum possible power advantage with little impact on performance.

2.3 Performance States

Processor performance states (P-states) are a predefined set of frequency and voltage combinations at which the processor operates. With higher frequencies, you get higher performance, but to achieve that the voltage needs to be higher as well, which makes the processor consume more power. With P-states, the operating system can dynamically change the tradeoff between power and performance all the time. Although changing from one voltage/frequency combination to another takes a bit of time, on current chips, this time is actually really short. This time, as well as certain other characteristics, determines how the operating system should control the frequency/voltage combinations.

Some older x86 processors, as well as embedded processors, have a different behavior, and for that reason the Linux kernel implements several different algorithms for controlling (governing) the performance state that works best on the various processors. For current kernels, you can find information on what is running on your system by looking at the files in this directory: /sys/devices/system/cpu/cpu0/cpufreq. If this directory is not present, there is a good chance that your kernel does not have the CPUFREQ feature enabled.

You can list the available governors by using this command:

```
#cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_available_governors
ondemand userspace performance
```

In the example above, there are three governors available. In addition to the ondemand governor, there are the userspace and performance governors.

You can see what governor is currently active with this command:

```
#cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_available_governors
ondemand
```

You can also change the currently running governor by echoing one of the available governors to the *scaling_governor* file node:

```
# echo ondemand > /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor
```

2.3.1 P state

While a device or processor operates (D0 and C0, respectively), it can be in one of several power-performance states. These states are implementation-dependent, but P0 is always the highest-performance state, with P1 to P*n* being successively lower-performance states, up to an implementation-specific limit of *n* no greater than 16.

P-states have become known as SpeedStep in Intel processors

P-states have become known as SpeedStep in Intel processors as PowerNow! or Cool'n'Quiet in AMD processors and as PowerSaver in VIA processors.

The Performance state are as follows :-

- **P0** max power and frequency
- **P1** less than P0, voltage/frequency scaled
- **P2** less than P1, voltage/frequency scaled

- ....
- **Pn** less than *P(n-1)*, voltage/frequency scaled

2.4 System Monitoring Tools in Linux

System monitoring tools are an indispensible part in analyzing system performance and profiling the system behavior. These tools provide metrics which can be used to get information about system activities. They are basically Unix commands that return realtime data and statistics of system resources like CPU, Memory, Network Interface card, applications consuming most of CPU time. They provide detailed accounts of resources like cpu-idle time, percentage of cpu usage over the last interval, CPU time spent in different performance states, amount of free memory, used memory, swapped memory,amountof data flowing through different network ports etc. Here is a list of useful system commands used to monitor subsystem activity :-

2.4.1 top - Process Activity Command

The top program provides a dynamic real-time view of a running system i.e. actual process activity. By default, it displays the most CPU-intensive tasks running on the server and updates the list every five seconds.

2.4.2 vmstat - System Activity, Hardware and System Information

The command vmstat reports information about processes, memory, paging, block IO, traps, and cpu activity.

2.4.3 mpstat - Multiprocessor Usage

The mpstat command displays activities for each available processor, processor 0 being the first one. mpstat -P ALL to display average CPU utilization per processor

2.4.4 free - Memory Usage

The command free displays the total amount of free and used physical and swap memory in the system, as well as the buffers used by the kernel.

2.4.5 iostat - Average CPU Load, Disk Activity

The command iostat report Central Processing Unit (CPU) statistics and input/output statistics for devices, partitions and network filesystems (NFS).

2.4.6 netstat and ss - Network Statistics

The command netstat displays network connections, routing tables, interface statistics, masquerade connections, and multicast memberships. ss command is used to dump socket statistics. It allows showing information similar to netstat. See the following resources about ss and netstat commands

2.5 Parallel Performance Analysis

Parallel Performance Wizard (PPW) is a performance analysis tool designed for partitioned global-address-space (PGAS) programs, in particular UPC and SHMEM programs. The tool features an easy-to-use interface and tight integration with PGAS programming models via the GASP interface.

In experimental performance analysis, there are two major techniques that influence the overall design and workflow of performance tools. The first technique, ***profiling***, keeps track of basic statistical information about a program's performance at runtime. It gives the user a high-level view of where time is being spent in their application code. The second technique, ***tracing***, keeps a complete log of all activities performed by a user's program inside a trace file. Tracing usually results in large trace files, especially for long-running programs. However, tracing can be used to reconstruct the exact behavior of an application at runtime. Performance analysis in performance tools supporting either profiling or tracing is usually carried out in five distinct stages: ***(i)instrumentation, (ii) measurement, (iii) analysis, (iv) presentation, and (v) optimization.***

User has to take her original application, instrument it to record performance information, and run the instrumented program. The instrumented program produces raw data (usually in the form of a file written to disk), which she gives to the performance tool to analyze. The performance tool then presents the analyzed data to her, indicating where any performance problems exist in her code. Finally, she changes her code by applying optimizations and repeat the process until she achieves acceptable performance.

2.5.1 Instrumentation Process

During the *instrumentation stage*, an instrumentation entity (either software or a user) inserts code into a user's application to record when interesting events happen, such as when communication or synchronization occurs.

Instrumentation may be accomplished in one of three ways: through source instrumentation, through the use of wrapper libraries, or through binary instrumentation. While most tools may use only one of these instrumentation techniques, it is possible to use a combination of techniques to instrument the user's application. Source instrumentation places measurement code directly inside a user's source code files. While this enables tools to easily relate performance information back to the user's original lines of source code, modifying the original source code may interfere with compiler optimizations. Source instrumentation is also limited because it can only profile parts of an application that have source code available, which can be a problem when users wish to profile applications that use external libraries distributed only in compiled form. Additionally, source instrumentation generally requires recompiling an entire application over again, which is inconvenient for large applications.

2.5.2 Measurement

In the *measurement stage*, data is collected from a user's program at runtime. The instrumentation and measurement stages are closely related; performance information can only be directly collected for parts of the program that have been instrumented.

The term *metric* is used to describe what kind of data is being recorded during the measurement phase. The most common metric collected by performance tools is the *wall clock time* taken for each portion of a program, which is simply the elapsed time as reported by a standard clock that might hang on your wall.

This timing information can be further separated into time spent on communication, synchronization, and computation. In addition to wall clock time, a performance tool can also record the number of times a certain event happens, the amount of bytes transferred during communication, and other metrics. Many tools also use hardware counter libraries such as PAPI to record hardware-specific information such as cache miss counts.

2.5.3 Analysis

During the *analysis stage*, data collected during runtime is analyzed in some manner. In some profiling or sampling tools, this analysis is carried out as the program executes. This technique is generally referred to as *online analysis*. More commonly, analysis is deferred until after an application has finished execution so that runtime overhead is minimized. Performance tools using this technique are often referred to as *post-mortem analysis* tools.

The types of analysis capabilities offered varies significantly from tool to tool. Some performance tools offer no analysis capabilities at all, while others can compute only basic statistical information to summarize a program's execution characteristics. A few performance tools offer sophisticated analysis techniques that can identify performance bottlenecks. Generally, tools that provide minimal analysis capabilities rely on the user to interpret data shown during the presentation stage.

2.5.4 Presentation

After data has been analyzed by the performance tool, the tool must present the data to the user for interpretation in the *presentation stage*.

For tracing tools, the performance tool generally presents the data contained in the trace file in the form of a *space-time diagram*, also known as a *timeline diagram*. In timeline diagrams, each node in the system is represented by a line. States for each node are represented through color coding, and communication between nodes is represented by arrows. Timeline diagrams give a precise recreation of program state and communication at runtime. For profiling tools, the performance tool generally displays the profile information in the form of a chart or table. Bar charts or histograms graphically display the statistics collected during execution. Text-based tools use formatted text tables to display the same type of information. A few profiling tools also display performance information alongside the original source code, as profiled data such as the percentage of time an instruction contributes to overall execution time lends itself well to this kind of display.

2.5.5 Optimization

In most performance tools, the *optimization stage* in which the code for the program is changed to improve performance based on the results of the previous stages is left up to the user. The majority of performance tools do not have any facility for applying optimizations to a user's code. At best, the performance tool may indicate where a particular bottleneck occurs in the user's source code and expects the user to come up with an optimization to apply to their code.

2.6 Parallel Performance Wizard

To analyze the performance of your UPC programs, you will need to configure Parallel Performance Wizard to use a UPC compiler. When measuring performance data for UPC programs, all shared data references occurring through direct variable accesses will be attributed to the ‘upc_get’ and ‘upc_put’ regions. Shared data references with affinity to the current thread will be attributed to the ‘upc_get_local’ and ‘upc_put_local’ regions. Additionally, in some UPC implementations (including Berkeley UPC), a ‘upc_barrier’ will be split into ‘upc_notify; upc_wait;’ and show up in the ‘upc_notify’ and ‘upc_wait’ regions.

3 Background and Related Work

In recent work, Liu et al (7) survey power management approaches for HPC systems. As they discuss, since processors dominate the system power consumption in HPC systems, processor level power management is the most addressed aspect at server level. It involves controlling the sleep states or the C-states (8) and the P-states of the processor when the processor is idle (6) C-state is the capability of the processor to go in various low power idle states with varying wakeup latency. P-state is the capability of running the processor at different voltage and frequency levels (9). The Advanced Configuration and Power Interface (ACPI) specification provides the policies and mechanisms to control the C-states and P-states of the processor when they are idle. Modern operating systems (e.g. Linux kernel) implement ACPI-based policies to reduce the processor performance and power when it is less active or in idle state (10). Some processors allow frequency and voltage scaling by which the processor

performance and power can be reduced when the processor is less active or in idle state. The operating system acts as the master controller of the ACPI policies and mechanisms. The ACPI policies and mechanisms are controlled by operating system using Operating System Directed Power Management (OSPM) (11). The Linux kernel can operate in a dynamic ticks mode to save power (10). The dynamic ticks mode eliminates the periodic timer tick and allows processor to be in a deeper sleep states when idle without waking it up at constant interval.

Several approaches to enforce power management based on the workload characteristics have already been developed. Some of the most successful approaches were based on overlapping computation with communication in MPI programs, and using historical data and heuristics.

Kappiah et al. (12) developed a system called Jitter that exploits inter-node bottleneck in MPI programs (i.e. execute blocked processes due to synchronization points in lower P-sates). Lim et al (13) developed a MPI runtime system that dynamically reduces CPU performance during communication regions assuming that in these regions the processor is not on the critical path. Other approaches have also studied the bound on the energy saving for an application without incurring in significant delay (14) and implemented solutions for scientific applications (15). Freeh et al. proposed a model to predict execution time and energy consumed of an application running at lower P-states (16) and techniques based on phase characterization of the applications, assigning different P-states to phases according the previous measurements and heuristics (17). Cameron et al. (18) proposed power management strategies based on

application profiles but they concentrate only on power management of the CPU using dynamic voltage and frequency scaling (DVFS) and does not implement any power control of the peripheral devices. Horvath et al. (19) exploited DVFS with dynamic reconfiguration in multi-tier server clusters, which is a typical architecture of current server clusters. Ranganathan et al. (20) designed a cluster level power management controller, which employes a management agent running on each server and the server which exceeded the power budget according to a Service Level Agreement (SLA), is throttled down to an appropriate level. Wang et al. (21) proposed a control algorithm to manage power consumption of multiple servers simultaneously. The controller monitors the power value and CPU utilization of each server to set the frequency of the processors in a coordinated way. Weissel et al. (22) defined an energy-aware scheduling policy that benefits from event counters. By exploiting the information from these counters, the scheduler determines the appropriate clock frequency for each individual thread running in a time-sharing environment.

Leveraging DVFS mechanisms, Hsu et al. (23) proposed an automatically-adapting, power aware algorithm that is transparent to end-user applications and deliver considerable energy savings with tight control over DVFS-induced performance slowdown. They used Millions of Instructions Per Second (MIPS) as a metric to measure CPU boundedness and take decisions on DVFS control, whereas Malkowski et al. (24) took advantage of memory-bound phase to select CPU frequencies. Rountree et al developed a system called Adagio to collect statistical data on task execution slacks (25) compute the desired frequency and

represent the result in a hash table. When task executes again, an appropriate frequency can be found in a hash table. Other techniques based on switching on/off nodes and other networking devices have been also proposed (26). These techniques have been also applied to virtualized environments (27) but also DVFS techniques have been exploited (28) (29).

