RUNTIME METHODS TO IMPROVE ENERGY EFFICIENCY IN SUPERCOMPUTING APPLICATIONS

Sridutt Bhalachandra

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2018

Approved by:

Allan K. Porterfield

Jan F. Prins

Stephen L. Olivier

Robert J. Fowler

Montek Singh

ABSTRACT

Sridutt Bhalachandra: Runtime Methods to Improve Energy Efficiency in Supercomputing Applications
(Under the direction of Allan K. Porterfield and Jan F. Prins)

Energy efficiency in supercomputing is critical to limit operating costs and carbon footprints. While the energy efficiency of future supercomputing centers needs to improve at all levels, the energy consumed by the processing units is a large fraction of the total energy consumed by High Performance Computing (HPC) systems. HPC applications use a parallel programming paradigm like the Message Passing Interface (MPI) to coordinate computation and communication among thousands of processors. With dynamically-changing factors both in hardware and software affecting energy usage of processors, there exists a need for power monitoring and regulation at runtime to achieve savings in energy.

This dissertation highlights an adaptive runtime framework that enables processors with core-specific power control by dynamically adapting to workload characteristics to reduce power with little or no performance impact. Two opportunities to improve the energy efficiency of processors running MPI applications are identified - *computational workload imbalance* and *waiting on memory*. Monitoring of performance and power regulation is performed by the framework transparently within the MPI runtime system, eliminating the need for code changes to MPI applications. The effect of enforcing power limits (capping) on processors is also investigated.

Experiments on 32 nodes (1024 cores) show that in presence of workload imbalance, the runtime reduces Central Processing Unit (CPU) frequency on cores not on the critical path, thereby reducing power and hence energy usage without deteriorating performance. Using this runtime, six MPI mini-applications and a full MPI application show an overall 20% decrease in energy use with less than 1% increase in execution time. In addition, the lowering of frequency on non-critical cores reduces run-to-run performance variation and improves performance. For the full application, an average speedup of 11% is seen, while the power is lowered by about 31% for an energy savings of up to 42%. Another experiment on 16 nodes (256 cores) that are power capped also shows performance improvement along with power reduction. Thus, energy optimization can also be a performance optimization. For applications that are limited by memory access

times, memory metrics identified facilitate lowering of power by up to 32% without adversely impacting performance.

To my beloved family, the eternal source of my strength and inspiration.

ACKNOWLEDGMENTS

A dissertation is a culmination of the efforts and support of many individuals, and it is no different in my case. Many people have helped me along the way, and I would like to acknowledge and thank them for their contributions.

First and foremost, I would like to thank my advisor, Dr. Allan Porterfield for giving me the opportunity to work with him and guiding me through the process. Over the last six years, his faith and support in trying times have inspired and motivated me to put forth my best. I also had the privilege to be co-advised by Prof. Jan Prins who gave me the freedom to pursue my research with Allan, while at the same time providing invaluable support, feedback, and encouragement. I am grateful to Jan for doing everything he can to facilitate my research at the Renaissance Computing Institute (RENCI).

I thank Dr. Stephen Olivier, who has been a mentor to me since my summer internship at Sandia National Laboratories and served on my thesis committee. I am grateful to Stephen for his candid feedback and advice on both my research and professional career. I am thankful to Dr. Robert Fowler who showed faith and hired me as a Research Assistant with Allan at RENCI and provided continuous guidance. His constructive feedback as a committee member has greatly helped in improving the quality of this thesis. I also thank Prof. Montek Singh for agreeing to be on my thesis committee and for his advice and support since my first Preliminary Research Presentation and Exam (PRP).

I acknowledge and thank the countless professors, the non-teaching staff at CS department/UNC and colleagues at RENCI and other research groups and institutions. I also would like to call out the System Administrators at RENCI, Sandia, and UNC who facilitated access to various machines for experiments. I thank Dr. Barry Rountree for giving me the opportunity to be a summer intern at Lawrence Livermore National Laboratory. I acknowledge the contribution of various funding sources that have supported me – U.S. Department of Energy XPRESS project under Contract DE-SC0008704, Office of Science SciDAC SUPER Institute on grant DE-SC0006925, 2017 UNC Dr. Tom V. Morris Summer Fellowship and travel awards to conferences.

My graduate life has been enjoyable and purposeful because of the many amazing people I have met. They have contributed in different ways with some becoming friends, others mentors and a few both. I would like to thank Karthik Bangera, Ankur Bahri, Kishore Rathinavel, Diptorub Deb, Neha Gholkar, Wei Wang, Rajesh Reddy, Abhinav Golas, Anirban Mandal, Matthew Dosanjh, Noah Evans, Taylor Groves, Ryan Grant, Ronald Brightwell, Aniruddha Marathe, Tapasya Patki, Ravikiran Janardhana, Siddharth Padmanabhan, Praneeth Chakravarthula, Shubham Gupta and Ravish Mehra.

I would like to thank my family for all the love and support. My parents, Bhalachandra and Shakila, have given me more than I could have asked for. It was my father who first instilled in me the idea of pursuing a Ph.D. and gave me the freedom to chase my own dreams. My mother has always showered me with affection and helped me in every way. My mother-in-law, Pancharatna, a professor herself, empathized with the day-to-day of my graduate life and encouraged me to achieve my goals. Last but not the least, my wife, Apoorva has always been there for me and supported me in the toughest of times. It was due to her financial support and motivation that I completed my research and did not give up.

In the end, I thank my extended family and everyone else who might have contributed towards this thesis that I failed to mention.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ACR	Adaptive Core-specific Runtime
AMR	Adaptive Mesh Refinement
CPU	Central Processing Unit
CDP	Coarse-grained Dynamic Policy
DCT	Dynamic Concurrency Throttling
DDCM	Dynamic Duty Cycle Modulation
DRAM	Dynamic Random Access Memory
DVFS	Dynamic Voltage and Frequency Scaling
FDP	Fine-grained Dynamic Policy
HPC	High Performance Computing
LLC	Last Level Cache
MPI	Message Passing Interface
MSR	Model Specific Register
OS	Operating System
PMPI	Profiling Message Passing Interface
RCR	Resource Centric Reflection
RAPL	Running Average Power Limit
TDP	Thermal Design Power
TOR	Table Of Request
TORo	Table Of Request Occupancy

CHAPTER 1: Introduction

In the past few decades advances in large-scale computing have revolutionized many basic fields of science. The impact of this progress however, has not been restricted to the fields originally intended, but has helped to lay the foundations of an age centered around information and technology. In a large-scale computer a massive number of parallel processors work together to solve a problem in a fraction of time it would take on a single processor. Suppose a processor takes at least time *t* to solve a problem (sequential solution), then a parallel solution utilizing *p* processors can improve the sequential solution by at most a factor of *p* – Fundamental Law of Parallel Computation (Gustafson, 1988).

Traditionally, the scientific and commercial problems that used large-scale computing were called compute-bound problems since I/O bound problems involving massive data were more conducive to better data transmission technology than processing capability. In recent times however, more and more scientific problems require the aid of efficient memory sub-systems and I/O capabilities to be solved with closer to peak performance on modern HPC systems. This makes achieving high performance even more challenging in current supercomputers as a poor provisioning can lead to very low percentages of peak performance attainable.

The compute-bound problems are generally superlinear, typically exhibiting sequential time complexity in the range $O(n^2)$ to $O(n^4)$ for problems of size n. The reason $O(n^4)$ problems are common in science and engineering is that they often model physical systems with three spatial dimensions developing in time, for example a storm surge. A frequent challenge with these problems is not to solve a fixed-size instance of the problem faster, but rather to solve larger instances within a fixed time budget. Using this understanding along with the Fundamental Law of Parallel Computation, Lawrence Snyder (Snyder, 1986) observed that superlinear problems can be improved only sublinearly by parallel computation. Thus, parallel computation only offers modest potential benefit – Corollary of Modest Potential. This corollary along with the increasing complexity in current High Performance Computing (HPC) systems further mandates a need for efficiency in parallel systems as technological advancements are approaching their inevitable physical limits.

1.1 Energy efficiency in High Performance Computing

Advances in many branches of science and engineering increasingly depend on ever larger and more detailed numerical simulations performed on high performance computing (HPC) systems. The days of ever increasing single core performance have, however, come to an end as power dissipation and transistor scaling are reaching fundamental limits. With 10^5 parallel processors now present in top supercomputers, power consumption is increasingly becoming the limiting factor. These machines, with an aggregate performance of 10^{15} (Peta) floating point operations per second (FLOPS), already consume in excess of 15MW (Fu et al., 2016), with energy costs of over $10M dollars a year. To make the next generation of supercomputers feasible, the Exascale Computing study (Kogge et al., 2008) set a definitive power challenge to deliver exaFLOPS using just 20 MW. A commensurate increase in power is no longer feasible with the performance needing to improve over 10x, while power less than doubles.

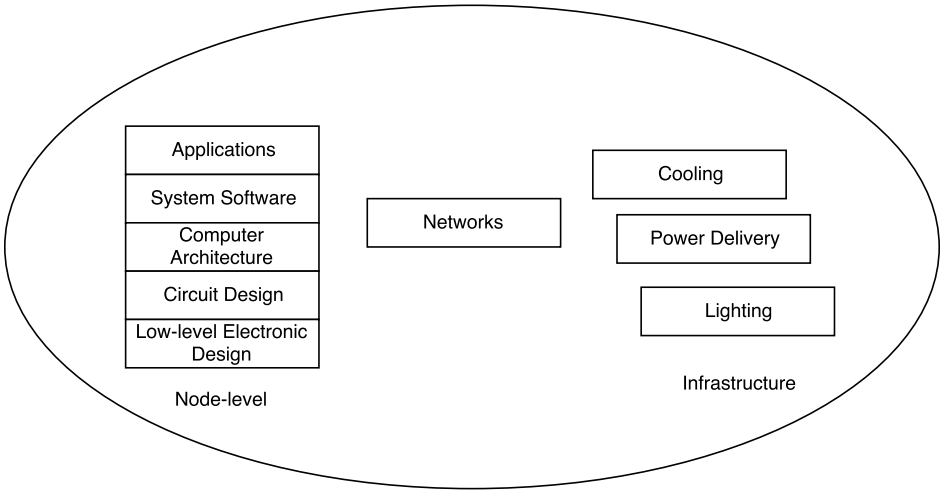


Figure 1.1: Improving energy efficiency in a supercomputer

The scope for improving energy efficiency of a supercomputer exists at many levels as shown in Figure 1.1. The hardware community is exploring alternative technologies like dark silicon to improve chip efficiency, while also investing time and energy to come up with better micro-architectures, interconnects, memory technologies among countless other improvements. The infrastructure research is focussing on designing energy-efficient buildings, power delivery infrastructure and cooling solutions to name a few. A lot can be achieved on the software side within a node to bridge the gap. HPC applications more often than not exhibit workload imbalances, wait on memory or poorly utilize the available resources like Central Processing

Unit (CPU), memory and network. These in-efficiencies present numerous opportunities to improve energy efficiency in system software.

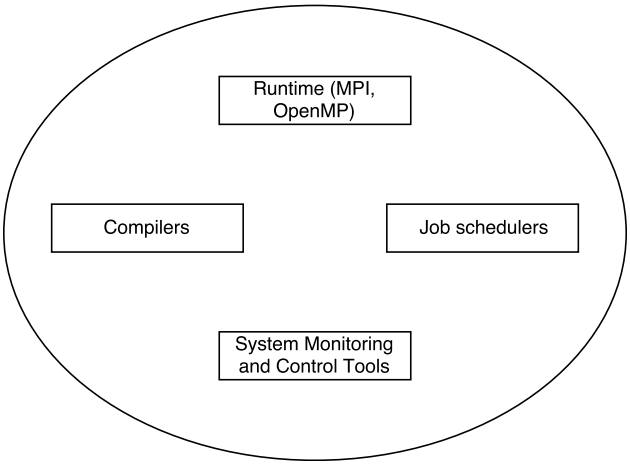


Figure 1.2: Improving energy efficiency in system software

In software (Figure 1.2), multiple hardware components like CPU, memory, storage and network interfaces can be targeted within a conventional compute node of a supercomputer to improve energy efficiency. Graphics Processing Units (GPU) too are making their way into newer systems and consume large portion of the energy budget. Network fabric, interconnects, switches and other components outside of a node too can be controlled through software. With more than 30% of energy being consumed in both idle and active operations (David et al., 2010), CPU is an ideal target for efforts involving software energy efficiency research.

With tighter power budgets likely in the near future, HPC researchers are searching for ways to improve performance with the available power budget. *Overprovisioning* of processor nodes (Patki et al., 2013; Sarood et al., 2014; Marathe et al., 2015; Gholkar et al., 2016) using hardware enforced power bounds is one likely alternative. An overprovisioned system is a system with more capacity (nodes), with the stipulation that we cannot simultaneously power all components at peak power because of strict power constraints. The maximum power budget is then met with additional nodes (with attendant memory and bandwidth) by constraining power for each node below the Thermal Design Power (TDP). This allows a greater number of nodes to be run than normally possible at peak power. In an overprovisioned system, improvements in energy efficiency increase performance by allowing more nodes or a higher effective power limit on actives node without exceeding the power cap. Increased power cap on the active nodes may also lead to improved performance as the cores can potentially run faster. The effect is improvement in the throughput of the system.

1.2 Challenges and Goals

Energy efficiency of computations can be improved by completing a computation in less time or by running the computation at lower power. The state-of-the-art of processor energy savings techniques mainly address problems arising from uneven work distribution, waiting on a resource, mostly memory and poor resource utilization. However, the techniques still faces many challenges:

Static Analysis. Many related studies (Feng et al., 2005; Kamil et al., 2008; Huang and Feng, 2009) make use of static analysis of programs/applications to determine optimal frequency for an application execution. Though these techniques provide useful insights, they are not always accurate as it is difficult to statically predict behavior of a running application as the system factors can change dynamically. As making actual test runs on production machines can be expensive, many a times synthetic benchmarks are used to obtain parameters for static analysis. Such benchmarks may not always be representative of actual application runtime conditions as energy usage of a processor depends both on physical and software factors that are dynamic. Physical variations arise during the fabrication process and induce variations in performance among otherwise identical processors (Rountree et al., 2012). Placement of components can induce variations in performance due to different cooling efficiencies and steady-state temperatures. Software variations are mainly due to varying data locality, compiler optimizations, and the number of threads utilized. These variations are 20% in general and in the extreme case over 2x (Porterfield et al., 2013b). External factors like cooling also contribute to the variation in energy consumed by an application. These variations suggest a need for dynamic power regulation to achieve savings in energy.

Code instrumentation. Many solutions (Ge et al., 2005; Wang et al., 2015) require changes in application code to support instrumentation. While feasible for small experimental codebases, this is a serious problem for production applications involving several decades of software development. Moreover, the addition of complex instrumentations and logic to an application code can affect the performance characteristics and program flow. This necessitates making the solution transparent to the application, and also simple.

Simulation and synthetic benchmarks. Many results (Hsu and Feng, 2005; Kandalla et al., 2010; Rountree et al., 2012; Livingston et al., 2014) are simulated or use synthetic benchmarks. Simulations provide extremely useful insights on systems that are not readily accessible and as a preliminary work pave the way for better solutions. The dynamic nature of HPC systems and applications however, make actual application run results not always conform to the results from simulation. The results from synthetic benchmarks like NAS are

useful as preliminary results, but are not always representative of actual HPC workloads. Therefore, there is a need for research that show results on mini-apps and real-world applications.

Chip-wide effect of power control. Most research (Kappiah et al., 2005; Rountree et al., 2009; Ge et al., 2007; Freeh and Lowenthal, 2005) to regulate energy and performance in CPU have revolved around chip-wide Dynamic Voltage and Frequency Scaling (DVFS). Chip-wide DVFS is not used in production HPC because it is difficult to find applications where slowing the entire multi-core processor does not result in noticeable slowdowns. Until recently[1], DVFS has only allowed frequency and voltage changes that applied to all the cores of a multi-core processor. Slowing the critical path slows execution. Thus, DVFS-centric research has focused on finding situations where the slowdown is greatly outweighed by the energy savings. This chip-wide effect limits the effectiveness of DVFS to be used for fine grained control. The use of chip-wide DVFS also restricts access to higher turbo frequencies for the critical core (core whose performance determines the performance of the application). Chip-wide DVFS can save energy but slows applications, a side-effect unacceptable for most HPC systems as the primary intent of these supercomputers is to improve performance.

1.3 Scope for improving energy efficiency

With the introduction of core-specific voltage regulators in Intel Haswell microarchitecture, new options for software energy control are now available. Each physical core (or 2 logical cores if using Hyper-Threading) can be independently controlled allowing only non-critical threads to have their frequency reduced in software using runtimes like the one discussed in this dissertation. In addition to DVFS, core-specific Software Controlled Clock Modulation has been supported by Intel since Pentium. The effective core frequency is adjusted nearly instantaneously by gating only a fraction of the clock cycles to that core. We call this approach Dynamic Duty Cycle Modulation (DDCM), given that the objective is to match duty cycle of the core to its work dynamically. The fine-grained control allows DDCM to save power effectively for unbalanced applications. As voltage regulators are unaffected, changing clock frequency with DDCM requires less work (and time) than with DVFS (Wang et al., 2015).

This dissertation identifies two opportunities (Figure 1.3) to improve energy efficiency in CPU - *computational workload imbalance* and *waiting for a resource, mostly memory*.

[1]Intel Haswell architecture introduced core specific voltage regulators that allow per-core frequency control.