Substantial work has been also done for adapting the RAM memory subsystem for saving energy. Delaluz et al (30), (31) studied compiler-directed techniques, as well as OS-based approaches (32) to reduce the energy consumed by the memory subsystem. Huang et al. (33) proposed power-aware virtual memory implementation in OS to reduce memory energy consumption. Fradj et al. (34), propose multi-banking techniques that consist of setting individually banks in lower power modes when they are not accessed. Diniz et al. (35) study dynamic approaches for limiting the power consumption of main memories by limiting consumption by adjusting the power states of the memory devices, as a function of the memory load. Hur et al. (36) propose using the memory controller (thus, at chip level) to improve RAM energy efficiency. They exploit low power modes of modern RAMs extending the idea of adaptive history-based memory schedulers.

Existing research work also addresses the storage subsystem management to improve energy efficiency of servers. Rotem et al. (37) focus on the energy consumed by the storage devices like hard disks in standby mode. They suggest file allocation strategies to save energy with a minimal effect on the system performance i.e. the file retrieval time, while reducing the I/O activity when there is no data transfer. Pinheiro et al. (38) study energy conservation techniques for

disk array-based network servers and propose a technique that leverages the redundancy in storage systems to conserve disk energy (39). Other approaches have addressed energy efficiency of storage systems by spinning-down/up disk (40) and the reliability of such techniques (41). Solid State Drive (SSD) disks have been also taken into account towards saving energy consumption for the storage subsystem (42).

The research work discussed above addresses energy efficiency by managing different subsystems individually (e.g. CPU via DVFS). However, recent approaches have proposed energy efficiency techniques for processor and memory adaptations (43) (44). Li et al. (45) combine memory and disk management techniques to provide performance guarantees for control algorithms.

In contrast to all these approaches, we consider dynamic configuration of multiple subsystems within a single server. Thus, we propose using different mechanisms and techniques that have been already developed in different domains. Thus, our approach is complimentary to existing and ongoing solutions for energy management for data centers.

3.1 Background and Related work on UPC-PGAS programming model

PGAS is a distributed-shared memory programming model that is slowly gaining recognition in High Performance Computing for programmability reasons since it incorporates the shared-memory style of programming, but follows the

distributed-local memory organization which provides scalability and performance. Since the message passing, though present, is not explicit in PGAS, and since it incorporates shared-memory style of programming with remote accesses to shared memory, one-sided communication it merits a study and analysis into it's behavior. Research work on PGAS/UPC have been carried out, focusing on primarily performance, scalability issues and compiler optimizations to improve it's efficiency and make it comparable with that of MPI. Unified Parallel C (UPC) is an extension of ANSI C and is based on partitioned global address space programming model. UPC keeps the powerful concepts and features of C and adds parallelism; global memory access with an understanding of what is local; and the ability to read and write memory with simple statements. The simplicity, usability and performance of UPC have attracted interest from high performance computing users and vendors. UPC (46) utilizes a distributed shared programming model that is similar to shared memory model with the addition of being able to make use of data locality. The distributed shared memory model divides it's shared address space into partitions where each memory partition Figure 1 M_i has affinity to thread Th_i.

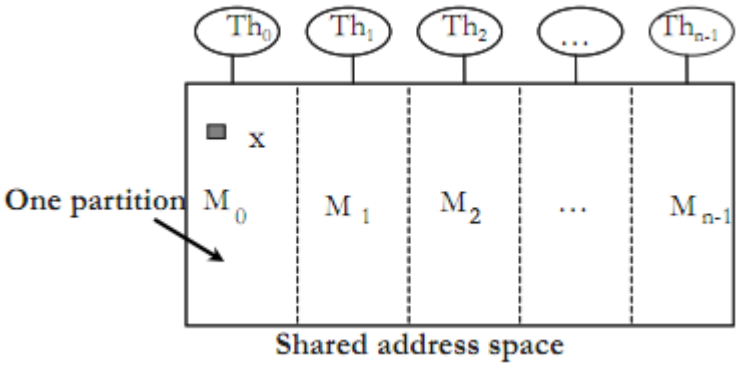

Figure 1

3.1.1 UPC Memory Model

The UPC memory view is divided into private and shared spaces. Each thread has its own private space Figure 2, in addition to a portion of the shared space. Shared space is partitioned into a number of partitions each of which has affinity with a thread, in other words it resides on the thread's logical memory space as seen in figure

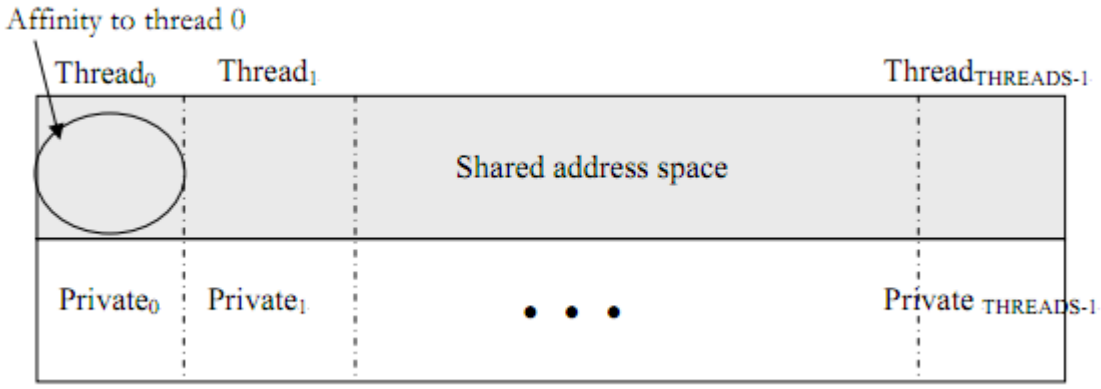

Figure 2

A UPC shared pointer can reference all locations in the shared space; while a private pointer may reference only addresses in its private space or in its local portion of the shared space. Static and dynamic memory allocations are supported for both shared and private memory.

3.1.2 Data Distribution and Coherency in UPC

Data distribution in UPC is simple due to the use of the distributed shared memory programming model. This allows UPC to share data among the threads using simple declaration statements. To share an array of size N equally among the threads the user simply defines the array as a shared, and UPC will distribute

the array elements in a round robin fashion. Since shared memory is accessible by all the threads, it is important to take into consideration the sequence in which memory is accessed. To manage the access behaviors of the threads, UPC provides several synchronization options to the user. First the user may specify strict or relaxed memory consistency mode at the scope of the entire code, a section of a code, or to an individual shared variable. Secondly the user may use locks to prevent simultaneous access by more than one thread. Thirdly, the user may use barriers to ensure that all threads are synchronized before further action is taken.

Chen et al (47) compare the performance of benchmarks compiled with their own optimizations in BUPC with that of HP UPC compiler. They worked on overlapping computation of current iteration and communication of next iteration, message coalescing and aggregation, reordering of shared operations message pipelining or communication overlap techniques, and providing block size information at compile time to avoid affinity tests in order to increase performance. Mallon et al provides an upto-date UPC performance evaluation at various levels, evaluating two collective implementations, comparing their results with their MPI counterparts, and finally evaluating UPC and MPI performance in computational kernels. Efforts have been put by Tarek et al (48) to perform compiler optimizations and hand tuning like space privatization of local shared accesses and block pre-fetching are shown for Sobel edge workload. They also consider hand-tuned versions of NPB in UPC and show that if such optimized libraries are used, they can performs as well as MPI implementations. Tarek in

his paper (49) use synthetic benchmark and UPC application suite involving Nqueens Sobel Edge detection and matrix multiplication to study compiler performance with different optimizations and highlight it's results with regard to remote and local shared memory accesses. They also test the Nqueens benchmark for scalability. William Kuchera and Charles Wallace in their paper (50) also study the UPC memory model and memory consistency issues

4 System Level Power Management

4.1 Studying Energy Saving Possibilities

The fundamental requirement to study the potential energy saving with the approach suggested in this work is to gather reliable usage data for processor, memory, storage subsystem and the NIC for a set of representative and standard HPC workloads.

However, we first characterize and analyze the power dissipation of the different subsystems and quantify the possible saving using existing techniques based on using low power modes to reduce the energy consumption. From the profiling information we will be able to obtain patterns (coarse grain) and then identify possible opportunities of energy saving in servers for large-scale data centers.

4.2 Experimental Environment

The experiments were conducted with two Dell servers, each with a Intel quad-core Xeon X3220 processors, 4GB of memory, two SATA hard disks, and two 1Gb Ethernet interfaces. We also used a 160GB Intel x25-M Mainstream SATA Solid State Drive (SSD) disk. The processors operate at four frequencies ranging from

1.6GHz to 2.4GHz. This is intended to represent a general-purpose rack server configuration, widely used in large data centers.

To empirically measure the “instantaneous” power consumption of the servers a “Watts Up? .NET” power meter was used. This power meter has an accuracy of ±1.5% of the measured power with sampling rate of 1Hz. The meter was attached between the wall power and the server. We estimate the consumed energy integrating the actual power measures over time.

4.3 Power Saving Quantification

4.3.1 Quantification

In order to quantify the possible power savings of a server, we have studied empirically the power characteristics of different subsystems individually. Specifically, we have studied CPU, RAM memory, disk storage, and NIC.

Equation 1 shows the simplified dynamic power dissipation model that we consider for CPU, where C is the capacitance of the processor (that we consider fixed), α is an activity factor (also known as switching activity), and V and f are the operational voltage and frequency, respectively.

$$P_{cpu} \sim C \times V^2 \times \alpha \times f \qquad (1)$$

Table I summarizes the server's power savings and the associated delays for the different subsystem. For the CPU, the workload was generated with lookbusy (a synthetic load generator). During CPU activity, the power demand differs up to around 82W (i.e. 39% of total server power) depending of the frequency used, but without any load the difference is only up to around 8W (i.e. 3.78% of total server

power). However, although CPU power is the more power consuming subsystem of the server, we rely on the CPU frequency management performed within the OS with “cpufreq” using the “ondemand” governor. For disk storage we consider two different possibilities, on the one hand, using spin down/up techniques with traditional disks, and, on the other hand, using a SSD disk. With a traditional disk we can save almost 10W of power (i.e. around 7.5%). However, there is an overhead for spinning down/up the disk. For spinning down the disk the delay is around 0.05 seconds and for spinning up the delay is around 5-6 seconds. There is also an overhead of energy due to the peak power required to spin up the disk's motor (around 60J of energy, according to our experiments). We also consider using a SSD drive, which can save around 14W of power when it is idle (i.e. 3% less power with respect to a disk in low power mode), according to our experiments. The SSD drive also has a much faster access time and does not require spinning down techniques to reduce its power consumption.

Subsystems	Savings	Delay
CPU freq (idle)	8	“instantaneous”
CPU freq (loaded)	82	“instantaneous”
RAM memory	8	“instantaneous”
Hard Disk	10	5-7s
Solid State Disk	52	“instantaneous”
NIC	3	0.15s (on) 3-4s (off)

Table I

4.4 Server's power savings and associated delays

We use low power mode for the network subsystem switching off/on the NIC dynamically. We made the assumption that data centers' servers have usually two different network interfaces (a faster one for actual computations and a slower one for control/administration purposes). Disabling the NIC we can save around

3W (i.e. 2.47%) and the overheads for switching on and switching off the NIC are around 0.15s and 3-4s, respectively.

Memory power dissipation can be classified as being dynamic power dissipation that occurs only during reads and writes, or static power dissipation due to transistor leakage. Equation 2 shows a simple model for memory static power dissipation, where Vcc is the supply voltage, N is the number of transistors, k_{design} is a design dependent parameter, and I_{leak} is a technology dependent parameter. We will consider k_{design} and I_{leak} as fixed parameters.

$$P_{static} = V_{cc} \times N \times k_{design} \times \hat{I}_{leak} \qquad (2)$$

Since the increasing contribution of static power is clearly evident even in today's design, we can reduce the static power dissipation reducing either Vcc or N. Some existing approaches based on multi-banking techniques try to set banks of memory in lower power modes when they are not accessed, thus reducing N. Other approaches may reduce dynamically the voltage when memory is not in the critical path of the running workload. Since these techniques are not standardly available in widely used systems (such as ours), we estimate the potential savings from memory removing physically two of the four banks of memory that are available in the server. Using the same subsystems configurations, but with only 2GB of RAM memory installed, we were able to save around 8W of power (i.e. 5.78%), on average. We estimate short delay for switching to low power mode.

4.4.1 Workload Profiling -

The methodology involves profiling the workload behavior into I/O intensive, memory intensive, communication intensive and computes intensive regions with respect to time. Most of the standard profiling utilities are designed for comparing computation efficiency of the workloads and systems on which they are running, hence their outputs are not very useful from the subsystem usage point of view.

We profiled standard HPC benchmarks with respect to behaviors and subsystem usage on individual servers. To collect run-time OS-level metrics for CPU utilization, hard disk I/O, and network I/O we used different mechanisms such as ``mpstat'', ``iostat'', ``netstat'' or ``PowerTOP'' from Intel. We also patched the Linux kernel 2.6.18 with the ``perfctr'' patch so that we can read hardware performance counters on-line with relatively small overhead. We instrumented the applications with PAPI and, since the server architecture does not support total memory LD/ST counter, we counted the number of L2 cache misses, which indicates (approximately) the activity of memory.

A comprehensive set of HPC benchmark workloads has been chosen. Each stresses a variety of subsystems - compute power, memory, disk (storage), and network communication. They can be classified in three different classes:

- *Standard:* **HPL** Linpack that solves a (random) dense linear system in double precision arithmetic, and **FFTW** that computes the discrete Fourier transform.
- *CPU intensive*: **TauBench**, which is an unstructured grid benchmark of Navier Stokes solver kernels.

- *I/O intensive*: **b_eff_io**, which is a MPI-I/O application, and **bonnie++** that focus on hard drive and file system performance. We run two distributed instances of bonnie++ using a script and ssh.

Figures 3-20 show the obtained profiles for different benchmarks, along with the server power consumption during their execution. Axes of the plots have time as the X-axis and on the Y-axis we show, from the top to the bottom: CPU utilization, memory utilization (L2 cache misses), disk utilization (number of blocks accessed), network utilization (traffic of packets on the NIC), the average p-state residency of the CPU's cores, and power consumption. The plots show the measurements as well as the bezier curves (dashed lines) to better identify their trends, except the plots of p-state residency that only show the bezier curves, for readability.

In the following subsection we discuss the trends and the power saving opportunities of the profiles shown in Figure 3-20

4.5 Power Saving Opportunities

The plots included in Figures 3-20 plot different application profiles for resource utilization and power consumption. Some observations are listed below.

The HPL benchmark was configured to run two problems of the same size. Thus, as is shown in the CPU utilization plot, there is an interval of time in the middle of the benchmark's execution with lower CPU utilization. It helps us to appreciate a clear correlation between the CPU utilization and power consumption.

In Figure 4, L2 cache misses increase steadily during the HPL's execution, which suggests high memory activity. The NIC is used in bursts, except at the end of the two problems solved by HPL (Figure 9).

P-state residency is distributed almost evenly among the maximum and minimum CPU frequencies (Figure 15). This is explained due to the fact that the OS scales the frequency down/up dynamically following the iterative compute+synchronization pattern of HPL. Also, since the network was not very fast (100Mbps), the communication slack is significant.

In contrast to the other subsystems, disk utilization is scarce (Figure 12). Therefore, we find higher opportunities for energy saving using disk power management techniques.

With CPU intensive benchmarks (i.e. TauBench), the CPU utilization steadily remains close to the maximum utilization (400%), and the p-state residency at maximum CPU frequency is at 100% during almost the whole execution of the benchmark. This is because TauBench has much less MPI synchronization than HPL.

As both b_eff_io and bonnie++ benchmarks are I/O intensive, the CPU utilization running these benchmarks is low and the p-state residency is predominated by low frequencies.However, memory and disk have high activity. In fact, the disk is accessed frequently (see the disk utilization plot for b_eff_io) thus making it infeasible to spin-up and spin-down the disk.

The main difference between b_eff_io and bonnie++ is that b_eff_io has periods of intensive NIC utilization (Figure 10), and bonnie++ does not perform synchronization over the network. Therefore, b_eff_io has lower average CPU utilization than bonnie++, but the CPU utilization is steadier for bonnie++. Moreover, we can appreciate a clear correlation between memory utilization and power consumption with bonnie++ that is not present with b_eff_io.

The NIC utilization plot for b_eff_io (Figure 10) shows multiple long duration idle periods offering opportunities to save energy even though other subsystems are active most of the time.

As a result, with bonnie++ we have higher opportunities for energy saving from NIC (Figure 11) and memory (Figure 8) because the network traffic is only at the beginning and at the end of its execution, and there are some long intervals of time without L2 cache misses.

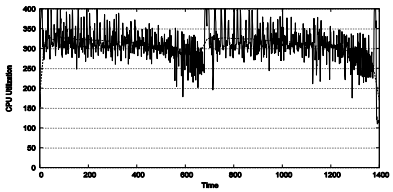

Figure 3

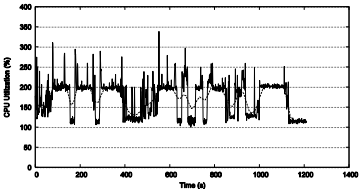

Figure 4

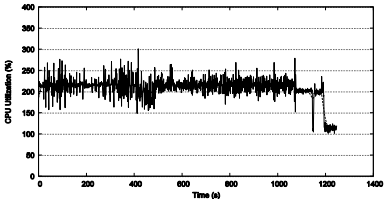

Figure 5

CPU Utilization of HPL, b_eff_io, bonnie 1

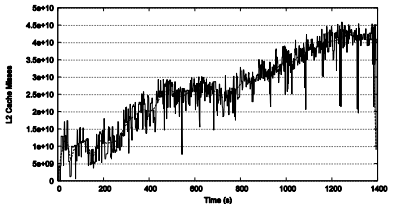

Figure 6

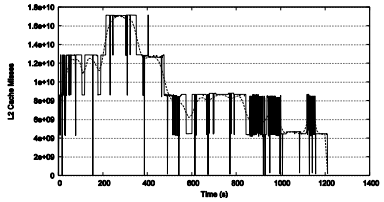

Figure7

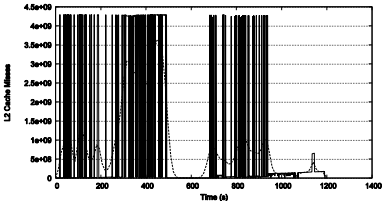

Figure 8

Memory Utilization of HPL, b_eff_io,bonnie++

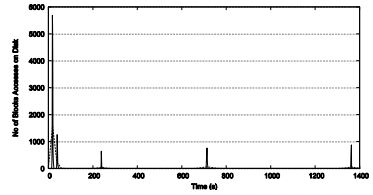

Figure 9

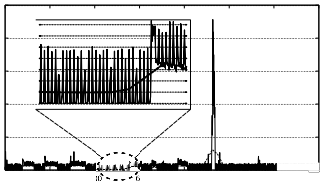

Figure 10

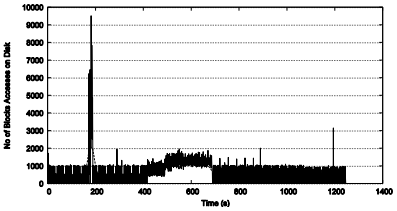

Figure 11

Network Utilization of HPL,b_eff_io,bonnie++

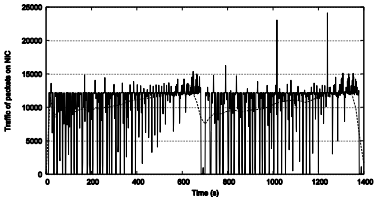

Figure 12

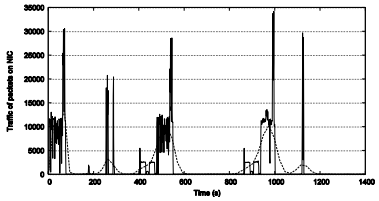

Figure 13

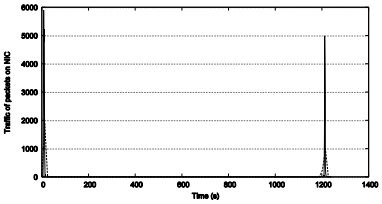

Figure 14

Disk Utilization of HPL, b_eff_io,bonnie++

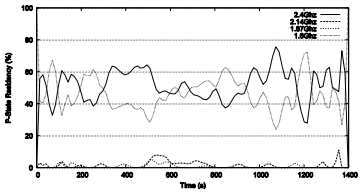

Figure 15

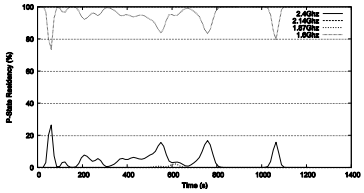

Figure 16

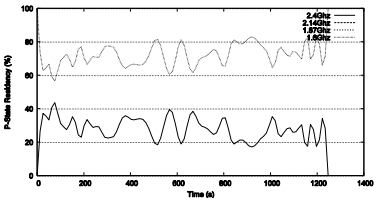

Figure 17

P-State Residency of HPL,b_eff_io,bonnie++

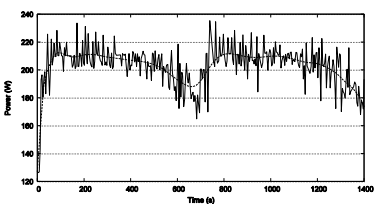

Figure 18

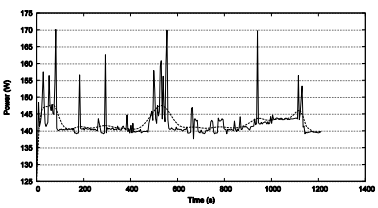

Figure 19

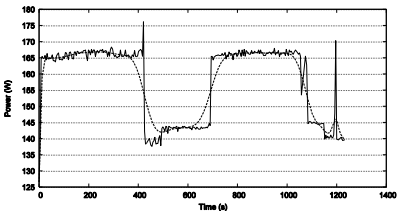

Figure 20

Power Profiles of HPL,b_eff_io,bonnie++

4.6 Towards Application-Centric Aggressive Power Management

In this section, we build a model with the goal of developing an algorithm for aggressive power management. The power dissipated by a server P can be computed as the sum of it's static and dynamic power as described in Equation (3)

$$P_{dynamic} = P_{static} + P_{dynamic} \quad (3)$$

$P_{dynamic}$ is composed of power contributions from the CPU and sub-systems such as memory, storage and the NIC,

$$P_{dynamic} = P_{cpu} + P_{mem} + P_{disk} + P_{nic} \quad (4)$$

Since we assume that the OS manages the CPU power configuration efficiently, we do not consider CPU henceforth.

We made the simplification that each subsystem has two operational modes: *active* (regular configuration) and *idle* (low power mode). Therefore, we consider the latencies and overheads that would be involved in switching between these subsystem power modes. We define *ton* as the latency to switch from idle to active mode and t_{off} the latency to switch from active to idle mode. We also define Et_{on} and Et_{off} as their respective associated energy costs. Since we consider that different power configurations for the subsystems can be used during a workload execution, we model the workload execution time *(T)* as the sum of time intervals

where each two consecutive time interval have different power configurations (Equation 5).

$$\sum_{i=1}^{n} t_i \quad (5)$$

The energy consumed by the different subsystems (sys={mem, disk, nic}) during the execution of a workload *(E)* is defined as the sum of the energy consumed in each time interval (Equation 6).

$$E = \sum_{i=0}^{n} \cdot \sum_{.}^{sys} E_{t_i}^{sys} \quad (6)$$

The energy consumed within a time interval includes the energy overhead of switching between power modes as Equation 7 shows.

$$E_{t_i}^{sys} = P_{active}^{sys} \cdot t_{i,active} + P_{idle}^{sys} \cdot t_{i,idle} + E_{t_{on/off}}^{sys} \quad (7)$$

P_{active}^{sys} is the power consumed if the subsystem *sys* is active and $t_{i,active}$ is the time duration in active mode in seconds. P_{idle}^{sys} is the power consumed if the subsystem *sys* is idle (low power mode) and $t_{i,idle}$ is the time duration in idle mode in seconds.

4.7 Predictive and Aggressive Power Management (PAPM)

Building on the model described above, we develop the PAPM algorithm that transitions the subsystems to the appropriate power mode based on an a-priori knowledge of the application profile. The algorithm takes into account the latency of each device in making a transition from one state to another, as well as the overhead energy consumed to make the transition.