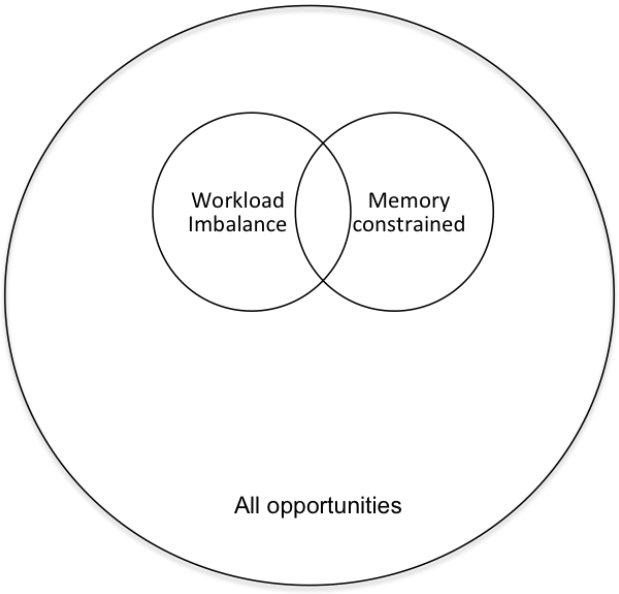


Figure 1.3: Problems addressed in this dissertation

Computational workload imbalance: Multi-core CPUs often have load imbalances between their cores. Applications can have fixed loads or the amount of work can vary between steps. Most HPC applications use parallel programming paradigms that use barriers. A processor that has been allocated less work may finish work faster, but still it has to wait for all other processors to proceed further. Nominally homogeneous computing elements exhibit heterogeneous performance and limiting power increases the performance variation (Porterfield et al., 2015). Transistor thresholds and leakage currents vary within and between wafers during fabrication, resulting in processor chips that require different supply voltages to operate at the design frequency and that therefore consume different amounts of power. Furthermore, the amount of cooling available to a chip depends on its position in the system, resulting in temperature differences that cause additional variations in power consumption in many air-cooled HPC systems. Power and thermal constraints will affect each chip differently causing on-chip mechanisms that control operating frequency to also vary. Performance will thus vary between sockets for even perfectly balanced parallel applications.

Waiting on memory: HPC applications need access to memory often, and sometimes even I/O. Disparity between CPU and memory speeds and the latency of accessing DRAM across the system bus contribute significantly to CPU cycling while waiting on memory and wasting power. Memory operations are not operating system (OS) visible and coarse-grained enough for the hardware circuitry to stall (or switch off) cores and reduce power as memory accesses are serviced without any information from runtime/application.

The hardware circuitry needs the core to idle for a certain amount of time (be coarse-grained enough) before it can be put into a sleep state. The amount of idle time required by the circuity is not always met in the case of HPC applications where the computation and memory sections are mostly alternating in bursts or could even overlap. However with smaller idle times too, there exist opportunities to slow down the CPU and reduce power in software using application-specific information. Also, for certain classes of HPC applications that are memory-bound, reducing the processor speed or using related approaches like CPU throttling for power savings has shown little adverse impact on performance, or in some cases slightly speeds up execution from reduced contention (Porterfield et al., 2013b; Wang et al., 2015).

In this research, the above two problems have been addressed in the Message Passing Interface (MPI) (Gropp et al., 1996) runtime from the available avenues shown in Figure 1.2. MPI includes protocols and semantic specifications for how its features must behave in any implementation. MPI is considered a *de facto* standard for communication among processes of a program that run on a distributed memory system. The following broad considerations have been made in our implementation for MPI:

1. Only pure MPI applications with one process per core are targeted. The techniques presented have not be validated on hybrid programming models that use other shared-memory paradigms like OpenMP in addition to MPI (No MPI+X).

2. The runtime policies and framework have been tested and use features specific to Intel processors. Many of these techniques should be suited for processors from other vendors like IBM, AMD and others.

3. The techniques discussed have not been evaluated on hardware accelerators like Intel Xeon Phi, Field-programmable gate arrays (FPGAs) or GPUs.

These considerations serve as basis for future work as we move towards exascale computing. Any other specific assumptions are addressed when relevant.

1.4 Thesis statement

Measurements and controls local to each core in a multi-core system can on average reduce power consumed by large supercomputing applications at runtime while having little or no adverse impact on execution times.

To prove this thesis, we design and evaluate runtime methods that use measurements and controls local to each core to adapt the power consumption to the workload characteristics. The runtime methods dynamically identify scenarios exhibiting computational workload imbalance and/or memory-constrained behavior and improve energy-efficiency both on conventional and power-limited systems. The research in this dissertation differs from prior work by employing adaptive methods at runtime, and core-specific power controls that have not been readily applied to the above two scenarios. An adaptive runtime framework (MPI shim library) transparent to the application consisting of runtime techniques and user options is developed allowing processors to reduce power with little performance impact. Different core-specific power controls can be employed separately or combined to enhance effectiveness of the framework. In the view of strict power and energy budgets to control the operating costs of supercomputer centers in the future, the effect of enforcement of power limits by external agents on application performance is also studied. For applications with memory related issues, memory metrics that facilitate lowering of power without adversely impacting performance are identified.

1.5 Main Results

The goal of this research is to explore and evaluate methods that improve energy efficiency of HPC applications at runtime. Specifically, the focus is on using core-specific power controls available in recent processor architectures. The major work and ideas in this dissertation are as follows:

- **Dynamic Duty Cycle Modulation in High Performance Computing**: DDCM is shown to be an alternative to save energy, and improve performance in power-capped environments. The fundamental weaknesses of socket-wide DVFS can be overcome with DDCM as it has a per-core control with lower overheads allowing fine-grained core-specific clock frequencies. With power limits on a system, slowing non-critical cores in software is shown to increase the available thermal headroom to the critical core improving performance.

- **Adaptive Core-Specific Runtime (ACR) for Energy Efficiency**: The ACR showcased allows processors with core-specific power control to reduce power with little performance impact by dynamically adapting core frequencies to workload characteristics. Such a runtime could help alleviate heterogeneous processor loads that many future exascale applications will likely have. The MPI framework (shim

library) is transparent to application eliminating any code changes and allows use of multiple power controls (DDCM, per-core DVFS or both).

- **Memory-Metric Policy for Reducing Energy**: The presented characterization of HPC applications identifies a metric that conforms to the memory behavior exhibited by many HPC mini-apps and can be used to construct dynamic runtime polices improving energy efficiency.

We next summarize the results of the research presented in this thesis:

1.5.1 Dynamic Duty Cycle Modulation in High Performance Computing

On Intel processors before Haswell, Dynamic Voltage and Frequency Scaling (DVFS) affects all cores of a multi-core processor. Slowing the critical path slows execution. DVFS-centric research has focused on finding situations where the slowdown is greatly outweighed by the energy savings. Intel also supports Dynamic Duty Cycle Modulation (DDCM) where the effective frequency of each core can be adjusted nearly instantaneously by only gating a fraction of the clock cycles to that core. We show DDCM as an alternative to improve energy-efficiency, and performance in power-capped environments. An adaptive runtime DDCM policy is developed to reduce power in unbalanced MPI applications (Bhalachandra et al., 2015).

On Sandy Bridge systems, the adaptive DDCM policy for MPI was run on synthetic benchmarks and mini-apps – *miniAMR* and *graph500* on single and 16-node configurations. DDCM saved up to 13.5% processor energy on one node and 20.8% (for *miniAMR* with slowdown of less than 1%) on 16 nodes. By applying a power cap, DDCM effectively shifts power consumption between cores and improves overall performance. Performance improvements of 6.0% and 5.6% on one and 16 nodes, respectively, were observed. Saving energy in power-limited systems is also seen to improve performance.

The policy was then validated with production applications like *ADCIRC, WRF and LQCD* (Porterfield et al., 2015). For *ADCIRC* on 16 nodes, energy savings of over 10% with only a 1-3% slowdown is obtained. With a power limit of 50W, one version of the policy executes 3% faster while saving 6% in energy, and a second version executes 1% faster while saving over 10% energy. The effectiveness of DDCM is also shown for OpenMP by *Wang et al., 2015*[2] with savings of 21% in energy and improvement of energy-delay product (EDP) by 16%. With encouraging results on both shared and non-shared memory as well as production applications, DDCM is seen to be a viable alternative to achieve energy efficiency in HPC.

[2]Emphasized citations denote works including me as one of the authors, but not first author

1.5.2 Adaptive Core-Specific Runtime for Energy Efficiency

With addition of core-specific voltage regulators in Intel Haswell, DVFS can now slow down only non-critical cores like DDCM. An Adaptive Core-specific Runtime (ACR) that allows processors with core-specific power control to reduce power with little performance impact by dynamically adapting core frequencies to workload characteristics is developed. A policy to combine the benefit of larger power reduction with DVFS owing to reduction in both voltage and frequency, and the ability of DDCM to lower the frequency beyond the operating range of DVFS is also presented (Bhalachandra et al., 2017a).

This work highlights a generic policy that effectively utilizes core-specific power controls. Our previous work (Section 1.5.1) aimed only at showing the efficacy of DDCM as an alternative to socket-wide DVFS. However, the present work offers a context for comparing DDCM (with its simple per-core hardware implementation and fast switching capability) and DVFS (more complex and costly to implement per-core but with potential for greater savings), and for showing how and when they can be used together. A transparent adaptive runtime framework (library) is implemented that throttles frequencies of cores not on the critical path of an MPI application using either DDCM, per-core DVFS or both.

The framework is validated using six mini-apps (*miniAMR, miniFE, CloverLeaf, HPCCG, AMG, miniGhost*), and a real world application, ParaDis. The evaluation shows an overall 20% improvement in energy efficiency with an average 1% increase in execution time on 32 nodes (1024 cores) using per-core DVFS. An improvement in energy efficiency of up to 39% is obtained with the real world application ParaDis through a combination of speedup (11%) and power reduction (31%). The average improvement in performance seen is a direct result of the reduction in run-to-run variation and running at turbo frequencies.

As Exascale deploys over-provisioned systems that use per core power-limits in day-to-day operations, energy optimizations will be more important. Runtimes such as ACR will either allow more work to be run at one time by using less power or allow single applications to be run faster by allowing a higher power cap on critical cores than non-critical.

1.5.3 Memory-Metric Policy for Reducing Energy

HPC would have been much easier if all the data required could fit in the cache of a processor, but rarely is this true. For certain classes of applications that heavily utilize the memory sub-system, slowing the processor speed or related approaches like CPU throttling has shown little impact on performance, with some

cases showing performance improvement (Wang et al., 2015; Porterfield et al., 2013b). There exists a need for solutions that can dynamically identify such opportunities. Toward this end, we identify metrics to detect applications that are constrained by memory and build an adaptive runtime policy using one of these metric to reduce energy wastage (Bhalachandra et al., 2017b).

We present an experimental memory study on modern CPU architectures, Intel Sandy Bridge and Haswell, to identify opportunities to reduce CPU frequency. Since the Last Level Cache (LLC) is shared, each core has to create a request for a particular memory location that is not in its private cache into the Table of Requests (TOR). Using uncore performance monitoring hardware counters, we identify a metric, *TORo_core*, that captures all valid requests in TOR. This metric detects bandwidth saturation and increased latency in the memory system, and is used in a dynamic policy to modulate per-core power controls.

The policy is evaluated when applied at coarse and fine-grained levels on six MPI mini-applications. The best energy savings with the coarse and fine-grained application of the dynamic policy is 32.1% and 19.5% respectively with a 2% increase in execution time in both cases. On average, the fine-grained dynamic policy yields a 1% speedup while the coarse-grained dynamic policy yields a 3% slowdown.

1.6 Thesis Organization

The rest of the dissertation is organized as follows:

- In **Chapter 2**, the fundamentals of power consumption and control in processor are provided along with a literature review of the prior energy-efficiency techniques, and existing work in the area of runtime optimization for energy.
- **Chapter 3** introduces the policy for using DDCM in presence of unbalanced workloads. The chapter discusses the different steps of the technique, provides implementation details, and presents the results and analysis. It also discusses the results on power-limited systems
- In **Chapter 4**, the DDCM policy is extended to work with core-specific DVFS. Further, a combined policy that uses both DDCM and core-specific DVFS is introduced. An adaptive core-specific runtime is showcased for integrating these policies with user options. It is also shown how optimizing for energy can improve performance

- **Chapter 5** starts with the memory performance study on modern HPC systems. Next, it discusses the characterization of memory activity using LLC counters and presents results for dynamic policies using a LLC metric to reduce power without adversely affecting performance.
- **Chapter 6** concludes the thesis and discusses avenues for future work.

CHAPTER 2: Background and Previous Research

This chapter provides background information about power and energy consumption and control, and introduces the tools and techniques used in this dissertation such as metric measurement tools, and energy control methods. Also, this chapter provides a literature review of the field of Energy Efficient HPC (EEHPC).

2.1 Power consumption in a processor

The power consumption of a modern processor has two key components: static and dynamic (Brooks et al., 2000; Weste and Harris, 2015).

$$P_{total} = P_{static} + P_{dynamic} \tag{2.1}$$

Static power (Idle power) corresponds to the minimum amount of power that is required to turn on and run a processor without active computation/load. As a processor is made up of transistors, the static power dissipation depends on the voltage applied (V_{dd}) and the leakage current ($I_{leakage}$) in the transistors as

$$P_{static} = V_{dd} \times I_{leakage} \tag{2.2}$$

The leakages mainly occur at gates and junctions, with additional subthreshold leakage through OFF transistors. There is a small amount of static dissipation due to contention current.

When a computation is run on a processor, dynamic power is consumed due to the transistors constantly switching states. How often the transistors switch state depends on the frequency (f) at which the processor operates. While switching states, the transistors with capacitance, C are charged from a 0 to 1 or vice versa using the supply voltage, V. The dynamic power dissipation is given by

$$P_{dynamic} = ACV^2 f \tag{2.3}$$

A is known as the activity factor, denoting how often the transistors are active. In addition to above, there is a small contribution of short-circuit power in the circuit towards dynamic consumption.

2.2 Processor power control

In this section, we look at the different power control mechanisms available.

2.2.1 Dynamic Voltage and Frequency Scaling

The processor power consumption as explained in Section 2.1 depends on the operating frequency (f) and the supply voltage (V) of the processor. Intuitively, one can control the processor power by controlling either the V, f or both. For reliable operation at a given frequency f however, there is a minimum voltage V that is required (varies by chip); making V not readily modifiable during operation without changing f. Therefore, the power consumption is controlling f and thereby, scaling V. This technique is commonly referred to as Dynamic Voltage and Frequency Scaling (DVFS).

For controlling power using DVFS on a Linux based system, the *acpi-cpufreq* or other applicable kernel modules requiring root privileges need to be loaded with only a limited set of frequency/voltage pairs supported. To change or read the supported frequencies thereafter from the */sys/devices/system/cpu/cpu*/cpufreq/scaling_available_frequencies* file no root permissions are needed, though the cpufreq governor needs to be set to *userspace*. The other governors generally supported are conservative, ondemand, powersave, and performance Users change the value of frequency by writing the desired frequency value to the */sys/devices/system/cpu/cpu*/cpufreq/scaling_setspeed* file. In case hyper-threads are enabled, both the threads need to be set to the same frequency for the DVFS to take effect.

DVFS has facilitated cubic reduction in the dynamic power dissipation with decrease in f taking advantage of Dennard scaling (Dennard et al., 1974) in the past. As the V depends on the current f

$$V \propto f \tag{2.4}$$

Using this in Equation 2.3 gives

$$P_{dynamic} \propto f^3 \tag{2.5}$$

Because of hardware limitations to date, DVFS research has impacted all of the cores on a multi-core processor and potentially slowed the critical path. Thus, the research has focused on finding situations where the slowdown is greatly outweighed by the energy savings. Dennard scaling however, is running into limits as feature sizes and voltages decrease. As transistor sizes decrease, static losses become a significant fraction of the total power utilized and undermine the power savings, while lower voltages interfere with reliable operation (Esmaeilzadeh et al., 2011). This makes possible reduction in power using DVFS much lesser than theoretically possible. The chip-wide effect of DVFS also made effective fine-grain control of performance difficult. Therefore, there had been a need for core-specific implementation of DVFS.

With the introduction of per-core specific voltage regulators in Intel Haswell (Kurd et al., 2015; Hammarlund et al., 2014), DVFS has been made core-specific to facilitate new options for software energy control. Each physical core (or 2 logical cores if using Hyper-Threading) can be independently controlled allowing only non-critical threads to have their frequency reduced. Core-specific DVFS is also supported in IBM Power8 (Fluhr et al., 2014).

2.2.2 Dynamic Duty Cycle Modulation

Dynamic Duty Cycle Modulation (DDCM) has been supported in Intel processors since Pentium as Software Controlled Clock Modulation, and is core-specific. For each core some percentage of the clock signals can be gated with rest of the hardware skipping that cycle. Figure 2.1 shows the working of DDCM.

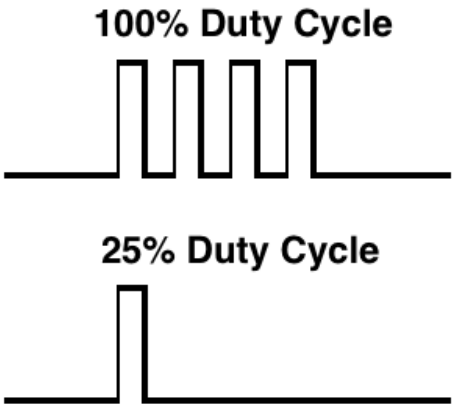


Figure 2.1: Dynamic Duty Cycle Modulation in operation

The top sub-figure shows the normal operation where 100% of the cycles generated by the clock is applied to the CPU. The bottom half shows one out of four (25%) of clock cycles generated applied to the CPU.

Power is dissipated in the processor core only during the active cycles. As the clock pulse is the same width even when few of the cycles are gated, the voltage required remains same to ensure reliable operation. The power reduction is thus a function of only the number of cycles gated (effective frequency). The stop-clock duty cycle is controlled with the **IA32_CLOCK_MODULATION** Model Specific Register (MSR) found in */dev/cpu/cpu*/msr* file using `rdmsr` and `wrmsr` operations, and requires root access on a Linux system. The need for root access at application level can be overcome using libraries like *msr-safe* (Shoga et al., 2014). The MSR kernel module needs to be loaded however, to make any changes to the MSR file.

Duty Cycle Level	Binary	Decimal	Hexadecimal	Effective Frequency
1	10001B	17	11H	6.25%
2	10010B	18	12H	12.50%
3	10011B	19	13H	18.75%
4	10100B	20	14H	25%
5	10101B	21	15H	31.25%
6	10110B	22	16H	37.50%
7	10011B	23	17H	43.75%
8	11000B	24	18H	50%
9	11001B	25	19H	56.25%
10	11010B	26	1AH	63.50%
11	11011B	27	1BH	69.75%
12	11100B	28	1CH	75%
13	11101B	29	1DH	81.25%
14	11110B	30	1EH	87.50%
15	11111B	31	1FH	93.75%
16	00000B	0	00H	100%

Table 2.1: Operational range of Dynamic Duty Cycle Modulation on Intel Sandy Bridge micro-architecture and beyond. Older versions mostly support only eight levels with a 12.5% transition step.