Algorithm 1 shows the PAPM algorithm focusing on switching off subsystems and deciding the appropriate low power state to transition a subsystem from an active state. The algorithm to switch back to an active state from an idle state is symmetric to Algorithm 1. It has three conditions: *Idle-Condition*, *Time-Condition* and *Energy-Condition* on which transition to a low power state is dependent.

```
Algorithm
Input: Physical memory, disk, NIC
Output: Power state for a given subsystem
for time = 0 to i do
    for sys = disk, memory, NIC do
        if Idle-Condition then
            if Time-Condition then
                if Energy-Condition then
                    P^sys_PwrSt = P^sys_idle;
                end
            end
            else
                P^sys_PwrSt = P^sys_active;
            end
        end
    end
end
```

Algorithm 1: Algorithm for idle state transition of memory, storage and NIC subsystems

Idle-Condition checks if the subsystem is going to be idle in the next time interval and it is given by

$$\boldsymbol{WorkloadProfile}(t_{i+1}^{sys}) \longrightarrow 0 \qquad (8)$$

Equation 8 indicates, based on the workload profile data, that if the next time instance has low or no activity then proceed further to transition the subsystem to a low power state. *Time-Condition* is given by,

$$(t_{i+1}^{sys} - t_{i}^{sys}) > t_{off}^{sys} + t_{on}^{sys} \qquad (9)$$

This condition checks if it is feasible to transition to any of the available lower power states based on the latency of the subsystems and the idle time between the two active periods. The active period is denoted by $t_{i\,active}^{sys}$.

If the time for which subsystem is idle is greater than the latencies to enter and exit any of the low power states, then the system should be transitioned to low power mode. For simplicity, in Algorithm 1 we only consider two possible power modes (active/idle).

'Energy-Condition is given by,

$$P_{active}^{sys} . t_i > P_{idle}^{sys} . t_{i,idle} + E_{t_{off}}^{sys} + E_{t_{on}}^{sys} \quad (10)$$

This condition checks if it is worthwhile to transition the system to a low power state and if any energy savings will be achieved. It also takes into consideration the power to transition the subsystem from a low power mode to operating mode. If the sum of energy consumed in low power state and energy needed to bring back the subsystem to active mode is less than the energy used by subsystem when it continues to be in high power state when idle, it is worthwhile to transition the system to a low power state.

Other considerations can be incorporated in this model such as ensuring that the energy saving is significant enough to over-ride the cost of reducing the lifetime of the disk.

4.7.1 Experimental Evaluation

In this section, we evaluate the energy savings that can be achieved with the PAPM algorithm proposed previously with a deterministic approach, in order to

show its potential in a single server. To do this, we firstly use simulation along with the workload profile information discussed in section 4.4.1. Then, we present results obtained from actual executions to validate the former simulations. Finally, we analyze the potential energy saving in large-scale data centers.

4.7.1.1 Simulation

In this subsection, we evaluate the PAPM algorithm in a single server using simulations and present the estimated energy savings. The PAPM algorithm simulation was developed using MATLAB. We used the benchmarks presented in section III that, as we discussed previously, have different requirements and behaviors in terms of subsystems utilization. Although we present the saving for a single server, the results were obtained using the testbed described in section 4.2.

We also perform our simulations based on the subsystem usage time obtained from the workload profile data of an individual server. Since the approach is deterministic, we assume that the workload profile is known in advance. Following the PAPM algorithm, when a subsystem is switched to another power mode (e.g. spinning up/down the disk), we consider the savings that we quantified in section 4.3 as well as the associated overheads in terms of delay and energy.

Table II present the obtained results. Specifically, it shows the run time, the energy consumption, and the estimated energy savings of various workloads. The granularity of the delays is in seconds due to the limitations of the instruments, and the fact that our approach is based on a coarse-grained model.

It is worth noting that run time and energy consumption are obtained through actual measurements, while the energy savings for memory, disk and NIC were obtained through the simulations.

We present the results for each benchmark with two different configurations: DVFS and non-DVFS. The former configuration uses ACPI enabling DVFS with ``cpufreq" and the ``ondemand`` governor. The latter configuration uses ``userspace" governor at the maximum CPU frequency. Therefore, the CPU savings are also obtained from actual measurements. Although the total energy savings are higher considering the CPU, we will consider the savings only from the other subsystems.

Benchmark	DVFS	RunTime(s)	Energy (J)	Energy Savings				
				CPU	Memory	Disk	NIC	Total (J)
HPL	x √	1,382 s 1,383 s	298,546 J 292,824 J	- 5,722 J	1,380 J	5,338 J	240 J	6,958 J 12,680 J
B_eff_io	x √	1,206s 1,212 s	164,224 J 161,460 J	- 2,764 J	5,297 J	-	1,124 J	6421 J 9,185 J
Bonnie++	x √	1247 s 1248 s	190,613 J 187, 533 J	- 3,080 J	3,263 J	574 J	3,841 J	7,678 J 10,758 J
tauBench	x √	1134 s 1136 s	251,904 J 244,473 J	- 7,431 J	3,377 J	7,297 J	1,979 J	12,653 J 20,084 J
FFTW	x √	1052 s 1055 s	198,621 J 193,146 J	- 5,431 J	2,112 J	6,927 J	297 J	9,336 J 14,811 J

Table II
Energy savings with PAPM Using Simulations For Different Benchmarks

In fact, the results state that DVFS does not penalize the execution time significantly (0.22% on average) while it provides significant savings of energy (2.23% on average). This supports our argument that modern OS-driven power management mechanisms can be leveraged for application-aware power management, thus not requiring any specific CPU power management. From Table II we can appreciate that CPU and network-intensive benchmarks provide

more opportunities of energy savings from the disk (e.g. FFTW) while I/O-intensive benchmarks provide more opportunities of energy savings from other subsystems (e.g. NIC). Furthermore, benchmarks with higher utilization of the different subsystems (i.e. HPL) obtain less energy savings. Although the average energy saving is around 4%, we can infer that PAPM can improve the energy efficiency and hence the power efficiency of HPC workloads from around 2% to around 5% depending on the workload profile. However, our simulations are very conservative because we only consider static power to estimate the potential energy savings from the memory subsystem (our quantification was done with the server under idle condition). In fact, the simulations may obtain higher energy savings without significant penalty in run time if we consider additional memory management techniques, such as Dynamic Voltage Scaling (DVS), during intervals of time where memory is not in the critical path.

4.7.1.2 Validation

In order to validate our model we implemented the PAPM algorithm and performed actual experiments. They provides us with a very good opportunity to prove the effectiveness of PAPM. Since we do not support any mechanism to reduce the voltage of RAM memory or switching off banks of memory using multi-banking techniques, we focus on in disk storage and NIC. Specifically, we used the following configurations:

- *Reference*: regular execution with DVFS enabled.

- *PAPM*: implementation of PAPM with a script that deterministically switches to appropriate power state, based on the profile obtained in previous executions of the same benchmark.
- *PAPM+SSD*: use of SSD technology for storage and PAPM algorithm for NIC only

Table III present the obtained results. It shows the run time and the difference with respect to *Reference* configuration, the energy consumption and the difference with respect to *Reference* configuration and, in order to consider both energy and performance, the Energy Delay Product (EDP) and the difference with respect to *Reference* configuration.

Benchmark	Configuration	Runtime (s)	%	Energy (J)	%	EDP	%
HPL	*Reference*	1,383 s	-	292,824 J	-	404,975,592	-
	PAPM	1,385 s	+0.14%	287,906 J	-1.67%	398,749,810	-1.53%
	PAPM+SSD	1,385 s	+0.14%	281,559 J	-3.84%	389,959,215	-3.70%
b_eff_io	*Reference*	1,212 s	-	161,460 J	-	195,689,520	-
	PAPM	1,217 s	+0.41%	157,335 J	-2.55%	191,476,695	-2.15%
	PAPM+SSD	1,134 s	-6.43%	143,768 J	-10.95%	163,032,912	-16.68%
Bonnie++	*Reference*	1,248 s	-	187,533 J	-	234,041,184	-
	PAPM	1,249 s	+0.08%	182,904 J	-2.47%	228,447,096	-2.39%
	PAPM+SSD	1,169 s	-6.33%	168,606 J	-10.09%	197,100,414	-15.78%
TauBench	*Reference*	1,136 s	-	244,473 J	-	277,721328	-
	PAPM	1,139 s	+0.26%	236,496 J	-3.26%	269,368,944	-3.00%
	PAPM+SSD	1,137 s	+0.08%	229.446 J	-6.14%	260,880,102	-6.06%
FFTW	*Reference*	1,055 s	-	193,146 J	-	203,769,030	-
	PAPM	1,057 s	+0.19%	187,677 J	-2.83%	198,374,589	-2.64%
	PAPM+SSD	1,051 s	-0.38%	177,071 J	-8.32%	186,101,621	-8.67%

Table III
Energy Savings with PAPM Using Actual Executions

With *PAPM* the overheads of switching the power modes do not penalize the run time significantly (0.16%, on average). However, the average energy saving is around 2.5% which is 37.5% lower with respect to the average energy saving estimated through simulations. The percentage of savings for EDP is only slightly

lower than the energy savings. We believe that the differences between the results obtained with simulations and with actual experiments are motivated by two factors. On the one hand, the lack of memory power management and, on the other hand, insufficient precision at some power mode modification times that may result in some energy saving loss.

The *PAPM+SSD* configuration allows us to identify the potentials and tradeoffs of using SSD technology. Since SSD disks do not require spinning down techniques to switch to low power mode, the PAPM algorithm only focus on NIC (potentially it may consider also memory). As is shown in Table III using SSD technology reduces the energy consumption 7.86% on average. For non-disk intensive benchmarks (e.g. HPL) the savings are moderate (5%, on average) while the energy savings are much higher for I/O intensive benchmarks (around 10%, on average). The run times obtained with *PAPM* and with the other configurations are similar for the non-disk intensive benchmarks. However, with I/O intensive benchmarks (b_eff_io and bonnie++) the run times are reduced up to 6.43%. This is explained due to the fact that both b_eff_io and bonnie++ benchmarks access the disk frequently (see Figure 8,14) and SSD drives have a much shorter access time. As SSD technology is still very expensive, we can assume that only some servers of a data center will have a SSD disk available (at least for cache data). Hence, the problem will include mapping the applications that can take more advantage of SSD technology to the nodes with a SSD installed, depending of the application profile.

Therefore, servers with SSD disks may be assigned to I/O intensive applications while non-disk intensive applications (e.g. HPL) may take advantage of power management techniques on traditional disks (spinning dow/up), if the benefit of using SSD does not compensate the cost of SSD technology. It is worth noting that this trade off will also depend of the estimated workload execution time.

4.8 Scaling to Large HPC Cluster Level

PAPM does power savings on per node basis per run basis, which can produce significant amount of energy savings if scaled to data center magnitudes. We assume that the data center is composed by the server configuration used throughout this thesis, the average energy savings obtained from our validation experiments (5.21%) and an average power demand of 200W due to almost continuous load. If the daily energy consumption is $200W ×24h × 3,600s = 0.48kWh$, the daily energy saving is around 0.25kWh. Thus, the yearly energy saving will be $0.25kWh × 365days = 91.28kWh per node. Considering that the average kWh price (March 2010) in United States is $0.125 and €0.36 in Europe, the yearly saving will be $11.41 or €32.86 per node. For a 1,000 node data center it will save approximately $11,410 or €32,860 per year only on computational costs. As the size of HPC cluster increases the savings would increase by same order of magnitude.

5 Runtime Power Management with PGAS

PGAS is a parallel programming model that presents a single shared partitioned address space, where variables may be directly read and written by any processor,

but each variable is physically associated with a single processor. Due to this feature, the portions of shared address space may have an affinity for a particular thread thereby exploiting locality of reference. Where parallel programs are usually written either using distributed memory model or Shared Memory model, depending on the feasibility and the demands, PGAS model brings with it, the advantages of the Shared memory programming like OpenMP as well as the Distributed memory programming like MPI. It has the performance and data locality (partitioning of data) features of MPI and

the programmability and data referencing simplicity of shared-memory model. Hence it has the potential to dramatically improve runtime performance and programmer productivity on increasingly ubiquitous multi-core architectures.

5.1 Main features of PGAS

It has support for distributed data structures like global arrays. The PGAS model uses “global address space” which presents the programmer a uniform view of the global shared memory. It allows the programmer to access the remote shared variables and the local shared variables in a seamless way, without him having to worry about the location of the data. The programmer has control over the placement of data and how the global data gets distributed among the different processors. One-sided communication for improved inter-process performance – One process can access a remote variable in another process, without involving the remote process in the communication.

5.2 Differences with MPI and OpenMP

One sided communication in PGAS remote accesses as opposed to two-way synchronization for SEND/RECEIVEs in MPI access to remote memory is implicit in PGAS. No explicit language-level APIs/library calls are required to access remote memory, unlike MPI which uses MPI calls to access remote memory. Two level memory hierarchy exists – private data on local machines, shared data on local and remote machines (accessed using global pointers). PGAS implementations like UPC have split-phase barriers apart from the regular barriers for synchronization. These split-phase barriers allow a process to perform operations on local data (without touching shared memory), while waiting for other process to reach the barrier. Programmer has control over data layout and has the flexibility to distribute data over processes differently.

Work distribution in order to leverage data locality is possible - similar to OpenMP.

Some of the PGAS implementations that are being developed are:-

- Unified Parallel C
- Global Arrays

Most popular implementation developed by UC Berkeley and Livermore Berkeley National Laboratories uses the SPMD model of computation. Library developed by scientists at Pacific Northwest National Laboratories follows MIMD model of computation.

Some of the implementations of PGAS model are

- Chapel – Developed by CRAY Inc
- Titanium – Parallel dialect of Java , developed at UC Berkeley

- X 10– Being developed by IBM research
- Fortress – Created by Sun Microsystems
- Co-array Fortran

In our thesis, we shall focus our work on UPC implementation-Unified Parallel C

5.3 Scope for Power Management with PGAS Programming Model

Load imbalance of parallel applications can be exploited to save CPU energy without penalizing the execution time. An application is load imbalanced when some nodes are assigned more computation than others. The nodes with less computation can be run at lower frequency since otherwise they have to wait for the nodes with more computation blocked due to synchronization calls. HPC systems use an increasingly large number of processors with MPI-based parallel programs. The resulting increase in the number of processes of an application usually decreases the load balance degree of the application. In load imbalanced MPI applications, there are processes that complete their computation and have to wait for the other processes to communicate. These nodes can run at lower frequencies and save energy consumed by CPU without increasing the execution time. The dynamic power is proportional to the product of the frequency and the square of the voltage and it can be reduced via DVFS technique.

It is possible to realize the potential for a runtime system Power Management mechanism, if one studies the application runtime pattern. By studying the application execution pattern, we can understand the application at a lower level of granularity which would help us see inside the application- the sequence of calls being made and the duration of the calls. It highlights the different sub-

routine library calls made, sequence of execution of the calls, slack involved in communicating between processes and the load imbalance in the execution runtime - from which we can appreciate the benefit of performing DVFS.

5.4 Opportunities for Power Management using DVFS

5.4.1 At Barriers – Due to upc_wait operation

The **Figure 21** below depicts an execution trace of UPC NAS-MG benchmark. The red colored lines denote threads after having reached their barrier and waiting for other threads to synchronize. We can see that the different threads hit the barrier at different points of time. Hence the threads wait at the barriers to synchronize with remaining threads, during which they do not perform any useful task. This provides us with an opportunity to reduce CPU frequency, since it is not required to be performing at it's peak.

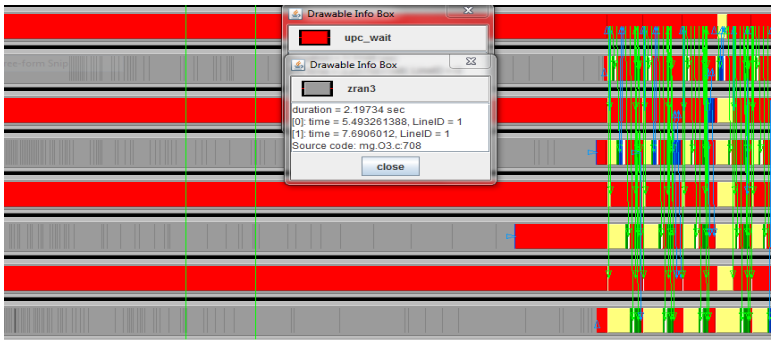

Figure 21 : Threads approaching Barrier at Different times due to load imbalance

5.4.2 During Remote Memory Calls (memget/memput)

We also know that when a thread performs remote memory operations from a local node, it has to wait for the data to get transferred from the remote node. Since the processor bound to the thread is again not doing any useful work, other than waiting for the data transfer, we can insert power management mechanisms

to reduce consumption of power. In **Figure 22,** we can see the threads performing memget operation – depicted in green colour. The processor bound to the thread is not performing any work other than waiting for the data to arrive from the remote node – which provides with ample time to benefit from DVFS techniques in saving power.

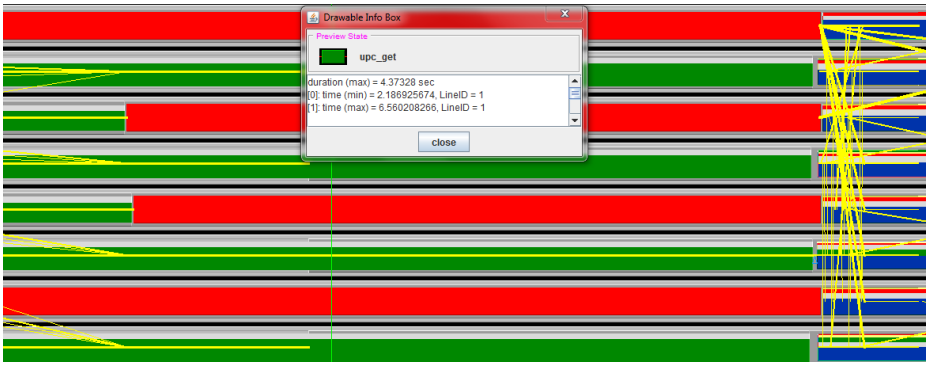

Figure 22: Memget operation (upc_memget)

6 Power Profiling of UPC-NAS applications/Measuring Power with Wattmeter

Power profiles of applications give a broad overview of the nature of power usage of applications. Comparing power profiles at different frequencies help us appreciate how significantly frequency plays a role in the power consumption of an application.

From the figures below, we can see the gap between the power traces at 1.6 GHz and 2.4Ghz.It also clearly shows that the application finishes execution faster when run at 2.4 GHz. The power profile also reflects the iterative nature of the applications.

The NAS-CG benchmark is a memory intensive benchmark. It performs computation for a short period followed by a longer remote memory fetch operation, followed by a barrier used for synchronization, and this happens iteratively. It mostly consists of small reads followed by small writes. Studying the profile of CG (Figure 23), we can see that it consumes energy at a steady rate. The runtime at 1.6 GHz consumes lot more time than the one at 2.4 GHz. NAS-IS application profile shows sharp spikes at regular intervals which contends to be a good candidate for performing DVFS.

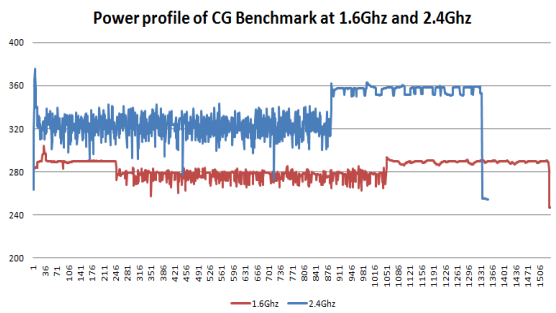

Figure 23 : Power Profile of CG

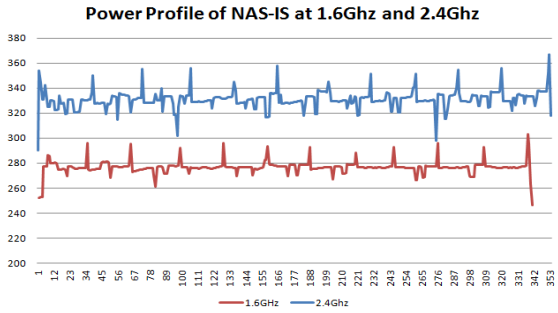

Figure 23 : Power Profile of IS

Power profile of CG and IS

NAS-MG Benchmark has remote memory fetch at the beginning of execution. As the application progresses, memory operations are mainly longer remote memory write operations and short local memory reads, with computation. NAS-MG varies constantly such that the spikes are spaced very closely to each other and hence does not seem to be good candidate for DVFS since it would result in overhead while performing DVFS so often. NAS-FT, unlike MG, consists of mainly longer memory read operations, with brief periods of computation and short local memory write operations.

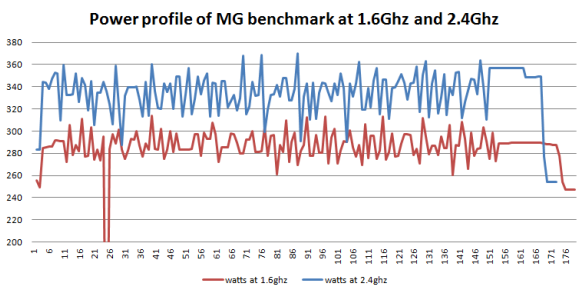

Figure 24 : Power Profile of MG

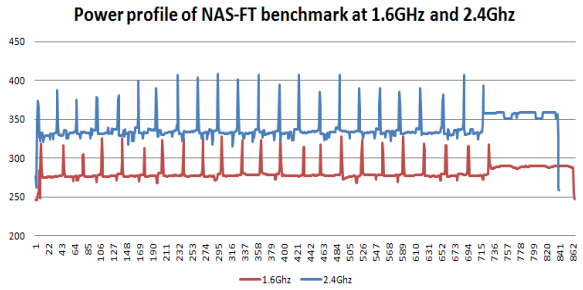

Figure 25 : Power Profile of FT

Next section shows further analysis about the energy savings and runtime features of NAS applications.

7 Power Profiling of UPC-NAS kernels

We run UPC-NAS benchmarks at different frequencies (1.6 GHz, 1.867 GHz and 2.4 GHz) to measure the energy consumed by them at each frequency. We notice that all benchmarks consume least energy at 1.6 GHz except the EP benchmark that consumes the least energy at 2.4 GHz.

The highest energy-gain percentage is registered by the NAS-FT benchmark (15%) whereas EP benchmark registers 18% energy loss when run at 1.6 GHz as compared to the run at 2.4GHz. This can be reasoned by observing the runtimes of the benchmark which also play a role in determining energy consumption of application. The runtime of EP at 1.6 GHz is nearly 40% greater than the runtime of EP at 2.4 GHz thus consuming power for too long a duration to register any energy savings. This result of EP can be explained by considering the nature of benchmark which is highly parallel with little communication. This causes a higher frequency execution to finish a lot faster than the lower frequency run, as it has very little slack- thus resulting in a huge difference between the two

runtimes. Since 2.4GHz run consumes lot lesser time, the energy consumed will be lesser than that due to 1.6GHz run. We can see that NAS-FT has the least runtime loss and the highest EDP gain percentage (13.8%) followed by NAS-IS and NAS-MG with 10.7% and 7.2% respectively.