On the Sandy Bridge architecture, the effective frequency of the clock can be reduced to 1/16th of the actual frequency. It is controlled by a 4-bit counter which allows clock rates evenly spaced between 6.25% and 100% (Table 2.1). The hardware implementation could be only a few instructions making the latency very low. Low latency facilitates dynamic fine-grain use. Since the actual clock rate is not changed, other hardware options like DVFS and TurboBoost are still operational, and can be combined with duty cycle modulation.

The duty cycle is set for a single clock domain, normally a single core. The major benefit of DDCM is that it allows each clock domain to be set to different duty cycle, thus a multi-core processor can have

different frequencies active on each core. Consequently, a critical thread can run faster than a waiting thread. Duty cycle modulation does not modify the voltage or the clock for shared regions of the processor. The energy savings are less than DVFS at the same clock rate, since less of the system is impacted. But, it can reduce the clock rate down to 6.25% which is much lower than the minimum frequency obtainable with DVFS.

2.2.3 Power capping

Sandy Bridge and later architectures contain **MSR_PKG_POWER_LIMIT** in the RAPL Interface (Section 2.3.1), that sets a hardware enforced power limit for the chip. The hardware enforces an adjustable chip-wide average power over a given time period and is used to set the power cap (Intel, 2015a). The user specifies a time window and a maximum average power for that window and the processor guarantees that it will not exceed this average by distributing the power budget across its cores. Two separate time windows can be programmed by the user in to the MRS. A higher power could be made available for shorter intervals, with a lower power limit enforced over longer periods to aid application performance.

The introduction of power capping capability has led researchers (Rountree et al., 2012; Patki et al., 2013; Sarood et al., 2014) explore the idea of *over-provisioning* future HPC systems. In an over-provisioned or power-limited system, for problems that benefit from higher number of processors, power capping could be used to run nodes at a low power bound allowing more processors to be operated at lower frequencies. For problems that perform best given a smaller number of faster processors, a smaller number of nodes running at higher power cap or no cap can be used.

In this work, a constant power limit is enforced across the length of the application run for power-limiting experiments.

2.2.4 Energy reduction with power controls

In this section, the potential energy reduction obtainable with DVFS and DDCM is discussed.

Figure 2.2 shows variation in the total power consumption of a processor core as the static power component changes while using DVFS theoretically. When static power component is 100%, DVFS mostly has no effect. In contrast, when the static power component is 0%, the total power consumption is total dependent on the frequency set using DVFS and can be observed to change cubically with change in f as V scales commensurately. The static power component in modern systems constitutes an increasing fraction of

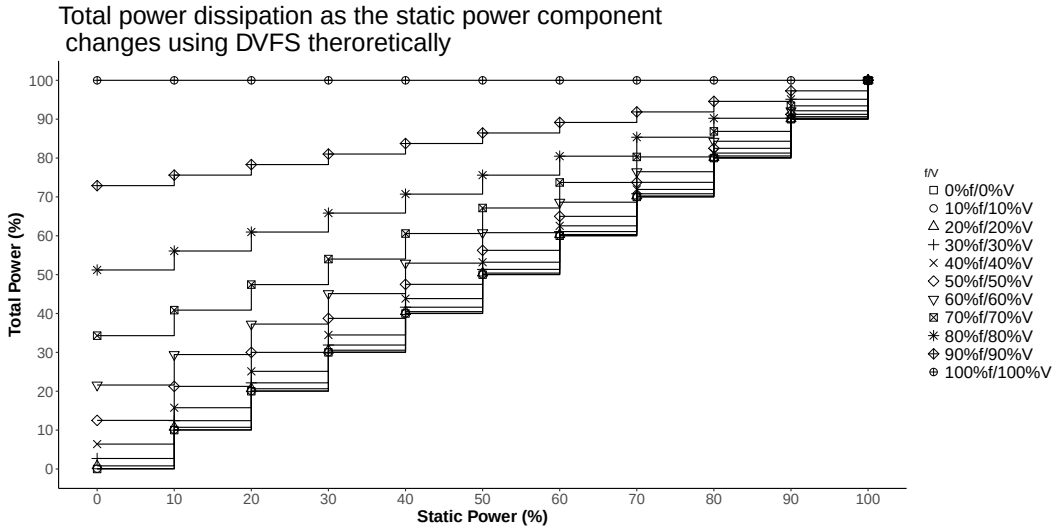


Figure 2.2: Variation in total power consumption of a processor core as the static power component changes while using DVFS theoretically

the total power and commonly contributes 10 to 30% or higher in extreme cases (Butts and Sohi, 2000; Goel and McKee, 2016). Given the contribution of static power to total power, Figure 2.2 gives a basic estimation of the total energy savings that can be achieved using DVFS. For example, if the static power component is 10% and the frequency is reduced by 10%, using Figure 2.2 the expected total energy consumption is 75.6% for a total energy reduction of 24.4% with no performance degradation for an application.

As discussed earlier in Section 2.2.1, the practical power reduction achievable with DVFS is at best quadratic. Figure 2.3 shows the variation in total power consumption of a processor core as the static power component changes while using DDCM and DVFS for a 2.4GHz processor core running at 1.2V by default. In comparison to the possible theoretical reduction of 75.6% seen above, reducing the frequency from 2.4GHz to 2.2GHz (8.3% reduction) gives an expected total energy consumption of 85.8% or reduction of 14.2% with no performance degradation and static power component as 10%. The reduction in energy consumption for DDCM is commensurate with the decrease in frequency, wherein a reduction of 12.5% in effective frequency reduces energy by 12.5%.

It can be seen that the power/energy reduction with DVFS at a particular frequency below default is much higher than DDCM. Thus, a core-specific implementation of DVFS is better for achieving greater power reduction within its operation range. DDCM on the other hand has larger operation range and can reach lower effective frequencies than DVFS at very low latency Experiments carried out on a Sandy Bridge machine show that the frequency transition latency for DVFS is about 30-40 microseconds and the latency for

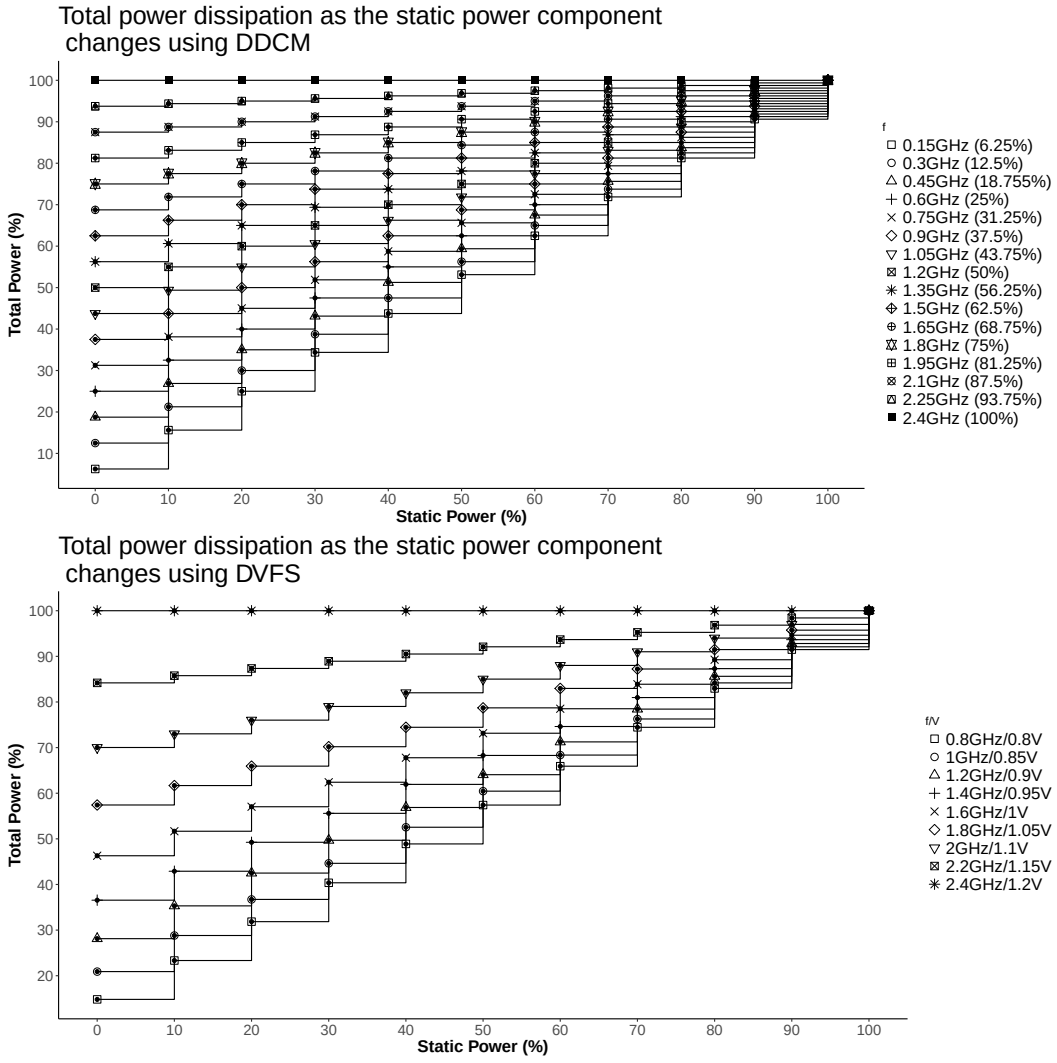


Figure 2.3: Variation in total power consumption of a processor core as the static power component changes while using DDCM and DVFS

switching between 16 levels of clock modulation is less than 2 microseconds (Wang, 2016). Running at low frequency can do more harm than good, hence due caution needs to be exercised to avoid severe degradation of performance that may lead to increase in overall energy consumption of an application. Thus, choosing the right power control for energy reduction is a function of how fast and low the frequency transitions need to happen.

2.3 Metrics Measurement

This section takes a look at the various counters available in the Intel processor micro-architecture and how they can be used to monitor the characteristics of an application during execution.

2.3.1 Running Average Power Limit

Starting with the Sandy Bridge implementation (Intel, 2015a), Intel extended the X86-64 architecture with the "Running Average Power Limit" (RAPL) interface. RAPL provides MSRs to measure and control microprocessor (PKG) energy/power utilization and also memory (DRAM). RAPL includes the **MSR_PKG_ENERGY_STATUS** counter which records the energy since boot in 15.3 nanojoule increments and can be accessed (read-only) using */dev/cpu/cpu*/msr* file through `rdmsr` operations. It is only 32-bit counter and rollover occurs frequently, but by counting the number of rollovers a very fine-grained measurement of chip power can be obtained. It provides the supervisor with energy measurements about 1000 times a second (1 millisecond). Temperature measurements are obtained from **IA32_THERM_STATUS_MSR**.

On the Sandy Bridge architecture, our understanding is that the energy measurements are modeled. Any resulting skew (e.g., difference between measured power and power-limit setting) should be consistent between runs of the same application. The relative difference between runs should be approximately correct, even if there is error in the absolute values. This weakness seems to have been overcome with the introduction of core-specific voltage regulators in the Haswell architecture.

2.3.2 RCRdaemon

Resource Centric Reflection (RCR) daemon (Porterfield et al., 2010), as the name suggests is a daemon that provides hardware performance counters to any program interested in how a system is performing. It focuses on counters for resources that are shared by more than one hardware core (e.g. energy consumption through RAPL, Last Level Cache performance or memory controller usage through Uncore Performance Monitoring). It can provide access to any counter available through PAPI (Mucci et al., 1999), PERF EVENTS (Per, 2008), or directly though hardware instructions. To provide access to some of the energy counters directly, traditionally RCRdaemon runs at root level for supervisor-privilege. This has been overcome by making RCRdaemon to use *msr-safe* (Shoga et al., 2014) to read counters that require root permissions. The RCRdaemon information is available to the programmer through a simple API that delineates a code region for measurement with a start and end call.

RCRdaemon writes the values into a shared memory region that can be shared with any other running application. To access the region the application only needs shm open a file in */dev/shm*. The shared memory region contains no pointers and access to it (after the first access) is accomplished with simple volatile loads

and stores. The counters tracked in the shared memory region is configured during RCRdaemon initialization and can be modified during daemon startup.

2.3.3 Effect of temperature on energy measurement

Temperature has a major effect on power used. Previous study (Porterfield et al., 2013a) has shown energy variation between nodes of an homogeneous cluster can be 10%. Also, power and time variations between consecutive executions of the same application can vary by 2+%. The runs on 'cold' processors can use up to 10% less energy compared to later runs with equivalent execution times. On several of the compute nodes used in the study, lower temperature correlated with reduced energy usage. Whether a cause or effect, hardware that is kept cooler uses less energy. To avoid energy variation with temperature, our experiments ignore results from the first several minutes until the system temperature is stable. Further, the values reported are generally averages taken over 10–12 runs.

2.4 Transparent monitoring and control of application

To facilitate transparent monitoring and control of applications the MPI profiling interface (PMPI) is used. `MPI_Init` and `MPI_Finalize` calls are intercepted to setup and clean the infrastructure. The `MPI_Init` starts the RCRdaemon to acquire program execution metrics like the power, energy, time, temperature among many others. While the `MPI_Finalize` call prints the profiling results and resets any power controls used during execution.

In typical use, calipers are placed around a region to be measured or controlled allowing instrumentations to be inserted. The region can be as large as the entire application, or as small as between two successive MPI calls. This strategy we will see is very useful in implementing the various energy policies discussed in the next chapters.

2.5 Applications

A number of Department of Energy (DOE) MPI mini-apps, and other real world applications were selected to simulate variety types of loads on HPC systems as required by this dissertation. This section provides a brief summary of the applications that were used in multiple works, other applications specific to a particular work or changes in the specific used for below applications are discussed in relevant chapters. It

can be noted that the problem sizes chosen are small, which is to facilitate multiple runs of the applications for arriving at average values and other statistics.

2.5.0.1 miniFE

miniFE is intended to be the best approximation to an unstructured implicit finite-element application that includes all important computational phases. The problem size on 32 nodes is 225x375x525 with 'load_imbalance' factor set at 100 to exploit maximum load imbalance that the application can present. The load imbalance however is much lower in actual operation.

2.5.0.2 miniGhost

miniGhost simulates highly structured stencil operations. It executes the halo exchange pattern important in structured and block-structured explicit applications. A problem size of 30x30x30 is spread across 1024 cores with 16, 8 and 8 cores along the three axes.

2.5.0.3 miniAMR

miniAMR does a stencil calculation on a unit cube computational domain and can emulate the interaction of different bodies in space. It uses Adaptive Mesh Refinement to better model the edges of the moving bodies. A test case of a sphere moving diagonally along 1024 cores with 16, 8, 8 cores along x, y and z directions is used. It runs for 10 time steps.

2.5.0.4 CloverLeaf

CloverLeaf investigates the behavior and responses of materials when applied with varying levels of stress using a two-dimensional Eulerian formulation. The input used is the provided "clover_bm512_short.in" and corresponds to a rectangular geometry of dimension 5.0x2.0 consisting of 30720 and 15360 cells along x and y axes respectively.

2.5.0.5 HPCCG

HPCCG is another approximation to an unstructured implicit finite but generates a synthetic linear system. The focus is entirely on the sparse iterative solver. The chosen problem size is 90x120x150.

2.5.0.6 AMG

AMG is a parallel algebraic multigrid solver for linear systems arising from problems on unstructured grids. The problem consists of a 27-point stencil on a cube with size 60x60x60. The processor topology is 16, 8, 8 cores along x, y and z direction respectively, and uses PCG with diagonal scaling as its solver for 1024 cores experiments.

2.6 Related Work

Energy efficient computing research in software fundamentally involves characterization of power/energy consumption of applications, architectures and infrastructure with an intent to design novel methods that promote optimal utilization of computational components at all levels of hierarchy. The mismatch between the workload and the resources mainly is due to current trend of using general purpose components to build the supercomputer. An important reason for using general purpose components apart from economics is to cater to a broad range of computational patterns characteristic to HPC applications. Table 2.2 attempts to classify the broad problems with their solutions that have been of interest to previous researchers doing power-aware or energy-efficient computing in HPC or general area of computing. The next few sections provide a brief survey of previous work. Finally, key distinctions of our work are summarized.

2.6.1 Computation work-load imbalance

Computational workload imbalance arises from uneven distribution work between the cores while running an application. The uneven distribution could be inherent to the nature of computation (e.g. sparse matrix) or due to poor coding practice. A few works aimed at detecting workload imbalance in applications and reducing power are discussed below.

Freeh and Lowenthal 2005 present a framework that uses Dynamic Voltage and Frequency Scaling (DVFS) for running a single application with several frequency-voltage settings. A program is divided into phases and then a series of experiments is executed, with each phase assigned a prescribed frequency. A significant potential is shown for energy savings in HPC applications without an undue increase in execution time. This basic idea is pursued by the authors further in the next couple of investigations.

Kappiah et al. 2005 introduce a system called “Jitter” that reduces frequency on nodes that are assigned less computation and therefore have slack time. The attempt is to ensure that nodes arrive “just in time” so

Problem	**Solution/Approach**	**References**
Computational work-load imbalance	Slack reclamation	Freeh and Lowenthal 2005, Kappiah et al. 2005, Hsu and Feng 2005, Ge et al. 2005, Kimura et al. 2006, Ge et al. 2007, Rountree et al. 2009, Tiwari et al. 2012
Waiting for a resource, mostly memory	Mitigate contention/Eliminate Idle waiting	Huang and Feng 2009, Eyerman and Eeckhout 2011, Livingston et al. 2014, Wang et al. 2015
Optimal resource utilization	Concurrency throttling	Curtis-Maury et al. 2006, 2008, Li et al. 2010, Porterfield et al. 2013b
	Operating under a power limit	Rountree et al. 2012, Patki et al. 2013, Sarood et al. 2013
	Switching off components	Youssef et al. 2006, Leverich et al. 2009
Communication	Eliminate idle waiting	Kandalla et al. 2010, Vishnu et al. 2010, Sundriyal and Sosonkina 2011, Hoefler and Moor 2014, Venkatesh et al. 2015
Resource aware scheduling	Energy/power	Fan et al. 2007, Etinski et al. 2010a,b, 2012, Mämmelä et al. 2012, Sarood et al. 2014
Analysis, profiling and surveys	Power/energy, architecture	Feng et al. 2005, Kamil et al. 2008, David et al. 2010, Hackenberg et al. 2015

Table 2.2: Classification of prior works aimed at improving energy-efficency in computing systems

that the overall execution time does not increase. As the work focusses on HPC applications, they assume iterative problems allowing the use of past behavior to predict future behavior.

Hsu and Feng 2005 provide the β-adaptation algorithm. This power-aware algorithm is implemented in the run-time system, and the performance is evaluated on commodity HPC platforms, both uniprocessor (SPEC CFP95 & CPU2000) and multiprocessor (NAS MPI benchmarks). The maximum allowed performance slowdown δ (e.g., δ = 5%) is specified by the user, and the algorithm uses DVFS to schedule CPU frequencies and voltages in such a way that the actual performance slowdown does not exceed the threshold value. No application-specific information *a priori*, e.g., profiling information is required making it transparent to the end-user applications.

Ge et al. 2005 present a study analyzing the tradeoffs of various DVFS scheduling techniques that exploit CPU slack time in distributed systems. A framework for application-level power measurement and optimization is proposed that implements distributed DVFS scheduling on power-aware clusters. The techniques used are largely manual and they remark that more work is needed to fully automate their process.

Kimura et al. 2006 showcase a toolkit called PowerWatch for power monitoring tools and DVFS control. A new energy reduction algorithm targets parallel programs that can be represented by DAG to reclaim slack using DVFS. They report empirical results for their new algorithm on real power-scalable clusters, a contrast from the simulated results presented in competing studies.

Ge et al. 2007 present a run-time system called CPU MISER and an integrated performance model. The runtime is application-independent and uses DVFS for target computational workload imbalance. Several types of inefficient phases are exploited including memory accesses, I/O accesses and system idle. The evaluation is carried out on the NAS parallel benchmarks.

Rountree et al. 2009 propose Adagio, a runtime that uses DVFS to slow down computation that is not on an applications critical path thereby reclaiming slack. Each task (period of computation between two MPI calls) is associated with a behavior to predict the time taken by the next task.

Tiwari et al. 2012 develop an application-aware analysis and runtime framework for job scheduling called Green Queue. It utilizes application behavior from intra-node and inter-node observations to churn out energy efficient runtime configurations using DVFS. The observations are obtained by instrumenting the application for data movement and MPI communication trace collection.

Most of the above works use DVFS with socket-wide effect as the control to reduce power. The socket-wide nature of DVFS is seen to sometimes increase the execution times greatly, making the researchers

consider scenarios where the large reduction in power outweighs the increase in time to show improvement in energy. The empirical software policies targeting computational workload imbalance discussed in this dissertation are similar to the intra-node methods discussed in above works, but focuses on measurements and controls at a local and not socket level. This makes the solution to scale well even on large systems with little impact on performance. Also, several of the above studies need application instrumentation for trace collection that is not required in case of the policies discussed in this thesis. In addition to using DVFS and DDCM separately, a policy to use them together is also demonstrated that shows further improvement in the energy reduction.

2.6.2 Waiting for a resource, mostly memory

HPC applications need access to memory often, and sometimes even I/O. Given the slowness of memory when compared to CPU, the data can take very long to reach CPU and the power is wasted. These memory operations are seldom visible to operating system (OS), making it difficult for the hardware circuitry to stall (or switch off) cores and reduce power while doing memory. The research below try to identify memory bottlenecks and reduce power.

Huang and Feng 2009 propose a systems-software approach that leverages accurate workload characterization through hardware performance counters guiding DVFS to improve power efficiency without degrading performance. The power-aware daemon called 'ecod' presented uses CPU stall cycles due to off-chip activities and does not require application-specific information a priori. The evaluation is carried out for SPEC CPU2000 and NAS Parallel Benchmarks.

Eyerman and Eeckhout 2011 show that coarse-grained DVFS is unaffected by timescale and scaling speed. They further suggest that fine-grained DVFS may lead to substantial energy savings for memory-intensive workloads. They propose a DVFS implementation based on mechanistic performance modeling utilizing on-chip regulators to record off-chip memory accesses with small performance penalty.

Livingston et al. 2014 present Runtime Energy Saving Technology (REST) that utilizes DVFS at runtime without prior knowledge. REST includes a memory versus non-memory phase detection system on two different Xeon architectures. The work also studies energy consumption on modern architectures using NAS and SPEC benchmark suites.

Wang et al. 2015 show that socket-wide DVFS has a high power state switching (frequency transition) overhead, preventing its use when a more fine-grained technique is necessary. They take advantage of the low

transition overhead of CPU clock modulation (DDCM) and apply it to fine-grained OpenMP parallel loops. They use memory access density to determining the right clock modulation setting and thread configuration for loops based.

Even while addressing memory bottlenecks, mostly socket-wide DVFS has been used to develop coarse-grain solutions that may suffer from the pitfalls of DVFS socket-wide effect. Some approaches use static analysis and may not always be flexible to adapt to dynamic changes. The memory metric policies presented in Chapter 5 are core-specific in nature and make decisions at runtime using measurements local to the core. They do not adversely impact performance whether computer or memory-bound, but lower energy if memory-bound. The metric identified detects bandwidth saturation and increased latency in the memory system.

2.6.3 Optimal resource utilization

Research focusing on optimal resource utilization generally put a constraint on an available resource and optimize the application to perform best within this constraint. The two major methods utilized are concurrency throttling and operating under a power budget. In concurrency throttling, the number of threads available to an application is restricted and the power corresponding to the idle threads is reclaimed. While operating under a power budget, nodes in a power-limited HPC system are forced to operate below the Thermal Design Power (TDP) thereby freeing up power to run more number of nodes than what is normally possible running the nodes at peak power. The research here mostly involves allocation of the available power budget wisely to improve the performance of the running application. There are also few works that discuss actively switching off computational components to reduce power.

2.6.3.1 Concurrency throttling

Curtis-Maury et al. 2006 present a user-level library framework for online adaptation of multi-threaded codes for low-power using hardware event-driven profiling and changing the processors/threads configuration as the program executes. The framework is implemented for OpenMP programs on multi-SMT system with Intel Hyperthreaded processors.

Curtis-Maury et al. 2008 combine multivariate regression techniques with data collected from hardware event counters to present a dynamic, phase-aware performance prediction model (DPAPP) for locating optimal

operating points of concurrency. The overhead of searching the optimization space for power-performance efficiency is reduced by using a prediction-driven runtime optimization scheme.

Li et al. 2010 present a system that uses both DVFS and Dynamic Concurrency Dynamic Concurrency Throttling (DCT) to improve energy efficiency. The implicit penalty of concurrency throttling on last-level cache misses is overcome by aggregating OpenMP phases using the proposed DCT algorithm. A power-aware performance prediction model is used for power-efficient execution of realistic applications from the ASC Sequoia and NPB MZ benchmarks.

Porterfield et al. 2013b reveal substantial variations exist in energy usage depending on the algorithm, the compiler, the optimization level, the number of threads, and even the temperature of the chip. They show that performance increases and energy usage decreases as more threads are used. However, for programs with sub-linear speedup, minimal energy usage often occurs at a lower thread count than peak performance. They design and implement an adaptive run time system that automatically throttles concurrency using data measured on-line from hardware performance counters.

2.6.3.2 Operating under a power limit

Rountree et al. 2012 evaluate the power capping capability in Intel Sandy Bridge micro-architecture in the HPC environment detailing opportunities and potential pitfalls. Their experiments with single-processor instances of several of the NAS Parallel Benchmarks show that manufacturing variation in processor power efficiency translates into variation in performance under a power cap.

Patki et al. 2013 demonstrate a policy to improve performance in over-provisioned systems using intelligent hardware-enforced power bounds. For several standard benchmarks they show that the optimal configuration depends on its parallel efficiency and memory intensity for a particular power cap.

Sarood et al. 2013 use power capping to improve execution time of an application by adding more nodes. The strong scaling of an application is profiled while using different power caps for both CPU and memory subsystems. The application profile is then used in an interpolation scheme to optimize the number of nodes and the distribution of power between CPU and memory subsystems to minimize execution time under a strict power budget.

Marathe et al. 2015 introduce an intelligent run-time system called Conductor to maximize performance under a power budget. Optimal thread concurrency level along with the required DVFS frequency is obtained

by exploring the configuration space such that it maximizes performance. Further, additional power is made available to predicted critical paths using adaptive power balancing.

Gholkar et al. 2016 overcome sub-optimal performance seen in power capped jobs arising from performance variation of identical processors using a two-level hierarchical variation-aware approach. At system level, the power available is partitioned across all the jobs without violating the power budget (PPartition). Thereafter, the optimal processor count and package power level is determined for a particular job taking into account processor performance variation to maximize performance under power budget (PTune).

2.6.3.3 Switching off components

Youssef et al. 2006 present a dynamic approach that has a better accuracy predicting the length of the sleep period with minimal power overhead aimed to reduce leakage power in processors. They maximize leakage savings by applying the sleep signal when it is most likely for the functional unit to stay idle long enough for the power savings to exceed the power overhead introduced by the application of the sleep signal. The proposed dynamic approach uses an instruction level analysis of the utilization of the functional units.

Leverich et al. 2009 propose per-core power gating (PCPG) as an additional power management knob apart from DVFS for datacenter workloads that exhibit load variability on a different time-scale than programs from conventional benchmark suites (e.g. SPEC CPU, MiBench). As PCPG has the ability to cut the voltage supply to selected cores, the leakage power for the gated cores is reduced to almost zero.

It has to be noted that datcenters also shut down entire computing racks during periods of low activity to save energy. This is in contrast to a supercomputer that is based on the principles of high availability where entire computing nodes/racks are switched off mostly during periods of maintenance and other unavoidable circumstances.

2.6.3.4 Sub-section summary

The idea of *overprovisioning* (operating under a power limit) is being explored aggressively, and future supercomputers may be power limited. The DDCM policy in Chapter 3 is shown to improve performance under a power limit. Since energy and power are being reduced by the policy, the hardware does not have to squeeze the chip quite as hard to stay below the limit. This allows the whole chip to run slightly faster. In effect, the policy is allowing power to be moved from the cores that do not need it to cores that do. The tent-pole thread runs faster. For a system that is not power-limited, when the frequency on the non-critical

cores is very low, the critical core can run faster with the additional available thermal headroom at turbo frequencies. The memory policies too improve performance slightly in certain cases mostly due to mitigation of resource contention.

2.6.4 Communication

The power used by the network in supercomputers is a considerable part of the total dissipation. Several works therefore analyze the power dissipation during communication operations and attempt to reduce the power consumption. As the CPU is generally idle during communication phase of an application there have been attempts to reduce CPU power using socket-wide DVFS or halt instructions.

Kandalla et al. 2010 use DVFS and CPU throttling on Intel Nehalem micro-architecture to reduce power during the communication phases of MPI applications. A theoretical model analyzes the power consumption characteristics of communication operations in generic work-loads and several optimized collective algorithms are presented. The evaluation is done using NAS benchmark and CPMD.

Vishnu et al. 2010 present Power Aware one-sided Communication Library (PASCoL) using Aggregate Remote Memory Copy Interface (ARMCI), the communication runtime system of Global Arrays, a Partitioned Global Address Space (PGAS) model. The impact of DVFS and a combination of interrupt (blocking)/polling based mechanisms is studied using communication primitives. The evaluation is done with synthetic benchmarks using an InfiniBand cluster.

Sundriyal and Sosonkina 2011 study all-to-all operation in MPI on per-call basis. They incorporate energy saving strategies within the existing all-to-all algorithms as the MPI_Alltoall collective in MVAPICH2 without modifying the standard algorithms used to perform this operation. They test this implementation on the OSU MPI benchmark as well as NAS and CPMD application benchmarks.

Hoefler and Moor 2014 analyze the tradeoffs between energy, memory, and runtime of different algorithms that implement collective operation in shared and distributed memory parallel applications. They show no known algorithm is optimal in addressing all of the three metrics and different algorithms can be the most optimal for a particular chosen metric.

Venkatesh et al. 2015 present Energy Aware MPI (EAM) – an application-oblivious energy-efficient MPI runtime using MVAPICH2. EAM uses a combination of communication models for common MPI primitives along with online monitoring of slack to maximize energy efficiency within performance degradation limits. They evaluate their runtime on ten applications using up to 4,096 processes on an InfiniBand cluster.

The works discussed above use socket-wide DVFS or halt instructions to reduced power during communication. The policies in the current research do not explicitly account for communication, but the communication time is considered to be a part of the collective waiting time. A few of the works discussed above make use of synthetic benchmarks like NAS that are not always representative of the actual HPC workloads. Some works use static models that may not always be accurate in dynamic changing heterogeneous environments. The policies discussed in this work use dynamic methods and are evaluated using HPC mini and full applications on HPC systems with and without power limits.

2.6.5 Resource aware scheduling

The works addressing resource (power) aware scheduling use socket-wide DVFS or power cap to control power assigned at a job level. These are mostly coarse-grained solutions applied to several known job schedulers.

Fan et al. 2007 present aggregate power usage characteristics of up to 15 thousand servers for different classes of applications over a period of approximately six months. The opportunities for maximizing the use of the deployed power capacity of datacenters and assess the risks of over-subscribing are evaluated. The evaluation shows 7 - 16% gap between achieved and theoretical aggregate peak power usage for well-tuned applications. With the observation that the energy/power savaing opportunities are greater at the cluster-level (thousands of servers) than at the rack-level (tens), they use their modeling framework to estimate the potential of power management schemes.

Etinski et al. 2010a showcase a power budget guided job scheduling policy that maximizes overall job performance for a given power budget. When DVFS is used under a power constraint, it is seen that more jobs can run simultaneously leading to shorter wait times improving performance. The DVFS frequency chosen for a job is dependent on instantaneous power consumption of the system and on the job's predicted performance. The evaluation is done on four workload traces from systems in production use with up to 4008 processors.

Etinski et al. 2010b introduce UPAS (Utilization driven Power-Aware parallel job Scheduler) on DVFS enabled clusters. A frequency assignment algorithm that uses DVFS is integrated into the EASY backfilling job scheduling policy. To avoid performance degradation, DVFS is applied only when system utilization is below a certain threshold to exploit periods of low system activity. They simulate five workload traces from systems in production use with up to 9216 processors for evaluation.

Etinski et al. 2012 demonstrate MaxJobPerf, a parallel job scheduling policy based on integer linear programming where optimization problem determines the DVFS frequency for a job. They also compare MaxJobPerf policy against other power budgeting policies for different power budgets, and provide a detailed analysis of the policy parameters including a discussion on how to manage job reservations to avoid job starvation.

Mämmelä et al. 2012 present an energy-aware scheduler for HPC data center that communicates with the resource management system. The scheduler tries to minimize the number of active servers of a system while still satisfying incoming application requests through changes made to the FIFO and Backfill First/Best Fit (E-FIFO and E-BFF/E-BBF) scheduling algorithms. The scheduler is evaluated with a simulation model and a real-world HPC testbed.

Sarood et al. 2014 propose an online resource manager called PARM that uses overprovisioning, power capping and job malleability along with the power-response characteristics of each job for scheduling and resource allocation decisions. A power aware strong scaling (PASS) performance model estimates application performance for a given number of nodes and CPU power cap. The resource manager is compared with SLURM scheduler on a 38 node cluster with two different job data sets.

The solutions discussed above mostly address power allocation up to node level. The power management at individual threads is not generally covered. All the policies presented in our work are intra-node and address fine grain power management across the cores.

2.6.6 Analysis, profiling and surveys

This section mostly consists of works that profile applications or architecture specific surveys with an emphasis on power/energy.

Feng et al. 2005 present a framework for direct and automatic profiling of power consumption for non-interactive, parallel scientific applications. Power-performance efficiency of the NAS parallel benchmarks is studied on a 32-node Beowulf cluster to generate profiles by component, node and system scale. They observe that power profiles are often regular corresponding to application characteristics and for fixed problem size increasing the number of nodes always increases energy consumption but does not always improve performance.

Kamil et al. 2008 provide power measurements for various computational loads on large scale HPC systems. They observe that the power consumed while running the High Performance Linpack (HPL)

benchmark is very close to the power consumed by any subset of a typical compute-intensive scientific workload. Further, they show that the power consumed by smaller subsets of the system can be projected accurately to estimate the power consumption of the full system as the difference in power usage due to switch fabric is not large.

David et al. 2010 present a comprehensive definition and evaluation of memory power estimation and limiting algorithm that significantly improves sensing accuracy, power limit enforcement and system performance. They discuss a mechanism for measuring DIMM power using a set of observable activity variables that share a tight correlation to the power consumption. Furthermore, they describe and evaluate the Running Average Power Limiting (RAPL) scheme for memory sub-system that can simultaneously enforce multiple memory power limits.

Hackenberg et al. 2015 cover a broad range of details and low-level benchmarks to provide researchers with fundamental understanding about the Intel Haswell processor generation. They analyze the impact of newly introduced features in Haswell on energy efficiency optimization strategies that use DVFS and DCT.

2.6.7 Summary

Most power aware computing research discussed has centered around DVFS that had a socket-wide effect used either inter-node (Kappiah et al. 2005, Rountree et al. 2009) or intra-node methods (Ge et al. 2005, Freeh and Lowenthal 2005). Computational workloads have been analyzed to propose ways to save power (Feng et al. 2005, Kamil et al. 2008). Models to amortize the effect of uneven work distribution through slack reclamation have been proposed (Kimura et al. 2006, Kappiah et al. 2005, Kang and Ranka 2010). Green Queue (Tiwari et al. 2012) automates the process of finding phases and optimal frequencies using power models. Automatic tuning of applications based on software performance options and processor clock frequency has also been explored (Rahman et al. 2012). The empirical software policy we present is similar to the intra-node models, but focuses on individual core (not socket) performance. The use of per-core DVFS and its combination with DDCM in the current dissertation is a departure from state of art solutions on architectures older than Haswell.

Moving beyond DVFS, duty cycle modulation (Porterfield et al. 2013a), power capping (Rountree et al. 2012) along with similar mechanisms on IBM Power 6 and 7 (capping) and AMD Bulldozer (amd 2012) (capping and thermal design power limits) have been explored. These solution have focused on specifically a single power control to improve energy-efficiency. Our runtime provides the flexibility of using multiple

core-specific power controls to save energy. Use of DDCM and DVFS together is shown to enhance energy reduction.

Applications have been profiled to determine the best configuration of nodes and power caps for overprovisioned systems (Patki et al. 2013, Sarood et al. 2013). Resource allocation schedulers that use overprovisioning incorporating power-response characteristics of each job along with power cap are being explored (Sarood et al. 2014). The use of core-specific power control is shown to improve performance with energy savings in such overprovisioned systems, or significantly reduce power (decreasing energy) without a power cap and may also improve performance by allowing the critical core to run at turbo frequencies.