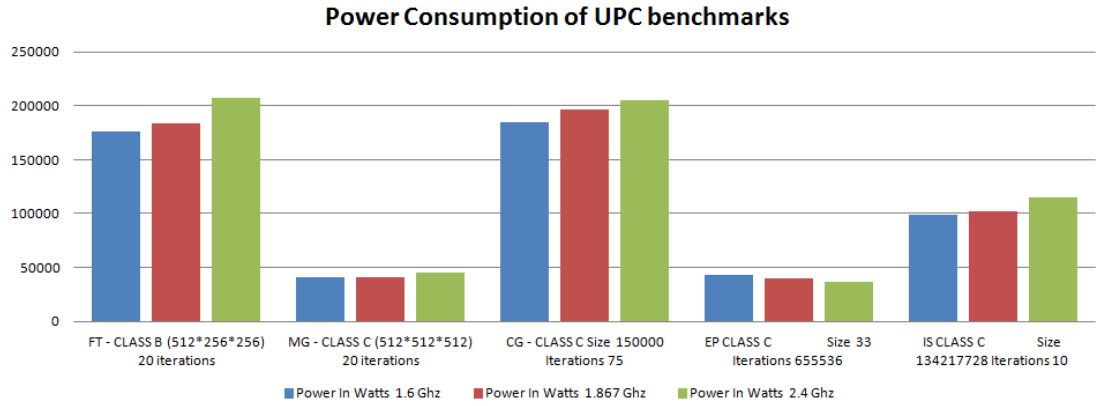

Figure 26 : Power Consumption of NAS-UPC Benchmarks

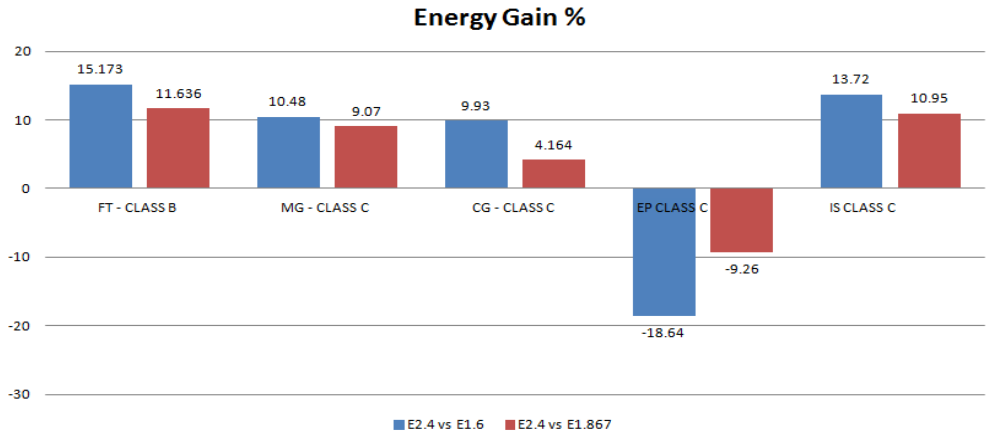

Figure 27 : Energy Gain of NAS-UPC Benchmarks

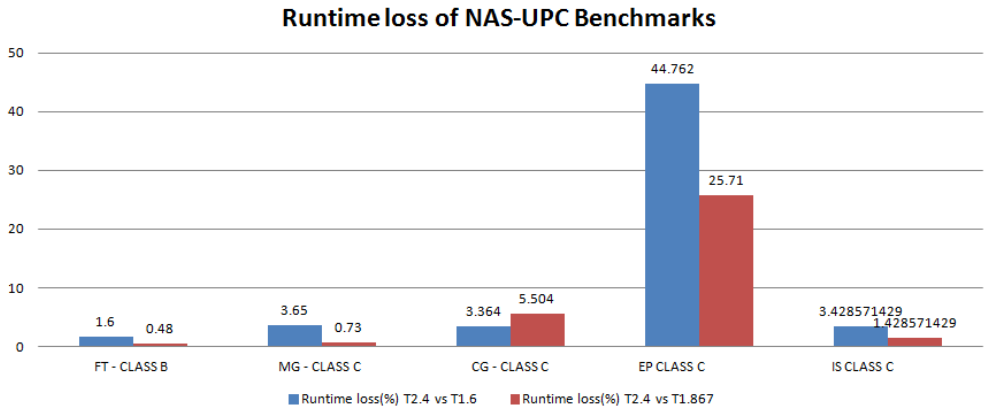

Figure 28 : Runtime Loss of NAS-UPC Benchmarks

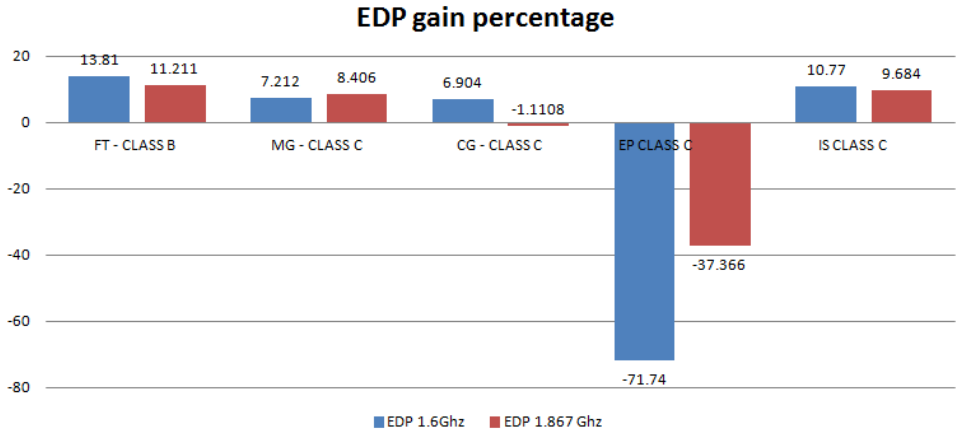

Figure 29 : EDP gain percentage of NAS-UPC Benchmarks

	Application Runtime (s)			Energy in Joules (J)			Runtime Loss (%)		Energy gain (%)	
	1.6 GHz	1.867 GHz	2.4GHz	1.6GHz	1.867GHz	2.4GHz	$T_{2.4/1.6}$	$T_{2.4/1.867}$	$T_{2.4/1.6}$	$T_{2.4/1.867}$
FT - B	635	628	625	176141.5	183487	207649	1.6	0.48	15.17	11.636
MG-C	142	138	137	40524.1	41162	45268	3.649	0.73	10.479	9.07
CG-C	676	690	654	184326	196133	204656	3.363	5.504	9.93	4.16
EP-C	152	132	105	43099	39694.7	36327.7	44.76	25.714	-18.63	-9.26
IS-C	362	355	350	99363	102555	115173.6	3.42857	1.42857	13.727	10.95

Table IV
Summary of Energy and Runtime data of NAS-UPC Benchmarks at 1.6Ghz, 1.867GHz and 2.4GHz

	EDP (Js in million)			EDP % gain	
	1.6 GHz	1.867 GHz	2.4 GHz	1.6 vs 2.4 GHz	1.867 vs 2.4 GHz
FT-B	111.85	115.23	129.78	13.81	11.21
MG-C	5.75	5.68	6.2	7.21	8.40
CG-C	124.6	135.33	133.845	6.90	-1.11
EP-C	6.55	5.24	3.81	-71.74	-37.36
IS-C	35.97	3.64	4.03	10.76	9.68

Table V : EDP gain of UPC-NAS Benchmarks

8 Comparing runtime characteristics of NAS applications at different frequencies

We ran the applications at frequencies 1.6 GHz and 2.4 GHz and collected breakup runtimes of individual subroutines. This also helps us in determining the UPC subroutine that should be instrumented with DVFS mechanism to reduce energy consumption.

By looking at the break up runtimes of individual subroutines, we can find that memget operation consumes most of the runtime during the execution.

For FT benchmark, memget operation consumes approximately 90% of total runtime. We can observe that the percentage change in runtime (Figure 32) of the remote memory fetch operation does not change significantly upon reduction of frequency whereas every other subroutine shows increase in runtime. This fact helps us in recognizing memget as a good candidate for DVFS for NAS-FT since we can safely reduce frequency and thus energy without affecting runtime severely.

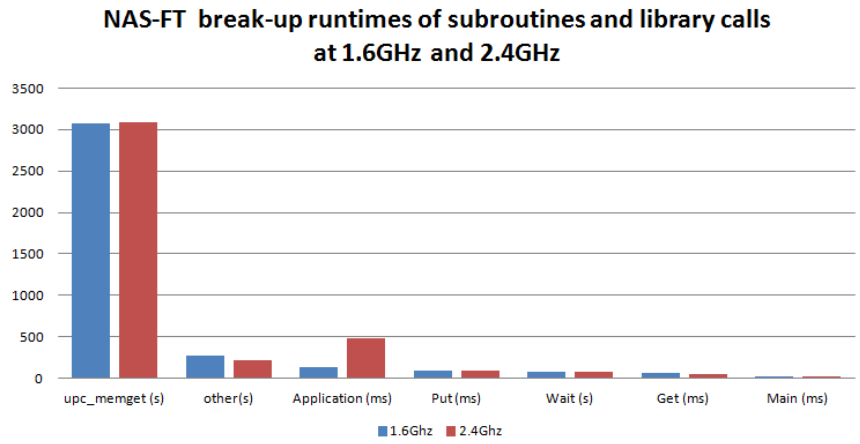

Figure 30: Runtime Breakup of FT at 1.6 GHz and 2.4 GHz

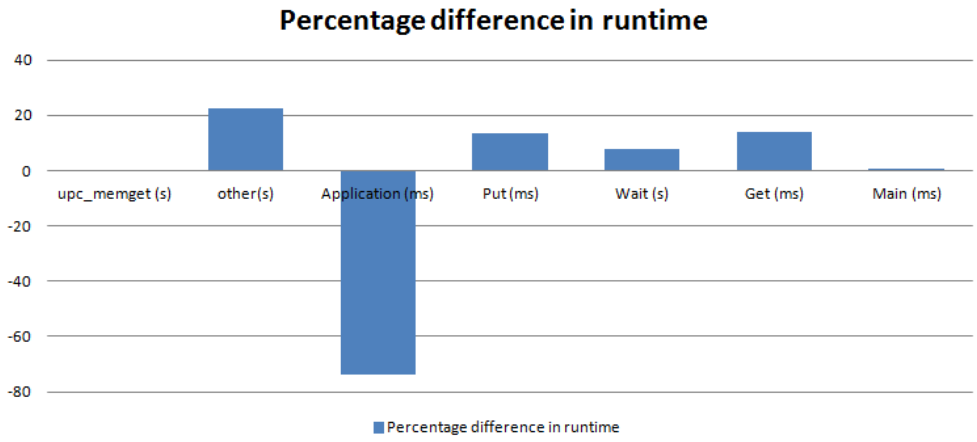

Figure 31 : Percentage Difference in runtimes of FT at 1.6 GHz and 2.4 GHz

8.1 Runtime break-up of MG at 1.6GHz and 2.4GHz

NAS-MG consumes 45% of runtime on memput operations. **Figure 34** shows that the percentage difference for memput operation is negligible- and thus performing DVFS during memput will not adversely affect runtime. We can see that wait operation consumes lesser percentage time (negative runtime loss) in the 1.6 GHz than the 2.4 GHz run. This means that NAS-MG achieves better load balance at 1.6 GHz compared to 2.4 GHz.

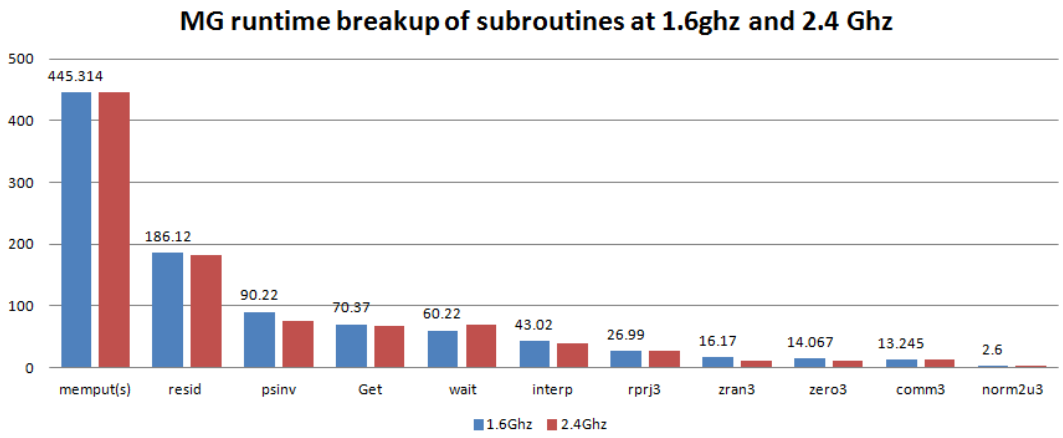

Figure 32 : Runtime Breakup of MG at 1.6 GHz and 2.4 GHz

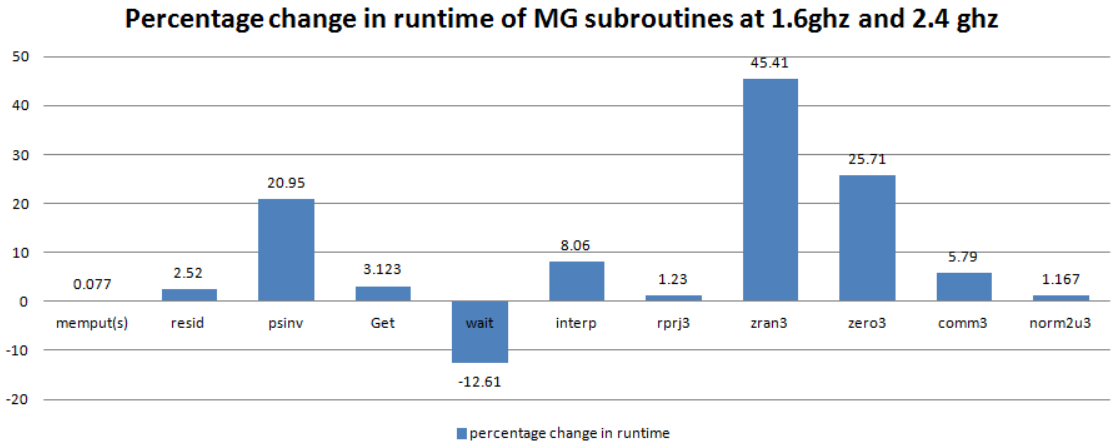

Figure 33 : Percentage Difference in Runtimes of MG at 1.6 GHz and 2.4 GHz

9 Runtime System to perform DVFS for PGAS Applications

Our second objective of research was to build an application-aware runtime DVFS system that controls CPU frequency during execution of application that would result in power savings.