There has been considerable amount of research to design energy-efficient runtimes oblivious to the running application (Venkatesh et al. 2015). These are aimed at alleviating the problems caused by system factors (OS noise, congestion) for runtimes that assume temporal patterns, and also to handle dynamic workloads. The runtime presented does assume temporal patterns, but we show that an adaptive solution is effective on par with premptive methods in handling dynamic conditions well.

A number of efforts use hardware performance counters (Snowdon et al. 2009, Choi et al. 2004, Lively et al. 2012) to compute optimal off-line settings. Several projects estimate energy usage based on hardware counters with direct correlation to cache access (Ge et al. 2007), MIPS (Hsu and Feng 2005) and CPU stall cycles (Huang and Feng 2009). The runtime policy targeting workload imbalance does not make use of any hardware performance counters and only makes lightweight dynamic adjustments to each core's individual clock frequency. Further, a number of studies show simulated results or use simple benchmarks that many a times do not accurately model the behavior of actual HPC applications. Standard benchmarks (mini applications) or full HPC applications are used in our evaluation.

While addressing memory bottlenecks, mostly chip-wide DVFS has been used to develop coarse-grain solutions that may suffer from the pitfalls of DVFS chip-wide effect. System-software approach that leverages accurate workload characterization via a unique synthesis of hardware performance counters in order to determine when and how to use DVFS to improve power efficiency while strictly maintaining performance has been proposed (Huang and Feng 2009). A fine-grained microarchitecture-driven DVFS mechanism that scales down voltage and frequency upon individual off-chip memory accesses using on-chip regulators is proposed in (Eyerman and Eeckhout 2011). Runtime Energy Saving Technology (REST) (Livingston et al. 2014) utilizes DVFS to modify frequencies of cores at runtime without prior knowledge using memory versus non-memory phase detection system on Xeon architectures. Some approaches use static analysis and may not

always be flexible to adapt to dynamic changes. The memory policies in this dissertation use the *TORo_core* metric at runtime and do not adversely impact performance, but save energy for memory-bound applications. The adjustments are made using local measurement at the core level.

CHAPTER 3: Using Dynamic Duty Cycle Modulation to improve Energy Efficiency

3.1 Introduction

Most research to regulate energy and performance in HPC has revolved around Dynamic Voltage and Frequency Scaling (DVFS) (Kimura et al. 2006, Kappiah et al. 2005, Rountree et al. 2009). DVFS is not used in production HPC because it is difficult to find applications where slowing the entire multi-core processor does not result in noticeable slowdowns. Until recently[1], DVFS has only allowed frequency and voltage changes that impacted all of the cores of a multi-core processor. Slowing the critical path slows execution. Thus, DVFS-centric research has focused on finding situations where the slowdown is greatly outweighed by the energy savings. This chip-wide effect limits the effectiveness of DVFS to be used for fine grained control. The use of DVFS also restricts access to higher turbo frequencies. This can save energy but slows applications, a side-effect unacceptable for most HPC systems.

Intel, as earlier discussed in Section 2.2.2, also supports Dynamic Duty Cycle Modulation (DDCM) in which the effective frequency of each core can be adjusted nearly instantaneously by only gating a fraction of the clock cycles to that core. This improved control allows DDCM to save power more effectively for imbalanced applications. With power-limits in place, the hardware can provide more power to the critical thread, improving overall performance. Changing the clock frequency with DDCM as discussed requires less work (and time) than with DVFS, since the voltage regulators are unaffected. DVFS does save more power at the same frequency, but as the operating voltage approaches the minimum voltage this advantage is shrinking (Esmaeilzadeh et al. 2011). The use of DDCM for energy conservation in HPC has not been heavily studied.

The major work and ideas presented in this chapter are:

- Using Dynamic Duty Cycle Modulation as an alternative to save energy and improve performance in power-capped environments.

[1]Intel Haswell architecture has introduced core specific voltage regulators that allow per core frequency control.

- An empirical runtime policy that uses DDCM to save energy in imbalanced MPI applications.
- Validation of the policy using a synthetic benchmark suite (RENCI Imbalance) and two mini-apps, miniAMR and graph500.
- The impact of node heterogeneity on DDCM use is investigated by running the policy on a machine with a different configuration from the testbed but same underlying architecture.

3.2 Dynamic Duty Cycle Modulation in HPC

DDCM has been supported in Intel processors since the Pentium era. In this section only the salient features of DDCM facilitating its use in HPC are highlighted, a more detail description can be found in Section 2.2.2. The major benefit of DDCM is that it allows each clock domain to be set to a different duty cycle, thus a multi-core processor can have different effective frequencies active on each core. Consequently, a critical thread can run faster than a waiting thread. Duty cycle modulation does not modify the voltage or the clock for shared regions of the processor. The energy savings are less than DVFS at the same clock rate. But, it can reduce the clock rate down to 6.25%, which is much lower than the minimum frequency obtainable with DVFS. Since the actual clock rate is not changed, other hardware options like DVFS and TurboBoost are still operational, and can be combined with duty cycle modulation.

3.3 RENCI Workload Imbalance micro-benchmarks

MPI applications that run on multi-core CPUs eventually hit a collective where all of the threads join and thus have multiple phases. The work distribution among the cores across phases can be fixed or vary. To explore the potential of DDCM-based energy saving techniques, a benchmark explicitly designed to be near best cases is used. In "Repeating Unequal" benchmark, each MPI process is a simple summation loop followed by a `MPI_barrier` as shown in the code snippet below.

```
chunk = rank % coresPerNode;
for(i = 0; i < REPEAT; i++)
{
  result = 0;
  for(j = 0; j < chunk * SCALE; j++)
  {
    result += j;
  }
```

```
  MPI_Barrier(MPI_COMM_WORLD);
}
```

Work is the same on every node, and within each node the same imbalances exist. The largest rank on each node does the most work and determines the execution time. Any slowdown of that thread increases execution time. “Repeating Unequal” was executed twice, once standalone and then with manually added optimal DDCM settings[2]. DDCM saved 22.2% in power and increased execution time by 0.14% for a net energy improvement of 22.1%. The potential savings are intriguing.

The next benchmark, “Equally Shifting Load”, implements a dynamic workload loosely resembling an Adaptive mesh refinement (AMR) code. The work increases each iteration until the maximum allowed is reached, at which point it is reset. Where “Repeating Unequal” energy can be minimized with a static clock frequency, “Equally Shifting Load” requires a dynamic solution, since each thread’s preferred clock frequency changes each iteration. AMR applications’ work will change over time although maybe not every iteration.

```
for(i = 0; i < REPEAT; i++)
{
  chunk = ((rank % coresPerNode) + i) % coresPerNode;
  for(j = 0; j < chunk * SCALE; j++)
  {
    result += j;
  }
  MPI_Barrier(MPI_COMM_WORLD);
}
```

In most AMR applications work can both grow and shrink over time. “Randomly shifting load” attempts to model this by allowing random changes in work either up or down each iteration. Where the previous micro-benchmarks work patterns are statically computable. The uncertainty in “Randomly shifting load” better expresses the performance variations that any dynamic policy must take into account.

```
srand(rank % coresPerNode);
for(i = 0; i < REPEAT; i++)
{
  randomValue = rand()% 3 - 1;
  chunk = (chunk + randomValue) % coresPerNode;
  if(chunk < 0)
  {
```

[2]The DDCM is manually computed, to have each thread arrive at the barrier with as little waiting time as possible.

```
      chunk = 0;
    }
    for(j = 0; j < chunk * SCALE; j++)
    {
      result += j;
    }
    MPI_Barrier(MPI_COMM_WORLD);
  }
```

For our experiment the load changes in the interval $\{-1, 0, 1\}$. For generic imbalance loads, DDCM can save significant power and energy with almost no impact on performance. Going through by hand and computing the "best" dynamic effective clock frequency is not feasible in practice. A policy is needed that can examine local system state and predict the proper duty cycle level to use for the next application region. We next arrive at a policy for reducing power empirically by applying an intuitive approach based on retrospective information, when an MPI collective is executed. The idea is that if a processor reaches the collective earlier than others, then we should slow it down (or speed up if it arrives late), so that on the next collective processors will more likely reach the collective at the same time.

Section 3.6.1 discusses the results obtained for the synthetic benchmarks with the policy in detail.

3.4 The policy

Hardware and software are evolving to increase performance variation between threads. Algorithm, compiler optimizations, number of threads and data locality impact performance and energy usage (Porterfield et al. 2013b). Physical variations during fabrication result in energy/performance differences during execution between otherwise identical processors (Rountree et al. 2012, Dighe et al. 2011). Our goal is to detect thread workload imbalances and automate the process of picking the right duty cycle level for the next application region.

The policy analyzes the time spent at and between collectives to determine when and how much a thread can be slowed down without impacting performance. It assumes that the next segment will match the current segment and determines the new DDCM setting. The policy chooses the lowest (slowest) setting that would have reached the collective without causing any additional delay. A thread doing more work will run at a higher duty cycle than the one doing less work. Optimally, all threads reach the next collective at almost the same time (Figure 3.1).

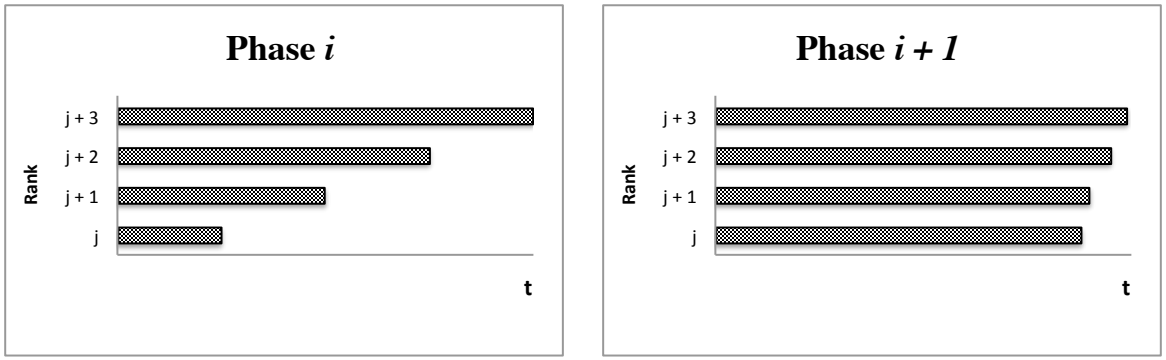


Figure 3.1: (left) Each MPI process in an MPI application is observed to complete its phase i in a fraction of the time in which all processes reach the barrier at the end of phase i. The runtime uses this fraction to adjust the duty cycle for each process in phase $i + 1$, according to eqns (1) and (2). (right) If the work per process is similar in phase $i + 1$, the completion times will be more nearly equal, energy will be saved and perhaps time will be decreased.

The policy uses two equations - one to decrease the duty cycle level of a core and the other to increase it. The equation to decrease the duty cycle level of a core is

$$L_{down} = \frac{T_{compute}}{T_{total}} * \frac{C_{max}}{C_{current}} \tag{3.1}$$

- $T_{compute}$ & T_{total} - time computing and total time since the last collective
- C_{max} & C_{min} - maximum (100%) and minimum (6.25%) duty cycle setting
- $C_{current}$ - duty cycle setting during last region
- L_{down} & L_{up} - number of levels by which the duty cycle should decrease and increase

It is a simple policy for computing the amount of slowdown required, but it does take into account the previous clock rate to prevent overly aggressive clock frequency reductions. The mapping of levels to duty cycle as used in the policy is shown in Table 3.1. The maximum duty cycle of 100% is mapped to level 16, while the lowest level (1) maps to a duty cycle value of 6.25%.

To handle real AMR-like applications, the policy needs to not only be able to reduce the clock rate, but must also be able to increase the clock rate to prevent the thread from being last to the collective. The idea is to provide breathing room to prevent threads running at less than maximum clock rate from slowing an application. One alternative is to always increase to max or to make it equal to $\frac{T_{compute}}{T_{total}}$, which would waste energy and could cause undue frequency oscillations. Another alternative is to always increase by one

Level	Duty Cycle (%)
1	6.25
2	12.5
3	18.75
4	25
5	31.25
6	37.5
7	43.75
8	50
9	56.25
10	62.5
11	68.75
12	75
13	81.25
14	87.5
15	93.75
16	100

Table 3.1: Mapping of levels to duty cycle as used in the DDCM policy

level, but that would respond slowly to changing conditions and increase execution time. Our approach is to compare the wait time with minimum wait time and increase more if the thread might have been the last to arrive.

$$L_{up} = \frac{T_{compute}}{T_{total}} * \frac{C_{min}}{C_{current}} \tag{3.2}$$

To better understand how these equations are used, consider the following scenario. When the application starts $C_{current}$ is set to C_{max} (i.e. all threads start at full clock frequency). At the first collective, if a thread is one of the last to arrive ($T_{compute} \simeq T_{total}$), then $L_{down} \simeq 1$ and the current clock frequency is left unchanged. At a later time, the thread may have significantly less work to do than the other threads and would wait at the collective for a significant amount of time. If $L_{down} = 0.8681$, it shows that the thread was running a little over 13% faster than required to get to the barrier before it is released. $C_{current}$ is set to the lowest rate that results in arriving at the next collective before the other threads. In this case $C_{current}$ is set to 87.5% (or 2 levels lower).

At some later collective, if $L_{down} > 1$. The thread is now doing more work and needs to have the clock frequency increased. Now applying the second equation, a new ratio is computed which indicates the amount

to increase the clock frequency. If $L_{up} = 0.1467$, the duty cycle is increased 2 levels (i.e if $C_{current}$ is 50%, it is set to 62.5%).

For very low duty cycle levels, higher performance degradation occurred than was predicted. To prevent this from slowing applications, the policy never sets the duty cycle below 18.75%. In practice, running a little too slow noticeably degraded performance. To avoid this effect, the policy was relaxed a bit. After computing the "correct" $C_{current}$, the speed is slightly increased $C_{current} = C_{current} + 1$. Some extra power is used, but the performance degradation is significantly reduced.

There are several options to avoid performance degradations from phase shifts. We use a damper to ignore the initialization phase at the start of several of the benchmarks. In practice, if the user or software could identify the start of a new phase, all threads could be reinitialized. If the phases are long and have a large number of collectives, the equations adjust and the performance impact may be small.

3.5 Infrastructure

The Intel Sandy Bridge architecture has hardware performance counters for measuring power/energy and temperature among others. It also provides several mechanisms to limit energy usage (power cap or modifying the clock rate). The power and temperature counters are read using RCRdaemon (refer Section 2.3).

3.5.1 Power Interface

PowerInterface allows the user system-wide introspection and control of processor energy and power usage. The PowerInterface is used directly by an application or embedded in libraries (e.g. MPI) to control energy with no application code changes. In typical use, calipers are placed around a region to be measured or controlled. The region can be as large as the entire application, or as small as between two barriers. The interface is pinned to the first core in the socket, avoiding any interruption to the other cores. By executing as root, no privilege level change is required and each call comprises only handful of staightline instructions, limiting the overhead. In our tests, the overheads are smaller than the run-to-run variation in performance and not detectable.

The PowerInterface allows applications to acquire program execution metrics like the power, energy, time and temperature through the RCRdaemon. `MPI_Init` and `MPI_Finalize` calls are intercepted to setup and clean the infrastructure. During program execution, the PowerInterface uses the empirical policy to

set the best effective clock rate between collective calls (`MPI_Barrier`, `MPI_Allreduce`). For current experimentation, a power cap is set at the beginning of the application for the duration of each run, and only the duty cycle is modified dynamically.

3.5.2 MPI

The MPI profiling interface (PMPI) is used to intercept `MPI_init`, `MPI_Finalize` and some collective calls. In the epilogue of the collective, the policy is called and DDCM is used to adjust the effective clock frequency. All of the work is done within the interface and should be effective for any MPI implementations (all testing is with MVAPICH2).

Within the MPI prologue and epilogue, the policy only uses values that can be locally computed. Some values are computed or set in prologue and used in the epilogue, but no data from other ranks is used. This eliminates the need for any communication, which could impact the application. The minor side-effects of using the policy with the application are the need to link against our MPI library (that uses PMPI) in addition to the standard MPI library, and the need to run as root to allow access to the DDCM control MSRs. The need for root access can be overcome by using libraries as discussed in Chapter 2.

3.5.3 Experimental setup

All tests used a portion of a local M420 Dell blade cluster. Each node has two Intel Xeon E5-2450 CPUs, each with eight cores, 96GB of memory running at 2.1GHz with hyperthreading enabled[3] and connected with Infiniband. The cluster runs Centos 6.5, is scheduled by Slurm 2.6.9 (with modified RAPL energy plugin) and runs a Linux 2.6.32 kernel.

Modern processors have enough internal heterogeneity that execution times often vary by several percent run to run (Porterfield et al. 2013a). The average is taken over 10 test runs. To avoid energy variation with temperature (Porterfield et al. 2013a), each test script ignored results from the first several minutes until the system temperature was stable. For each experiment, all runs at a particular power cap were run successively. The cap was varied from 40W to 100W in 5W increments (14 cases).

[3]The hyperthread enablement has no effect on our experiment as the benchmarks are run only on one thread per core and the duty cycle is changed by same amounts for both logical threads.

3.6 Results

The policy was validated using a set of synthetic MPI benchmarks created in-house. These are designed to be conditions where it should perform well. The policy was then tested with two standard benchmarks known to have uneven load, *miniAMR* and *Graph500*. *miniAMR* is part of the Mantevo Suite (Heroux and Barrett 2012) and uses a simple AMR algorithm. *Graph500* (Bader et al. 2010) is the MPI reference implementation of Breadth First Search (BFS).

3.6.1 Synthetic benchmarks

Three different unbalanced workload are simulated. *Repeating Unequal*, is static throughout execution. The next two, *Equally shifting load* and *Randomly shifting load* change the amount of computation performed by each thread every barrier. All simulated workloads exhibit very low memory traffic.

3.6.1.1 Repeating Unequal

The static unbalanced workload, mentioned in section 3.2, has work proportional to the rank number. On a single node with no power cap, Figure 3.2, energy consumption is reduced by 9.3% with a execution time increase of only 0.76%. When the benchmark is extended to run on a 16 node cluster, the energy savings is still 9.3% and the performance slowdown drops to 0.04%. Each execution of benchmark with the policy is on a average about $2^{\circ}C$ cooler on both one and 16 nodes. The combination of low overhead and providing each core with an effective clock rate tuned to its own needs, allows DDCM to save significant energy with almost no performance impact.