From our previous experiments and results above, we can deduce that performing DVFS on **memget** and **wait** operation would not affect the runtime severely. We inserted DVFS mechanisms inside the UPC-Memget and UPC-Wait operation subroutines. When the memget or wait was called by the application, the DVFS mechanism would reduce the frequency, causing the application to consume lesser power while memget/wait was in execution. We implemented this mechanism with different policies. The policies are based on the statistics of the duration of memget or a wait operation on a thread.

9.1 Algorithm/techniques

To introduce policies to our DVFS mechanism, we used the statistical data of the time spent in individual memget and wait operations.

From our experiments we found that the wait operations take a duration of the order of hundreds of micro-seconds(10^{-4}) while memget operations are of the order of hundreds of milli-seconds (0.1). We also computed the overhead in carrying out DVFS mechanism and found it to be of the order of tens of micro-seconds duration.

We also studied the pattern of time spent in memget and wait operations .We found that the the time spent on wait operation or a memget over subsequent calls falls under the same range. For example if a thread waits on a barrier for greater than 0.1ms, the subsequent calls to “wait“ operation by the same thread would consume same order of time repeatedly. We make use of this fact in our policy.

The policy we implemented makes sure that DVFS mechanism is triggered only if the time spent in previous wait/memget call was greater than the overhead time in performing DVFS.

9.2 Algorithm

The "threshold" value should be atleast equal to the DVFS-Overhead time, lest the mechanism would get triggered for wait durations of very small periods and the energy saved during this would not amortize with the energy spent in performing the DVFS.

```
If (wait_time_prev_iteration > threshold)
{
            PerformDVFS() ;                        //Push down frequency
}
Start = Start_timer();
upc_wait() ;
Stop = Stop_timer();
PerformDVFS();               //Bring up the frequency to Highest value
wait_time_prev_iteration = (Stop – Start);
```

We performed tests with different values of thresholds to see how often DVFS would get triggered for **wait** and **memget** operation.

9.3 Results and analysis

9.3.1 NAS-FT

For FT benchmark we found that a **wait** operation would trigger DVFS 99% of times if the threshold value was set to 0.1ms, 97% when threshold was set to

0.01s and 89% when it was set to 0.1s. However since memget operations are always of the order of hundred-milliseconds duration, DVFS always gets triggered-demonstrating that it was ***always*** beneficial to perform DVFS on a memget.

Since the duration of wait operation is lesser than that of memget operation, and for the reason that DVFS happens on fewer occasions for a wait operation, the contribution to power savings by wait operation is lot lesser than that of memget operation. We evaluated the percentage of contribution of wait and memget operation for different values of threshold on the FT benchmark.

When the threshold on wait is 0.1ms, it contributes to 9% of the total savings where as memget contributes to 91% of the power saved. When the threshold is increased to 0.01s, the power saved due to wait operation decreases to almost 4% and goes upto 7.5% when the threshold is further increased to 0.1s.

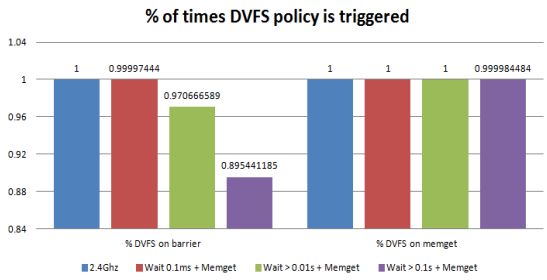

Figure 34

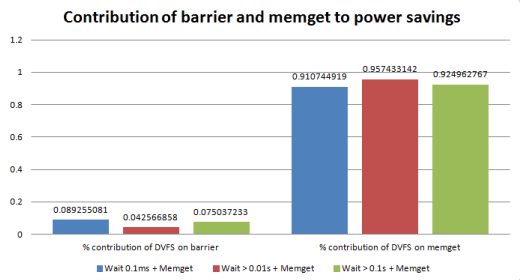

Figure 35

We can see reduction in EDP after DVFS, where the maximum energy savings is achieved when threshold is 0.01s.

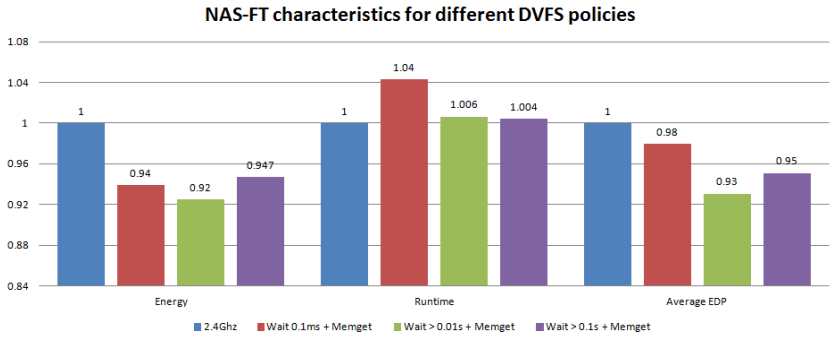

Figure 36

NAS-FT DVFS Results

9.3.2 NAS-MG

MG benchmark provides almost 5% energy savings when threshold is 0.1 ms. Increasing the threshold degrades the energy saving percentage due to the reduction in fraction of times DVFS is triggered on a barrier. The two policies produce similar results for MG benchmark. This shows that duration of **memput** calls are of the same range as of **wait** calls.

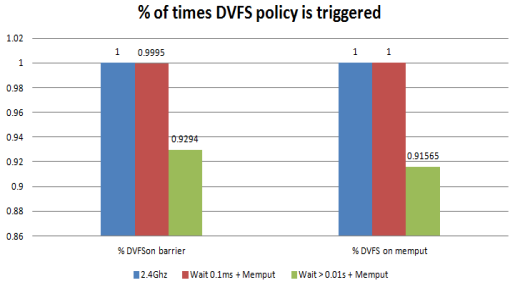

Figure 37

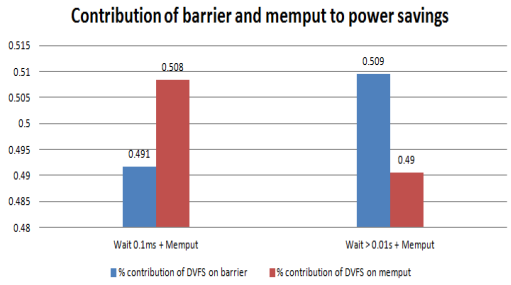

Figure 38

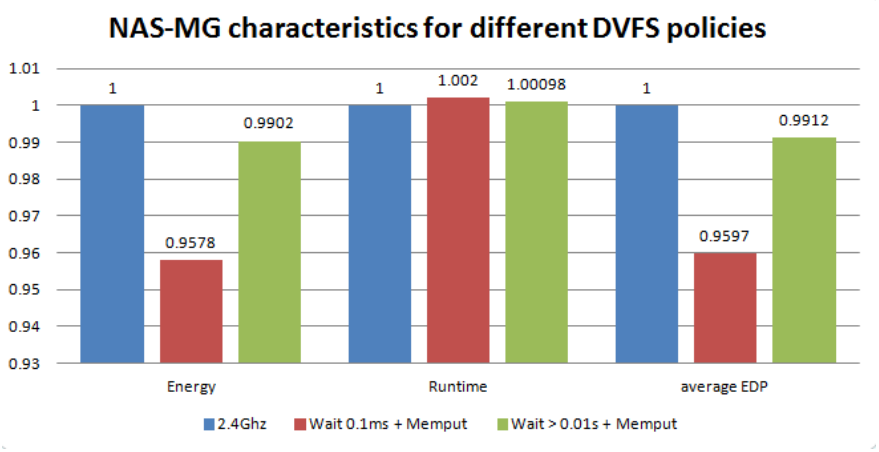

Figure 39

NAS-MG DVFS Results

9.3.3 NAS-CG

While CG benchmark registers no energy savings but consumes extra energy due to extra runtime caused by DVFS mechanism. For NAS CG Benchmark, energy consumed increases after performing DVFS - showing that the current policy is not effective.

The reason for this is that for CG, **wait** and **memget** operations contribute almost equally in runtime. Infact total time consumed by wait operation is greater than total time consumed by memget operation. Also in the earlier graph we saw that wait operation consumes 10% greater time at 1.6Ghz compared to 2.4Ghz. Hence runtime of the execution where DVFS is performed, exceeds the runtime when no DVFS is performed (ie frequency constant at 2.4Ghz), hence consuming more energy.

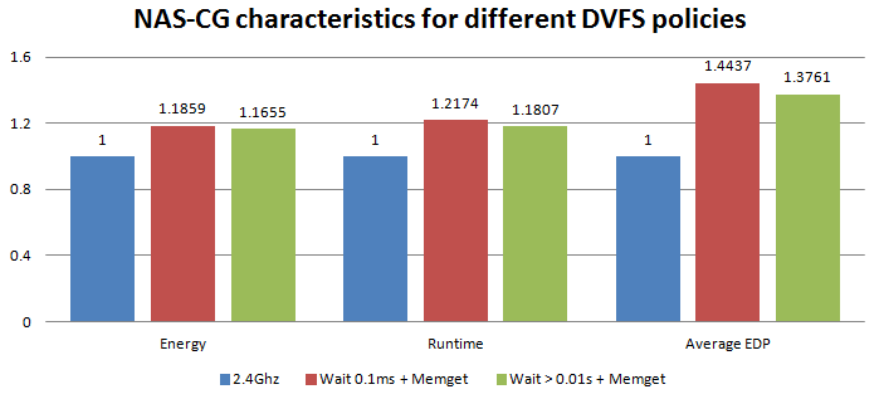

Figure 40

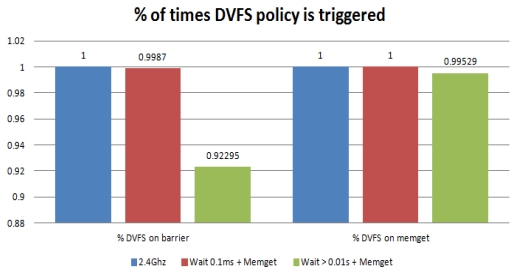

Figure 41

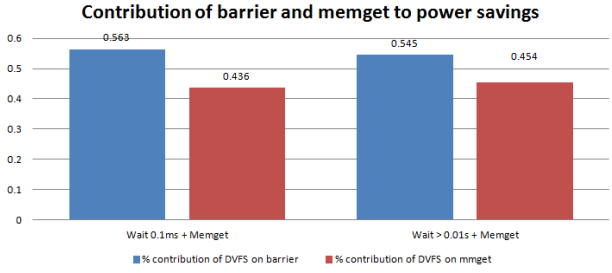

Figure 42

NAS-CG DVFS Results

	Energy Gain (%)				Runtime Loss (%)				EDP Gain (%)			
	1.6GHz	*Policy I*	*Policy II*	*Policy III*	*1.6GHz*	*Policy I*	*Policy II*	*Policy III*	*1.6GHz*	*Policy I*	*Policy II*	*Policy III*
FT-B	15.17	7.64	9.04	6.86	1.6	4.32	0.63	0.41	13.81	2.03	6.93	4.91
MG-C	10.47	4.21	0.97	-	3.64	-0.12	-0.098	-	7.21	4.03	0.86	-

Table VI

Policy I – DVFS if "wait" duration > 0.1ms and DVFS on memget

Policy II – DVFS if "wait" duration > 0.01s and DVFS on memget

Policy III – DVFS "wait" duration > 0.1s and DVFS on memget

10 Conclusions and Future Work

In this paper, we studied the potential impact of deterministic application-centric power control at the device level on the overall energy efficiency of the system. Specifically, we analyzed the power consumption of a node according to the usage of its processor, memory and storage subsystem, and the NIC. Based on a power and latency model and workload profiling, we have developed the PAPM algorithm, which attempts to improve energy efficiency with little or no performance loss. We evaluated our approach with simulations and actual executions to validate them. The simulations used empirical data that was obtained from executions in real hardware to quantify the potential energy savings.

The results stated that using application-aware subsystem power control can save additional energy without significant penalty in performance. Furthermore, the results showed that combining PAPM with using low power devices (e.g. SSD technology) can increase remarkably the potential to improve the energy efficiency of the system. It allows cross layer optimizations, such as application-aware mapping across the data center. We conclude that power management at the subsystem level cannot be neglected due to the increasing requirements of energy efficiency optimization in large-scale data centers. We believe that our proposed predictive application-aware power management approach has sufficient potential to tackle this problem.

It motivates us to investigate autonomic approaches for application-aware aggressive power management rather than the deterministic approach presented in this paper. Since HPC applications usually follow iterative patterns, we plan to

profile the applications on run time and, then predict the optimal subsystems' power configuration to be applied for the following iterations. We also can conclude that current and ongoing technologies such as SSD disk drives or memories that allow DVS must be adopted and supported in large-scale data centers to enhance global energy optimizations.