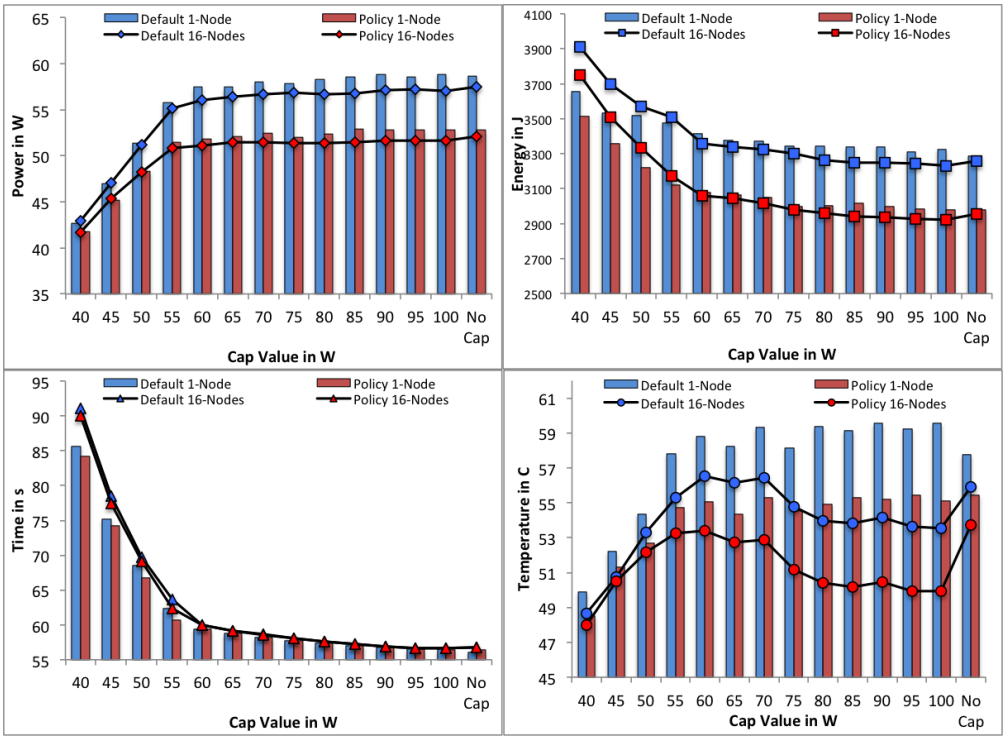


Figure 3.2: Power, Energy, Time and Temperature for *Repeating Unequal* pattern

More interesting results occur when socket-wide hardware power limits are applied. Since energy and power are being reduced by the policy, the hardware does not have to squeeze the chip quite as hard to stay below the limit. This allows the whole chip to run slightly faster. In effect, the policy is allowing power to be moved from the cores that do not need it to cores that do. The tent-pole thread runs faster. Performance improvements from saving energy will be important for *over-provisioned* systems of the future. The maximum energy savings are seen near where the cap starts effecting performance. At a cap of 55W, the policy runs about 2.6% faster with 10.2% savings in energy on one node. On 16 nodes, the speedup is 1.9% and energy saving is 9.5%.

3.6.1.2 Equally shifting load

Next, we see how the model reacts to a dynamic workload. Figure 3.3 shows the single and 16 nodes results. Execution time degrades by 0.59%, which is slightly less than before for the non-capped case. The power savings is 14.0% and energy savings is 13.5% are higher than for “Repeating Unequal”. Even on 16 nodes, the policy scales seamlessly with execution slowed down only by 0.79%. The power and energy

savings are 12.3% and 11.6% respectively. The reduced power again, results in lower socket temperatures ($3°C$ on one node and $4°C$ on 16).

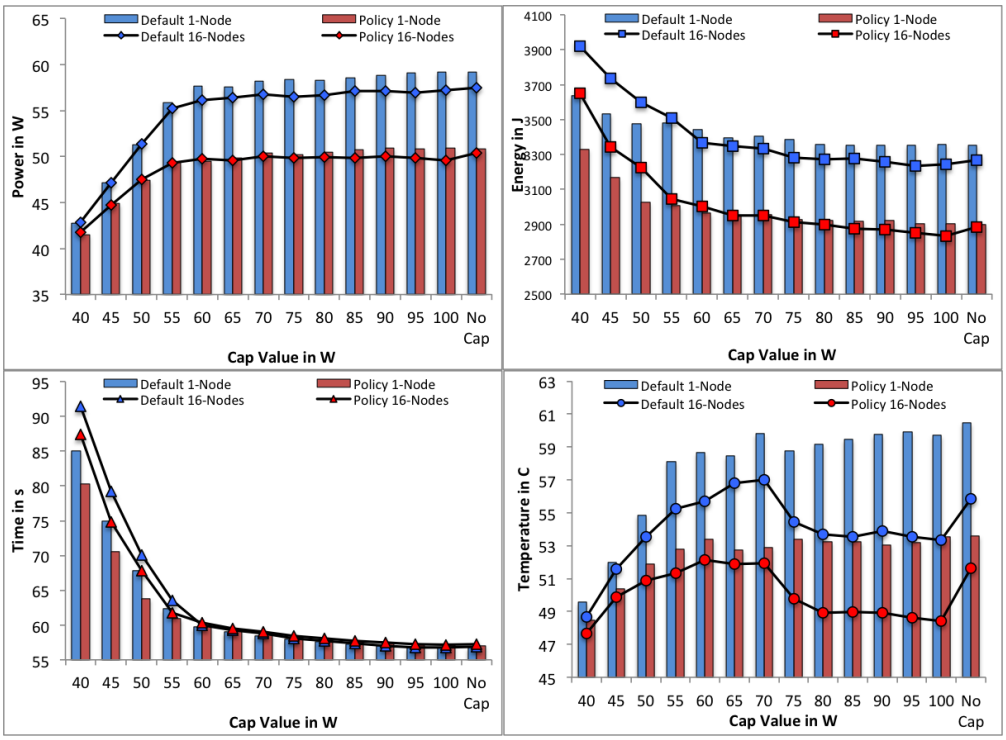


Figure 3.3: Power, Energy, Time and Temperature for *Equally Shifting Load* pattern

When running under a power cap, the adaptive policy translates the energy savings into faster performance. With a 50W cap on one node, the policy runs 6.0% faster on one node saving 13.0% in energy. On 16 nodes at a 45W cap, the application is sped up by 5.6% with a energy savings of 10.5%. Again the best energy saving occured when the cap was only slightly throttling the processor. For *Equally shifting load* the policy does a good job of handling multi-node barrier delay.

3.6.1.3 Randomly shifting load

Finally, we present the policy with a random dynamic workload. On single and 16 nodes without power cap, the execution time increased by 3.3% and 4.2% respectively (Figure 3.4). Even with this increase, the policy decreased the power consumption by 7.8% and saved 4.8% in energy on one node, and decreased power by 7.7% with 3.8% energy saving on 16 nodes. The increased randomness caused the policy some problems and execution time increased but the policy was still able to reduce the overall energy consumption.

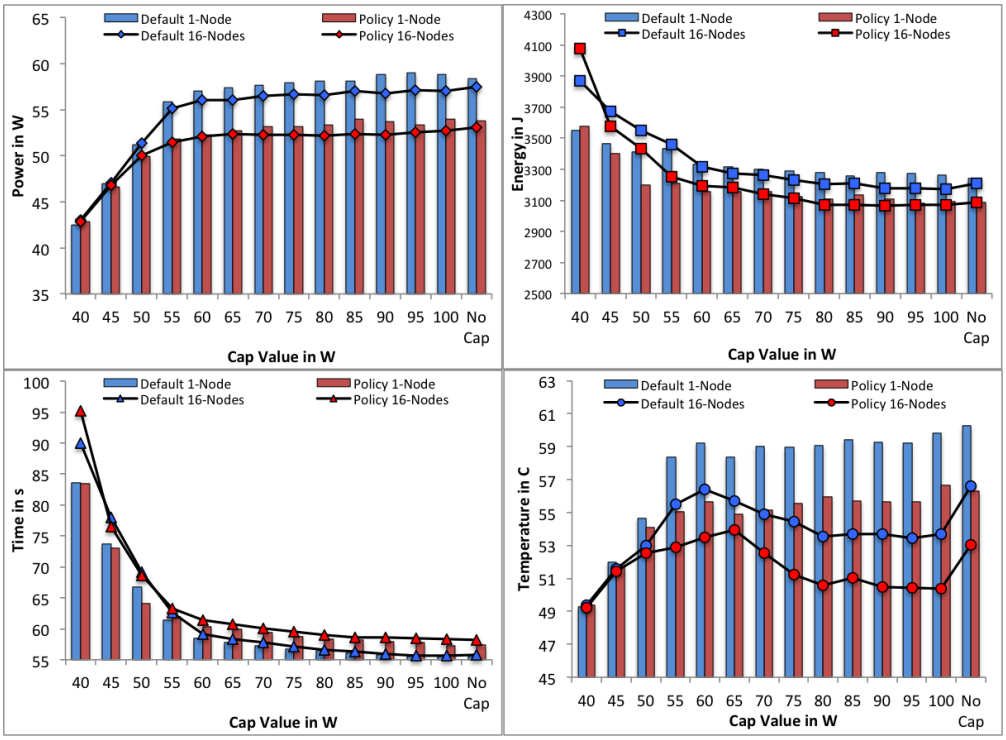


Figure 3.4: Power, Energy, Time and Temperature for *Randomly Shifting Load* pattern

Temperatures were reduced with DDCM between 3°C on one node and about 4°C on 16 nodes. With 50W power cap on single node and 45W power cap on 16 nodes, the application sped up by 4.0% and 2.0% respectively. The energy savings observed were 6.3% and 2.7%, mainly due to application speed up as the power reductions were 2.4% and 0.67%.

3.6.2 Standard benchmarks

The policy works for synthetic unbalanced workloads. The next test is to apply it to standard HPC benchmarks with unbalanced workloads. *miniAMR* is a scientific computation with a predictably varying workload and *Graph500* has a varying analytic workload.

3.6.2.1 miniAMR

miniAMR does a stencil calculation on a unit cube computational domain and can emulate the interaction of different bodies in space. It uses Adaptive Mesh Refinement to better policy the edges of the moving bodies. For the one node run, a sphere moves diagonally along 16 cores for 25 time steps with 4, 2, 2 cores along x, y and z direction respectively. On 16 nodes the problem size was increased 16x, the sphere moves

diagonally along 256 cores with 16, 4, 4 processors along x, y and z directions. The load balancing option is turned off in the current evaluation.

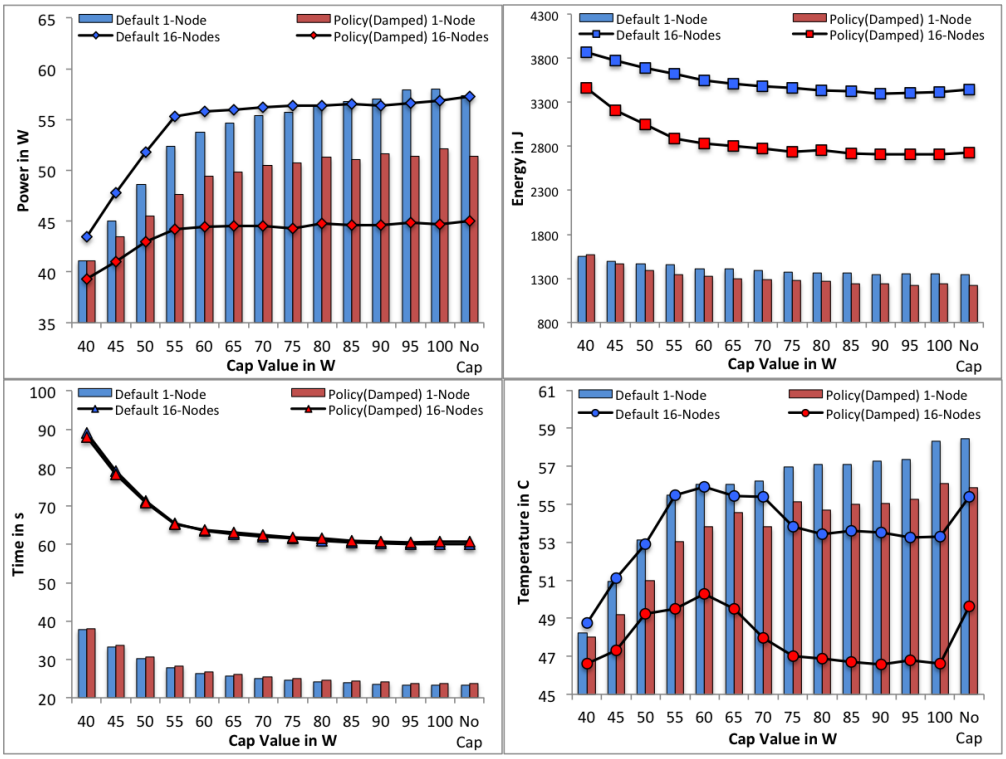


Figure 3.5: Power, Energy, Time and Temperature for *miniAMR*

On single node (Figure 3.5), the policy increased the application running time by 1.9% while decreasung the power by 10.4%, saving 8.7% in energy. On 16 nodes a 21.4% decrease in power and 20.8% reduction in energy are observed. The slowdown in application on 16 nodes is less than one percent (0.8%). Over the 60.2 second execution, the 20.8% total energy savings on 256 cores is 22.9kJ. *miniAMR* on 256 cores runs about $3°C$ cooler, while on 16 cores runs it is substantially cooler by $6°C$. The savings scale nicely as the mini-app is scaled to larger systems.

On a single node *miniAMR* the policy and the normal version performed approximately the same. On 16 nodes, the application sped up by 1.1%, and reduced power by 9.5% leading to 10.5% energy savings.

3.6.2.2 Graph500

The *Graph500* benchmark contains multiple analysis benchmarks that access a single data structure representing a weighted, undirected graph. The current evaluation uses the simple Breadth First Search (BFS) version that starts with a single source vertex, and then finds and labels its neighbor in phases. The single

node case uses 12 as its SCALE, while the 16 nodes case uses 11. A smaller SCALE on 16 nodes is used to avoid network overheads that increase execution time and may skew the energy results. No edgefactor is specified.

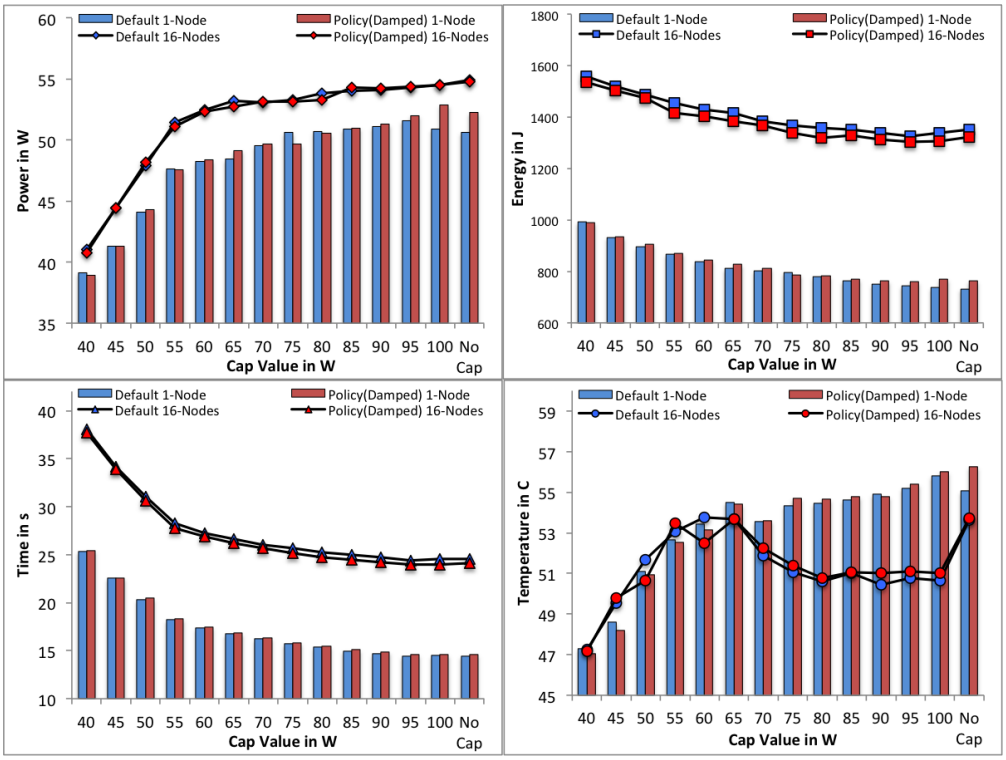


Figure 3.6: Power, Energy, Time and Temperature for *graph500*

Unlike previously discussed benchmarks, the behavior of *Graph500* with the policy is different. On single node (Figure 3.6), there was no power or energy reduction either with or without cap, and the application running time increased by about 0.7%. On 16 nodes, the energy savings are due to a combination of power reduction and application speedup even with no cap. The energy savings are predominantly due to speed up as the reduction in power is only 0.36%. The application energy is reduced by 2.2% with a speed up of 1.8%. No temperature change on the processor is observed as there is almost no power reduction. At 100W cap on 16 nodes the power reduction is less than 0.01%, but an energy reduction of 2.3% is observed due to a performance improvement of 2.3%.

3.6.3 Platform Variation

In some cases no observed speed up occurred or experienced more performance degradation than expected. To better understand these behaviors, the same experiments were repeated on a standalone system with a similar CPU architecture.