In conjunction with application aware predictive algorithms we also propose a cross layer and cross function predictive subsystem level power management system as future research directions.

We also analysed UPC-NAS applications with respect to power, studied their traces and instrumented the UPC library with a feature that dynamically takes decisions to perform DVFS.

We arrived at suitable policies for performing DVFS for each application that are optimal from the point of view of power consumption.

10.1 Scope for Future Work

1. **To perform further scalabilty tests**:- Benchmark UPC-NAS applications and application codes with different cluster configurations by using varying number of processes and nodes.
2. **Trying a different policy for NAS-CG benchmark** :- CG Benchmark has a better chance of showing power savings when DVFS is performed only on Memget operation.
3. **Using Global Arrays implementation** - which is an implementation of PGAS programming model and explore what kind of policies would work.

11 Bibliography

1. *Large-scale distributed systems at Google: current systems.* **Dean, J.** s.l. : 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware, 2009.

2. Mission Possible -- Greening the HPC Data Center. *HPC Wire.* [Online] 2009 йил. http://www.hpcwire.com/hpcwire/2009-08-31/mission_possible_--_greening_the_hpc_data_center.html.

3. **K. W. Cameron, R. Ge, and X. Feng.** High-performance, power-aware distributed computing for scientific applications. *Computer.* 2005 йил, Vol. 38.

4. Dynamic frequency scaling. *Wikipedia.* [Online] http://en.wikipedia.org/wiki/Dynamic_frequency_scaling.

5. L i n u x C P U F r e q. *www.mjmwired.net.* [Online] http://www.mjmwired.net/kernel/Documentation/cpu-freq/governors.txt.

6. *cpuidle-Do nothing efficiently...* **V. Pallipadi, S. Li, and A. Belay.** s.l. : Ottawa Linux Symposium, 2007.

7. **Zhu, Y. Liu and H.** *A survey of the research on power management techniques for high-performance systems.* 2010 йил.

8. *Power and thermal management in the Intel Core Duo processor.* **A. Naveh, E. Rotem, A. Mendelson, S. Gochman, R. Chabukswar,K. Krishnan, and A. Kumar,.** s.l. : Intel Technology Journal Tech. Rep., , 2006 йил.

9. *Processor power management features and process scheduler: do we need to tie them together?".* **V. Pallipadi and S. B. Siddha.** s.l. : LinuxConf Europe, 2007.

10. *Getting maximum mileage out of tickless.* **S. Siddha, V. Pallipadi, and A. V. D. Ven.** s.l. : Ottawa Linux Symposium, 2007.

11. *Advanced Configuration & Power Interface (ACPI).* 2009.

12. *Just in time dynamic voltage scaling: exploiting inter-node slack to save energy in MPI programs.* **Kappiah, Nandini and Freeh, Vincent W. and Lowenthal, David K.** s.l. : ACM/IEEE conference on Supercomputing (SC'05), 2005.

13. *Adaptive, transparent frequency and voltage scaling of communication phases in MPI programs.* **Lim, Min Yeol and Freeh, Vincent W. and Lowenthal, David K.** s.l. : ACM/IEEE conference on Supercomputing (SC'06), 2006.

14. *Bounding energy consumption in large-scale MPI programs.* **Rountree, Barry and Lowenthal, David K. and Funk, Shelby and Freeh, Vincent W. and de Supinski, Bronis R. and Schulz, Martin.** s.l. : ACM/IEEE conference on Supercomputing (SC'07), 2007.

15. *Adagio: making DVS practical for complex HPC applications.* **Rountree, Barry and Lownenthal, David K. and de Supinski, Bronis R. and Schulz, Martin and Freeh, Vincent W. and Bletsch, Tyler.** s.l. : 23rd international conference on Supercomputing (ICS'09).

16. *Exploring the energy-time tradeoff in MPI programs on a power-scalable cluster.* **Freeh, Vincent W. and Pan, Feng and Kappiah, Nandini and Lowenthal, David K. and Springer, Rob.** s.l. : 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05), 2005.

17. *Using multiple energy gears in MPI programs on a power-scalable cluster.* **Freeh, Vincent W. and Lowenthal, David K.** Chicago, IL, USA : ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP'05), 2005.

18. **Feng, Kirk W. Cameron and Rong Ge and Xizhou.** High-performance, power-aware distributed computing for scientific applications. *Computer.* 2005 йил, Vol. 38.

19. **Horvath, Tibor and Skadron, Kevin.** Multi-mode energy management for multi-tier server clusters. *International conference on Parallel Architectures and Compilation Techniques (PACT'08).* 2008 йил.

20. **Ranganathan, Parthasarathy and Leech, Phil and Irwin, David and Chase, Jeffrey.** Ensemble-level power management for dense blade servers. *SIGARCH Comput. Archit. News.* 2006 йил, Vol. 34.

21. *Cluster level feedback power control for power optimization.* **Chen, Xiaorui Wang and Ming.** Salt Lake City, UT, USA : IEEE 14th International Symposium on High Performance Computer Architecture (HPCA), 2008.

22. *Process cruise control: event-driven clock scaling for dynamic power management.* **Weissel, Andreas and Bellosa, Frank.** Grenoble, France : International conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'02), 2002.

23. *A power-aware run-time system for high-performance computing.* **Hsu, Chung-Hsing and Feng, Wu-Chun.** s.l. : ACM/IEEE conference on Supercomputing (SC'05), 2005.

24. *Phase-aware adaptive hardware selection for power-efficient scientific computations.* **Malkowski, Konrad and Raghavan, Padma and Kandemir, Mahmut and Irwin, Mary Jane.** s.l. : International Symposium on Low Power Electronics and Design (ISLPED'07), 2007.

25. *Adagio: making DVS practical for complex HPC applications.* **Rountree, Barry and Lownenthal, David K. and de Supinski, Bronis R. and Schulz, Martin and Freeh, Vincent W. and Bletsch, Tyler.** s.l. : 23rd International conference on Supercomputing (ICS'09), 2009.

26. *Energy conservation in heterogeneous server clusters.* **Heath, Taliver and Diniz, Bruno and Carrera, Enrique V. and Jr., Wagner Meira and Bianchini, Ricardo.** Chicago, IL, USA : ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP'05), 2005.

27. *pMapper: power and migration cost aware application placement in virtualized systems.* **Verma, Akshat and Ahuja, Puneet and Neogi, Anindya.** Leuven, Belgium : 9th ACM/IFIP/USENIX International Conference on Middleware (Middleware'08), 2008.

28. *VirtualPower: coordinated power management in virtualized enterprise systems.* **Nathuji, Ripal and Schwan, Karsten.** Stevenson, Washington, USA : ACM SIGOPS Symposium on Operating Systems Principles (SOSP'07), 2007.

29. *Power-aware scheduling of virtual machines in DVFS-enabled clusters.* **He, G. Laszewski and L. Wang and A. J. Younge and X.** s.l. : IEEE International Conference on Cluster Computing and Workshops, 2009.

30. *Energy-oriented compiler optimizations for partitioned memory architectures.* **Delaluz, V. and Kandemir, M. and Vijaykrishnan, N. and Irwin, M. J.** San Jose, California, USA : International conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'00), 2000.

31. **Delaluz, Victor and Kandemir, Mahmut and Vijaykrishnan, N. and Sivasubramaniam, Anand and Irwin, Mary Jane.** Hardware and Software Techniques for Controlling DRAM Power Modes. *IEEE Trans. Comput.* 2001 йил, Vol. 50.

32. *Automatic data migration for reducing energy consumption in multi-bank memory systems.* **Delaluz, Victor and Kandemir, M. and Kolcu, I.** New Orleans, Louisiana, USA : 39th annual Design Automation Conference (DAC'02), 2002.

33. *Positional adaptation of processors: application to energy reduction.* **Huang, Michael C. and Renau, Jose and Torrellas, Josep.** San Diego, California : 30th annual International Symposium on Computer Architecture (ISCA'03).

34. *System level multi-bank main memory configuration for energy reduction.* **Auguin, Hanene Ben Fradj and Cecile Belleudy and Michel.** s.l. : International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS), 2006.

35. *Limiting the power consumption of main memory.* **Diniz, Bruno and Guedes, Dorgival and Meira,Jr., Wagner and Bianchini, Ricardo.** San Diego, California, USA : 34th annual International Symposium on Computer Architecture (ISCA'07), 2007.

36. *A comprehensive approach to DRAM power management.* **Lin, Ibrahim Hur and Calvin.** Salt Lake Citi, UT, USA : 14th International Conference on High-Performance Computer Architecture (HPCA).

37. *Analysis of trade-off between power saving and response time in disk storage systems.* **Tsao, Doron Rotem and Ekow Otoo and Shih-Chiang.** s.l. : Fifth Workshop on High-Performance Power-Aware Computing (HPPAC'09) with IPDPS'09, 2009 йил.

38. *Energy conservation techniques for disk array-based servers.* **Pinheiro, Eduardo and Bianchini, Ricardo.** Malo, France : 18th International conference on Supercomputing (ICS'04), 2004.

39. **Pinheiro, Eduardo and Bianchini, Ricardo and Dubnicki, Cezary.** Exploiting redundancy to conserve energy in storage systems. *SIGMETRICS Perform. Eval. Rev.* 2006 йил, Vol. 34.

40. *Massive arrays of idle disks for storage archives.* **Colarelli, Dennis and Grunwald, Dirk.** s.l. : ACM/IEEE conference on Supercomputing (SC'02), 2002.

41. *An energy-efficient reliability model for parallel disk systems.* **Qin, S. Yin and X. Ruan and A. Manzanares and X.** s.l. : IEEE International Conference on Cluster Computing and Workshops, 2009.

42. *Empirical analysis on energy efficiency of flash-based SSDs.* **Urgaonkar, Euiseong Seo and Seon Yeong Park and Bhuvan.** San Diego, USA : 1st Workshop on Power Aware Computing and Systems (HotPower'08), 2008.

43. *Cross-component energy management: joint adaptation of processor and memory.* **Li, Xiaodong and Gupta, Ritu and Adve, Sarita V. and Zhou, Yuanyuan.** s.l. : ACM Trans. Archit. Code Optim., 2007 йил, Vol. 4.

44. *Memory-aware energy-optimal frequency assignment for dynamic supply voltage scaling.* **Cho, Youngjin and Chang, Naehyuck.** Newport Beach, California, USA : International Symposium on Low Power Electronics and Design (ISLPED'04), 2004.

45. *Performance directed energy management for main memory and disks.* **Li, Xiaodong and Li, Zhenmin and Zhou, Yuanyuan and Adve, Sarita.** s.l. : ACM Transactions on Storage, 2005 йил, Vol. 1.

46. **Sebastien Chauvin, Proshanta Saha, Francois Cantonnet, Smita Annareddy, Tarek El-Ghazawi.** *UPC Manual.* s.l. : The George Washington University - High Performance Computing Laboratory.

47. *A performance analysis of the Berkeley UPC compiler.* **Wei-Yu Chen, Dan Bonachea, Jason Duell, Parry Husbands, Costin Iancu, and Katherine Yelick.** s.l. : In Proceedings of the 17th Annual International Conference on Supercomputing (ICS'03), June 2003.

48. *UPC Performance and Potential: A NPB Experimental Study.* **Cantonnet, Tarek El-ghazawi and Francois.** s.l. : In Supercomputing2002 (SC2002), IEEE Computer Society, 2002.

49. *UPC Benchmarking Issues.* **Chauvin, Tarek El-Ghazawi and Sebastien.** 2001.

50. *The UPC Memory Model: Problems and Prospects.* **Wallace, William Kuchera and Charles.** 2004.

51. *Adaptive, transparent frequency and voltage scaling of communication phases in MPI programs.* **Lim, Min Yeol and Freeh, Vincent W. and Lowenthal, David K.** s.l. : ACM/IEEE conference on Supercomputing (SC'06), 2006.

52. *CPU MISER: A Performance-Directed, Run-Time System for Power-Aware Clusters.* **Ge, Rong and Feng, Xizhou and Feng, Wu-chun and Cameron, Kirk W.** s.l. : ICPP '07: Proceedings of the 2007 International Conference on Parallel Processing, 2007.

53. *Augmenting RAID with an SSD for energy relief.* **Noh, Hyo J. Lee and Kyu H. Lee and Sam H.** San Diego, USA : 1st Workshop on Power Aware Computing and Systems (HotPower'08), co-located with OSDI 2008, 2008.