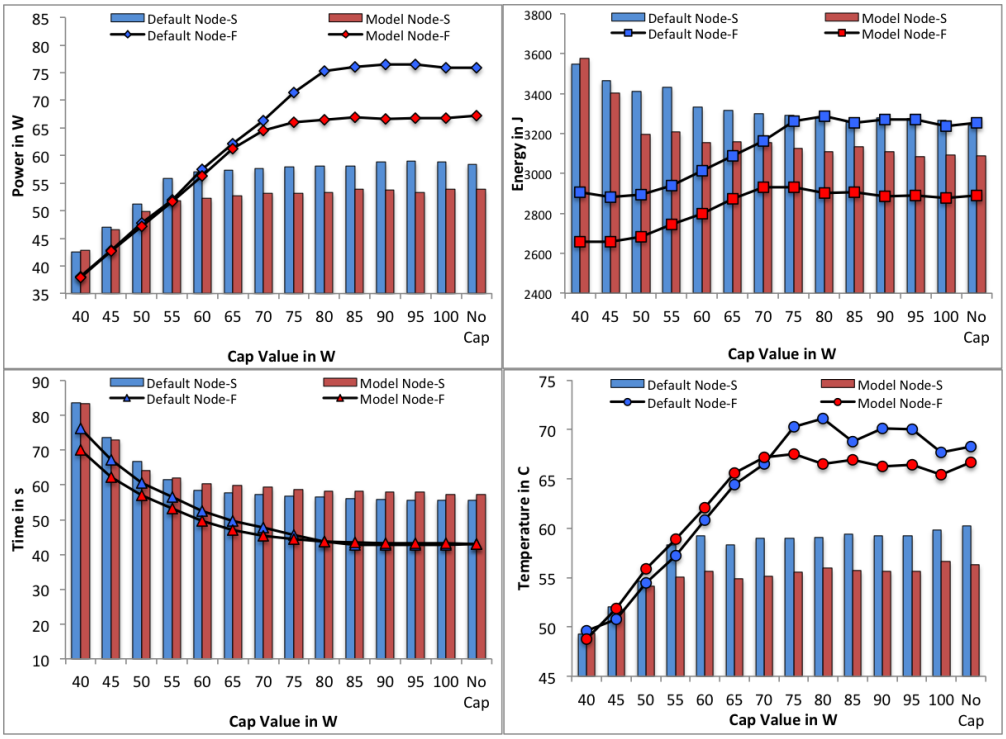


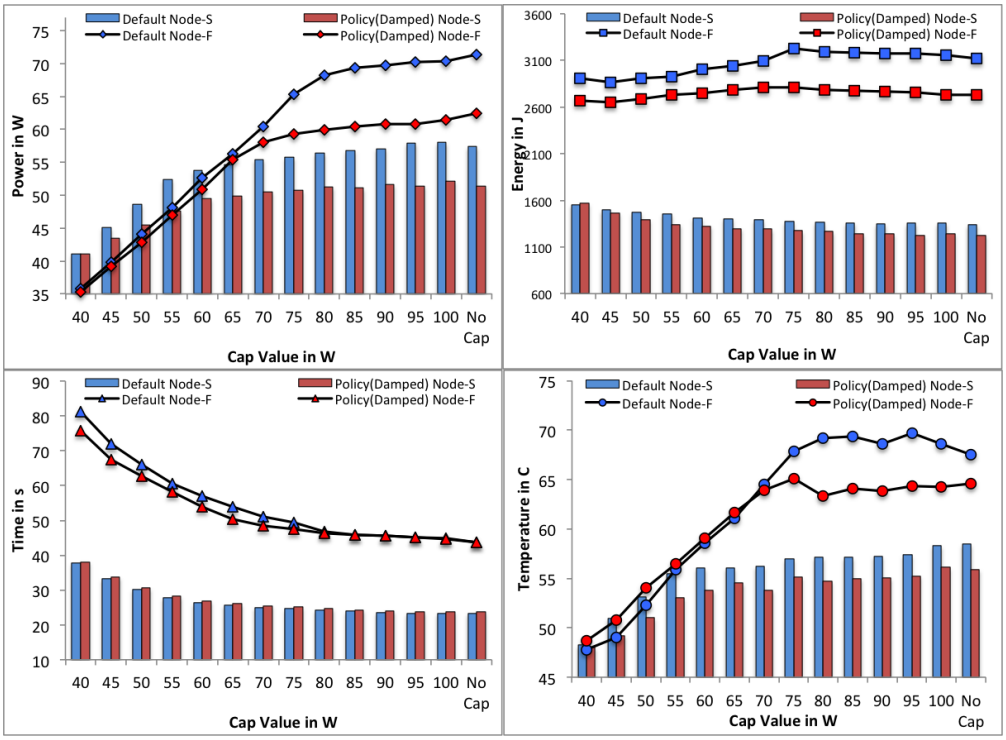
Figure 3.7: The graphs show variation in power, energy and time for two different nodes while using DDCM with RMR synthetic benchmark.

The new system (Node-F) has a higher default clock rate[4] and uses significantly more power than the original node (Node-S). The comparison of the two systems (Figure 3.7 & Figure 3.8) shows significant variation in results even for very similar architectures.

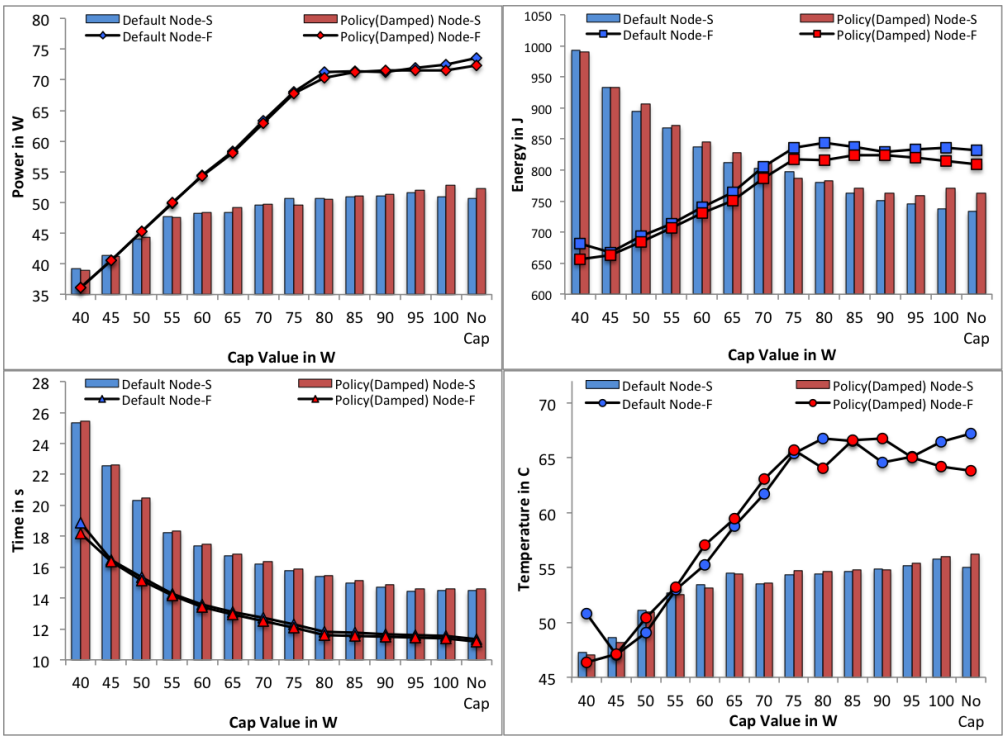
- The power consumed is substantially different. *miniAMR* consumed 57.4W on Node-S running at 58.5°C and 71.4W on Node-F 67.5°C. The faster (and maybe less well-cooled) system used 14W more and was 9°C hotter. System Design can have a major impact.

- DDCM seems to work better on the high power nodes. Benchmarks *Randomly shifting load*, *miniAMR* and *graph500* speed up under the power cap or see almost no performance degradation on Node-F.

[4]M620 Dell blade with two Intel Xeon E5-2680 at 2.70GHz containing eight cores each and 64GB of memory with hyperthreading disabled



(a) miniAMR



(b) graph500

Figure 3.8: The graphs show variation in power, energy and time for two different nodes while using DDCM on mini-applications. It is observed that in every case the results obtained using Node-F are superior to Node-S, suggesting the results obtained are highly machine dependent and agnostic to the actual mechanism using DDCM.

Further study to understand the significant differences is required, to determine if they are caused by minor difference in chip architecture or the significant differences in system design.

3.7 DDCM policy with full applications

Several full HPC applications are used regularly on RENCI system (Section 3.5.3) - ADCIRC for storm-surge modeling research, WRF within a larger climate-change research effort, and an optimization effort for LQCD. These applications were used to validate the DDCM policy (Porterfield et al. 2015). For the following tests, minimum speed that the policy could set the processor was limited. The Sandy Bridge architecture allows 16 setting of the duty cycle, the following tests used 3 different minimums; 100% no slowdown allowed; 75% or 12/16 maximum slowdown and 50% or 8/16 maximum slowdown.

Each application was sized to run for between 15-35 minutes This limits any initialization effects and allows the program to compute a significant result. All tests were run consecutively and are long enough to mitigate the effect of a cool start. Each application was run ten times for each power and software setting. The graphs show all results with the runs sorted by execution time to understand better the average difference between settings.

3.7.1 ADCIRC

ADCIRC+SWAN (Luettich et al. 1992, Westerink et al. 2008, Dietrich et al. 2010, Zijlema 2010), a storm surge, tidal and wind-wave model, uses a finite-element method to discretize shallow water equations, while simultaneously facilitating large domains with very high spatial resolution in coasts of interest, without unnecessary resolution in offshore areas. This high-resolution capability requires substantial compute resources (Blanton et al. 2012). Research and applications with ADCIRC include regional and local tidal phenomena (Westerink et al. 1994, Blanton et al. 2004), inlet and estuarine dynamics (Luettich et al. 2002, Hench and Luettich 2003); and storm surge and wave hindcasts (Atkinson et al. 2008, Dietrich et al. 2010, Lin et al. 2010). ADCIRC is approved by FEMA and was used for development of Digital Flood Insurance Rate Maps in TX, LA, MS, AL, DE, VA, NC, SC, GA, and FL (Blanton 2008, Ebersole et al. 2010, Niedoroda et al. 2010). It is also used as the core numerical model in real-time forecasting and prediction systems (Fleming et al. 2008, Blanton et al. 2012).

When running ADCIRC at 70W (effectively no limit) (Figure 3.9) the policy reduces energy consumption significantly. Running at 3/4 speed, a 6.0% reduction in total energy occurs, and as slow as 1/2 speed it increases to a 11.3% savings. As the power-limit is applied to the application the savings still occur. At 60W and 3/4 speed saves 5.6%, and at 50W, 3/4 at speed saves 6.6%. Examination of the application code reveals that most of the savings occur during IO phases, during which a majority of the work is on a single thread. This is because irregular finite elements of ADCIRC are not perfectly partitioned always. Furthermore, there is dynamic load imbalance (or change) when flooding starts to occur and previously dry elements get wet.

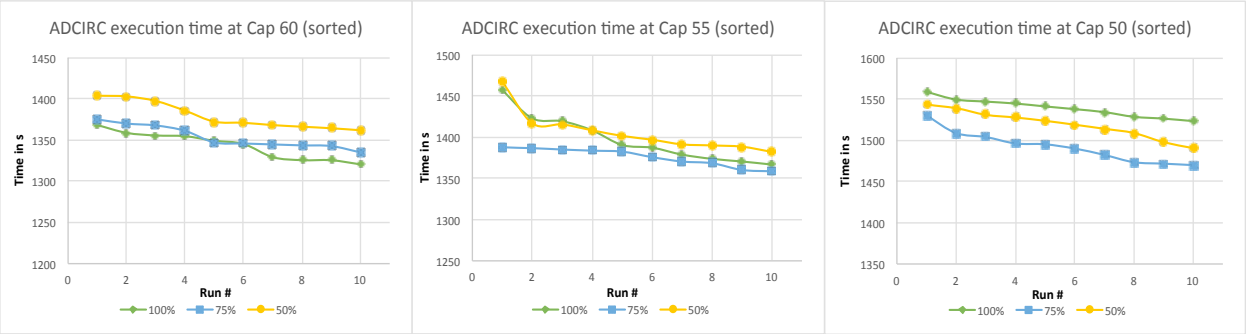


Figure 3.9: Execution times for ADCIRC with combinations of power cap and minimum threshold duty cycle

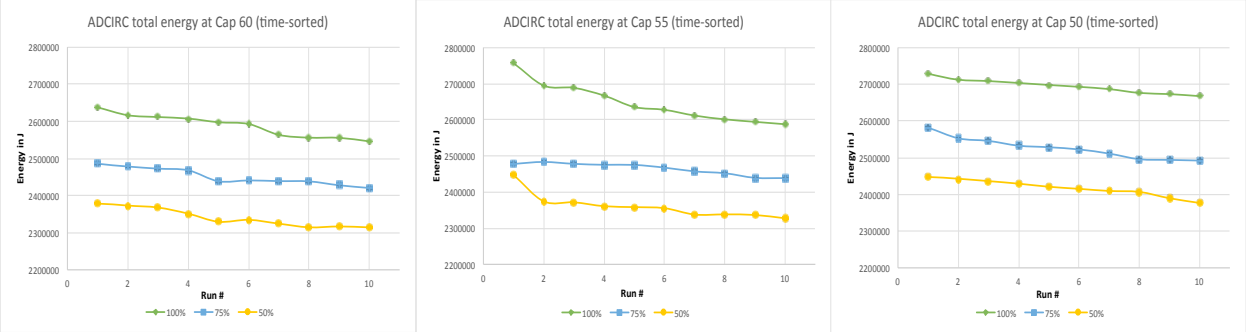


Figure 3.10: Energy consumed by ADCIRC with combinations of power cap and minimum threshold duty cycle

HPC systems are expensive, and any energy savings that increases execution time significantly is not cost-effective when you account for computer time. As Figure 3.10 shows at 60W running at 1/2 and 3/4 speed significant energy is saved, but execution slows slightly. As the power-limit is lowered and starts to impact execution time at 55W and 50W, the energy savings are 7.3% and 6.6%.

Since the power is limited to less than the desired amount, the energy savings can no longer come from a lower average power usage. Saving energy under a power cap means that the execution runs faster. The

average increase in execution speed is 1.5% at 55W and 3.1% at 50W. The hardware is intelligent enough to detect that power is being saved in some thread and effectively moves it to the critical threads, IMPROVING performance.

3.7.2 WRF

The Weather Research and Forecasting (WRF) Model (Michalakes et al. 2004) is a much used mesoscale numerical weather prediction system. The model serves a wide range of meteorological applications across scales from tens of meters to thousands of kilometers. Our test case is small and forecasts one day of weather for Puerto Rico given different boundary conditions obtained in a larger climate model simulation. It ran on the first six of the 16 available nodes.

WRF has low run-to-run variation and is well balanced between the threads. At no point does the policy detect code regions in which lowering the clock frequency by 6% or greater would not increase execution time. With the policy, the execution time, energy, and run-to-run variation are equal to the non-policy results presented earlier. However, WRF can become imbalanced when interesting atmospheric physics (condensation, ice, precipitation) occurs in part of the domain and the policy may be useful in such scenarios.

3.7.3 LQCD

The t_leapfrog program, distributed with Chroma (Edwards and Joó 2005), is a timing and functional test for a "leapfrog" integration scheme to compute trajectories. The test example is on a $32 \times 32 \times 32 \times 64$ lattice and uses periodic boundary conditions. This example is compute-bound near the limit of useful strong scaling with data fitting in cache as well as with load balance in both computation and communication.

t_leapfrog is perfectly balanced and has no distinct IO phase. The policy detects very few regions in which the clock rate could be reduced (8-15 during a 16 minute run). In Figure 3.11, the policy has minimal impact on execution time. At 60W and 70W, the various settings for the policy have no significant impact. At 50W, the policy slows the execution by 1% at 3/4 speed and 2% at 1/2 speed. LQCD has a low arithmetic intensity and is consequently memory bandwidth bound for large local problem sizes. With enough processors however, local problems fit in cache and the problem is greatly mitigated making the policy not that effective.

When energy is examined, Figure 3.12, at 70W there is no significant difference between the three policy settings. At 60W, the total energy used increases slightly (both graphs have the same bounds) but the

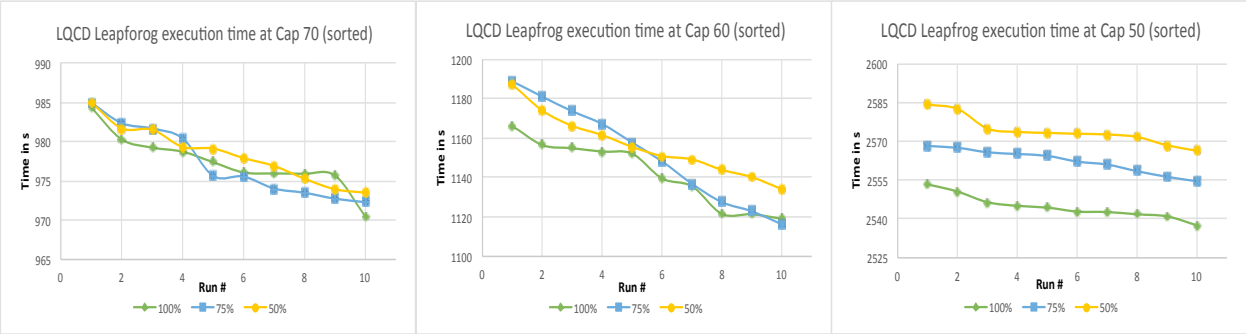


Figure 3.11: Execution times for `LQCD` with combinations of power cap and minimum threshold duty cycle

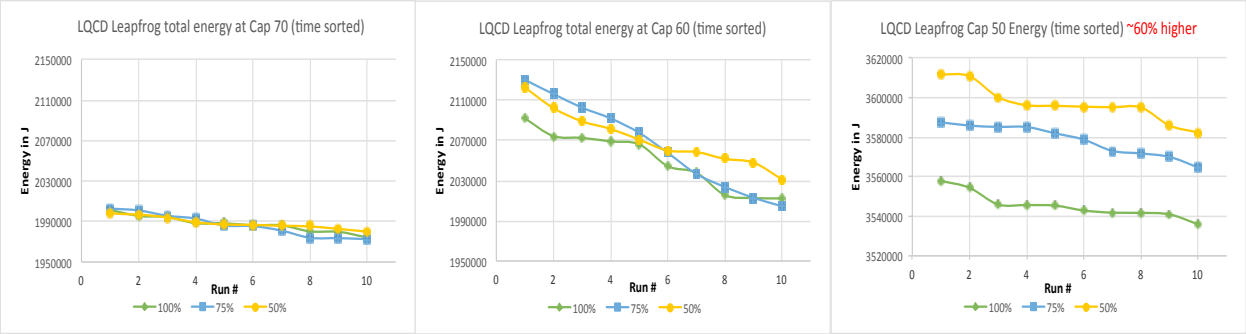


Figure 3.12: Energy consumed by `LQCD` with combinations of power cap and minimum threshold duty cycle

run-to-run energy variation increases significantly. Overall the increase in execution time at a cap of 50W, greatly increase energy usage. Because the policy slows execution speed, the policy slightly increases energy usage.

For a compute-bound application, such as `t_leapfrog`, the energy policy detects very few regions of code where the imbalance is large enough to change the clock rate. Applying the policy has little impact on the execution and energy consumption of compute-bound applications. The application of a power limit can significantly increase execution time and total energy usage.

3.8 Conclusion

Over the last decade much research into energy saving methods has been conducted using DVFS. The fundamental weaknesses of DVFS with respect to HPC can be overcome with DDCM as it has a per-core control with lower overheads and can be used to create fine-grained core-specific clock frequencies. HPC systems are likely to be *overprovisioned* as they approach Exascale on fixed power budgets. Each node will be power limited. Power limits will increase the heterogeneity of the systems as no two nodes will run at

quite the same clock rate. All applications will have some degree of imbalance between its threads. By identifying the fast threads in software and slowing their clock, the hardware can recognize the power savings and increase the global clock rate. A faster clock rate impacts program performance and system throughput.

The chapter described a policy that uses DDCM to achieve fine-grained power control in the MPI collectives. Synchronization delays is one of several applications features that could be exploited for energy savings with minimal impact on performance. In the future, examining rate limited application and determining if the performance could be improved (or energy saved) by shifting power to the busy resource from the lightly used resources would be interesting (slow threads when memory bound/slow cache/memory and faster threads when computationally bound). Saving energy in a power-limited systems is a performance optimization. HPC has historically ignored energy saving techniques. As they transition to a world where *overprovisioned* systems are common, runtime energy savings/optimization techniques will be important to an efficient HPC application. In the future, we would like to compare DDCM to the recently released core-specific DVFS on unbalanced applications to know, which saves the most energy – lower overheads or lower voltage?

CHAPTER 4: An Adaptive Core-specific Runtime for Energy Efficiency

4.1 Introduction

The runtime policy using DDCM in the last chapter showed potential for energy reduction in the presence of computational workload imbalances. With the introduction of per-core specific voltage regulators in Intel Haswell, new options for software energy control are now available. Each physical core (or 2 logical cores if using Hyper-Threading) can be independently controlled allowing only non-critical threads to have their frequency reduced. It is therefore fitting to explore the potential of the new control in aiding energy improvement. To make aforementioned policies generic enough to be relevant for a wide variety of HPC application and for their possible adoption in production systems a method to limit any performance degration is also necessary. These factors facilitate a need for a runtime framework consisting of the policies along with options to tailor the behavior of the policies as per the user.

In this chapter, we extend the DDCM policy discussed in Chapter 3 and present a *generic* policy that uses a core-specific power control like DDCM or per-core DVFS to throttle the frequencies of cores not on the critical path. The goal is to match a core's duty cycle to its workload to eliminate idle cycles. The *duty cycle* is given by,

$$Duty\ cycle = \frac{Time\ core\ (processor)\ in\ active\ state}{Total\ time} \times 100$$

The amount of time a core is active can be changed either by lowering its T-state (with DDCM) or by reducing frequency (DVFS). By dynamically adapting core frequencies to workload characteristics on that core, fewer idle clock ticks occur and less power is wasted. Many HPC applications comprise multiple phases of computation with each of the cores performing disparate amounts of work leading to workload imbalance. In the current policy, a core doing more work will have a higher duty cycle (and run at higher T-state/frequency) than the one doing less work. Ideally, all threads reach next phase boundary at almost same time (Figure 3.1).

Many HPC researchers are exploring *overprovisioning* of processor nodes (Patki et al. 2013, Sarood et al. 2013, 2014) to improve performance within the available power budgets. Many future exascale applications will have heterogeneous processor load, and with power-limits, most exascale systems will have heterogeneous performance. This can lead to significant run-to-run variations in the execution time and energy consumed. The adaptive runtime framework saves energy by dynamically setting core-specific frequencies. To the extent possible in the hardware, the power saved in non-critical nodes can be allocated to the cores on the critical path. This results in execution time reductions, particularly under a hardware power cap (Bhalachandra et al. 2015).

This chapter presents the following major work and ideas:

- A generic policy that effectively utilizes per-core specific power controls to improve energy efficiency. The DDCM policy from Chapter 3 aimed only at showing the efficacy of DDCM as an alternative to socket-wide DVFS. We now present work offering a context for comparing DDCM (with its simple per-core hardware implementation and fast switching capability) and DVFS (more complex and costly to implement per-core but with potential for greater savings), and for showing how and when they can be used together.

- Implementation of an adaptive runtime framework (library) that uses the duty cycle inspired policy to throttle the frequencies of cores not on the critical path of an MPI application. An important feature of this implementation is that it allows the flexibility to use multiple power policies to save energy - DDCM, per-core DVFS or both. Use of this library does not require any code changes to the underlying application.

- Validation of framework using six mini applications (*miniAMR, miniFE, CloverLeaf, HPCCG, AMG, miniGhost*), and full application, *ParaDis*. The evaluation shows an overall 20% improvement in energy efficiency with an average 1% increase in execution time on 32 nodes (1024 cores) using per-core DVFS.

- *Energy optimization is shown to improve performance in certain scenarios*. With a full application *ParaDis*, the runtime is seen to improve performance by lowering run-to-run variation and facilitating running at turbo frequencies. The performance improvement is achieved in addition to reducing power.

4.2 Adapting core frequency to workload characteristics

HPC applications have large varieties of CPU/memory usage patterns that are input driven and dependent on executing application phase. System noise from diverse factors like hardware, OS, network further complicates any static attempt to determine optimal core frequencies. These factors drove our choice to create an adaptive policy driven by runtime inputs.

Slack reclamation by trying to slow down the non-critical paths of computation is not new to *energy-efficient* HPC nor is the idea of an adaptive runtime. Most previous research has revolved around DVFS and its ability to obtain cubic savings in energy. The previous chip-wide requirement made it difficult to find applications where the savings did not result in significant execution time increases. Some of the previous investigations (see Section 2.6.1) used complex models requiring system-wide introspection that are better suited for off-line analysis. Other studies required application level source code changes making them tedious and difficult for production applications.

In the previous chapter, we saw how the DDCM policy examines local system state and predicts proper duty cycle level to use for next application region. It saved 13.5% processor energy on one node and 20.8% on 16 nodes for several benchmarks. On a production application, `ADCIRC`[1] energy savings of 10% were obtained with only a 1-3% increase in execution time.

In the current work, we offer a context for comparing DDCM (with its simple per-core hardware implementation and fast switching capability) and DVFS (more complex and costly to implement per-core but with potential for greater savings). This is done by showing how the previous policy (Porterfield et al. 2015) can be made *generic* to work with per-core DVFS in current work, and other core-specific power controls that the hardware might provide in the future. Further, a novel approach to combine per-core DVFS and DDCM is presented. The combination of multiple power controls in complementary ways is shown to achieve improved energy savings.

The new *generic* policy reduces power by applying retrospective information during an MPI collective to predict slack at the next MPI collective. The idea is that if a core reaches the collective earlier than others, then it should be slowed down (or sped up if it arrives late) so that on the next collective cores will more likely reach the collective at the same time (Figure 3.1). The policy depends on the amount of work performed

[1]The storm surge, tidal and wind wave model ADCIRC+SWAN is used to simulate and predict water inundation and wind wave impacts from coastal storms.

between MPI collectives to be relatively stable during execution. In practice, this has not been found to be overly restrictive.

Only local *timing* and state information are used at each core. No global communication or state is required. This allows the policy to scale to any application size. The policy is implemented within the MPI profiling interface (PMPI) and requires no application code changes. Calls to `MPI_init`, `MPI_Finalize` and most MPI collective calls are intercepted. Data is computed or set in the prologue and used during the epilogue to determine the next phase's clock frequency. No data from another rank is required, eliminating the need for any communication. The application does need to link against our MPI library in addition to the standard MPI library, and needs to access protected machine-specific registers (MSRs) only to control power using software-controlled clock modulation. The access to MSRs can be obtained either by using libraries like *msr-safe* (Shoga et al. 2014) or by running the application as root. For controlling power using DVFS, the *acpi-cpufreq* or other applicable kernel modules need to be loaded.

4.2.1 Working of policy with DDCM

The policy's goal as explained in Section 3.4 is to detect and reduce imbalances. When a core is running faster than needed, that core's effective clock frequency is reduced. The new frequency is chosen to be the lowest such that core will not be the last one to arrive at the next collective. If the last core to arrive at a collective is running at full speed, the application should experience no slowdown.

The policy automates the process of selecting the clock frequency for the next application region by comparing the computing and waiting times of each core. If a core reaches a synchronization point early, (e.g. has a significant fraction of waiting time), it is assumed that the core will also arrive at the next synchronization point early and is a candidate to have its clock frequency reduced. The clock frequency for the next phase is calibrated using the compute and waiting times for the previous region. A core doing more work will run at a higher effective frequency than the one doing less work.

Figure 4.1 shows working of the generic core-specific adaptive runtime policy. The policy uses two rules - one to decrease the clock frequency of a core and the other to increase it. It first attempts to decrease the clock frequency.

$$L_{down} = \frac{T_{compute}}{T_{total}} * \frac{C_{max}}{C_{current}} \tag{4.1}$$

- T_* - time

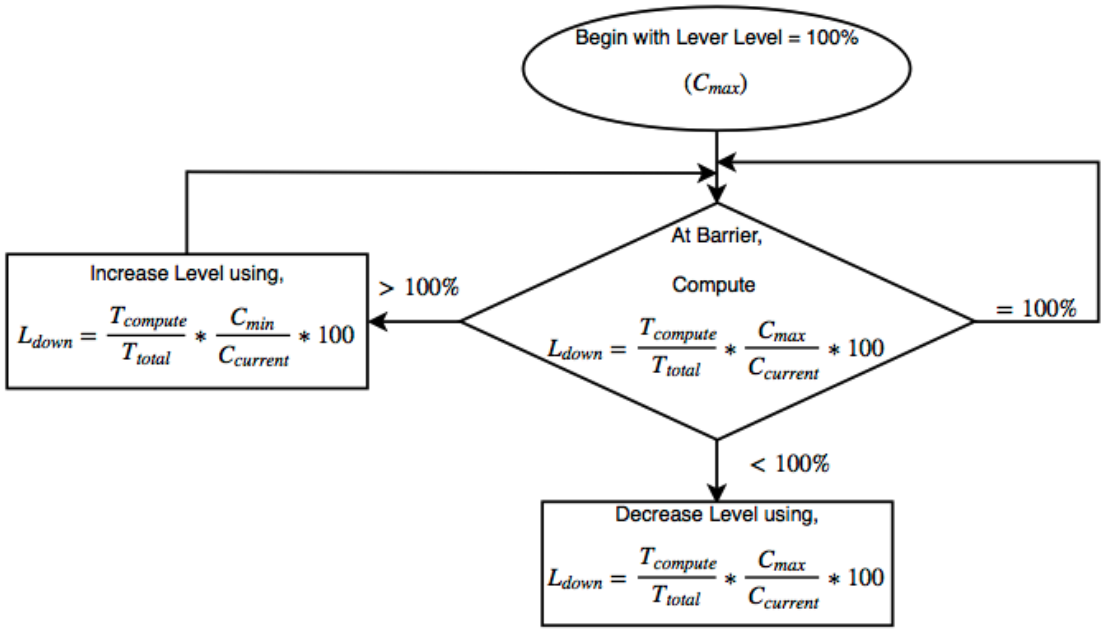


Figure 4.1: Working of the generic core-specific adaptive runtime policy

- C_* - clock frequency
- L_* - levels/steps to change clock frequency

The next clock frequency is a function of the ratio of computing time to total time between barriers and the previous clock frequency. In practice not every clock frequency is available, the one chosen is the lowest frequency such that it would have made the core to wait least at the last barrier.

When the previous rule determines that duty cycle does not need to be reduced, the policy then determines whether the duty cycle needs to be increased to prevent the current core from being the last to arrive at the next barrier, thus slowing the application. The policy aggressively increases frequency when it determines that this core may have been the last to arrive a the next barrier. Increasing a core from the minimum clock frequency to the maximum only takes a few policy invocations rather than one for each effective clock rate level possible.

The equation to increase the duty cycle level is given by

$$L_{up} = \frac{T_{compute}}{T_{total}} * \frac{C_{min}}{C_{current}} \tag{4.2}$$

The model estimates the next value for the duty cycle by comparing wait time with how close to the minimum duty cycle the last region was executed. Thus, the model again assumes some predictability between successive phases.

4.2.2 Making the policy generic (per-core DVFS)

The approach to making the adaptive runtime policy *generic* is straightforward. This is achieved by changing the maximum, minimum and intermediate values to the ones supported by core-specific power control that may be provided by the hardware in the future.

For per-core DVFS, C_{max} is the maximum non-turbo frequency on a machine, and C_{min} is the lowest frequency supported by DVFS. The transitions (L_*) occur at frequencies available in */sys/devices/system/cpu/cpu*/cpufreq/scaling_available_frequencies*[2].

4.2.3 Combined Policy

On an Intel Haswell machine, the cores can either use T-state (DDCM) or P-state (DVFS) transitions to lower frequency. DVFS can generally support frequencies only down to about half the standard non-turbo frequency of the processor. As discussed in Section 2.2.4, the power saved using DVFS is higher in comparison to DDCM as both voltage and frequency is reduced. When the clock frequency needs to be reduced beyond what DVFS allows, DDCM can be used to further reduce the clock frequency.

The combined policy can be modeled as a joint optimization problem. However, preliminary profiling of many HPC applications showed that the phase times are long enough (hundreds of milliseconds or more) so that DVFS can be used efficiently to reduce frequency even with larger transition overhead (Section 2.2.4). The lack of frequent changes required motivated a two level approach where DDCM is used to effectively to increase the operational range of DVFS. As the primary aim of policy is to reduce the power (thereby energy), DVFS is first used followed by DDCM.

In the combined policy (Figure 4.2), a core starts by using DVFS policy to lower frequency when its work corresponds to a frequency greater than or equal to the minimum frequency supported. Once the core is running at the minimum allowed by DVFS, and if it is determined that the clock rate should be further reduced, only then DDCM policy is applied. By using DVFS and DDCM together, effective clock rates up to 20% of maximum are possible before hardware glitches are seen.

[2]The minimum, maximum and intermediate values for DVFS are machine-dependent even for the same architecture, unlike DDCM.

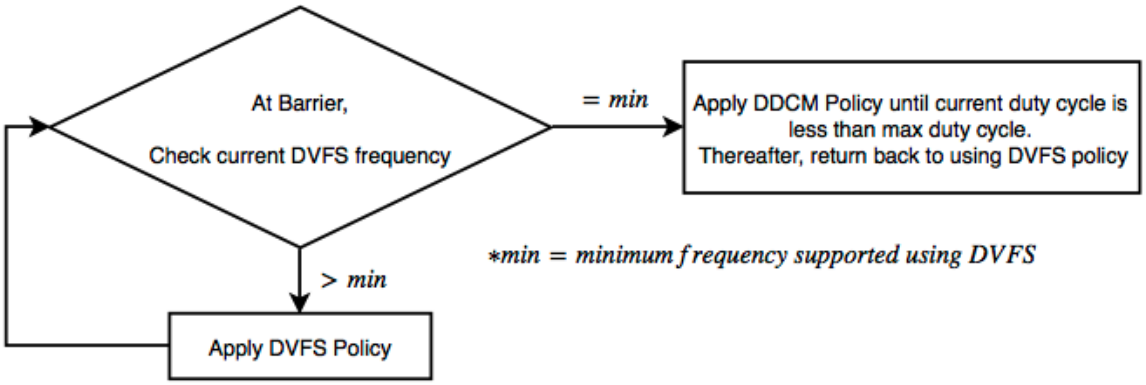


Figure 4.2: Working of the Combined policy that uses both per-core DVFS and DDCM

Once a core uses DDCM policy, it continues to use it every phase until duty cycle is increased back to 100%. Only then is the DVFS policy used. During highly unbalanced code regions, cores can have both DVFS and DDCM active, attempting to reduce the effective clock frequency as much as possible.

4.2.4 Adaptive Core-specific Runtime

The Adaptive Core-specific Runtime (ACR) implements the DVFS, DDCM and combined policies described above. It provides the user with a choice to select one of the three policies to control processor power usage, and in addition supports system-wide introspection through data reported from hardware counters. To avoid aggressive lowering of frequency/duty cycle that may cause unnecessary performance degradation a few modifications have been made to the policies in the ACR. These changes below can be easily overridden by the user to apply the policies in their purest form.

- *Frequency headroom aimed at minimizing execution time penalties*: The chosen clock-rate by default is rounded up to the next highest clock-rate. If the chosen value is too low the execution time penalty is seen to be larger than the potential energy savings.

- *Limit on the minimum permissible clock-rate*: For low clock-rates, observed performance degradation is observed to be higher than predicted. The minimum clock-rate is thus set to be at most 18.75% of the maximum non-turbo frequency, a value that is obtained empirically. This minimum is likely to be architecture specific and can be changed by using a set of environment variables provided.

ACR provides support for user options to facilitate user customization of the framework to fit specific use cases. The options below may allow further improvement in energy savings or limit performance degradation.

1. *Introduction of a limit on minimum phase length*: This consideration is to avoid frequency change decisions based on characteristics of smaller non-computational phases (like startup). This limit also prevents decisions from taking place too frequently, while the history is carried forward from skipped phases.

2. *Monitoring performance degradation at the end of every phase*: To minimize performance deterioration, a maximum flexible slowdown factor is introduced. It is expressed as a percentage value to monitor performance degradation. When the phase degradation in the last phase is greater than the specified value, the policy is skipped in the current phase and the frequency, as well as the duty cycle, are reset to maximum. In the next phase, the policy is applied with reset values. Additionally, this serves as a rudimentary way to reset clock frequency when a phase change is detected based on changes in total phase time.

3. *Support for user-annotations*: A user can easily override the preselected behavior of the runtime through environment variables.

Effects of OS noise and performance jitter that cause some applications to have irregular temporal patterns are somewhat smoothed by these user options and in practice, more predictable results have been observed.

ACR can be used directly by an application or embedded in libraries (e.g., MPI) to control energy with no application code changes. To measure energy, temperature and other execution metrics like frequency, it requires access to MSRs in user space through interfaces like "intel-rapl" kernel module and */sys/class/powercap/intel-rapl* or RCRdaemon (Porterfield et al. 2010).

The ACR interface for performance measurement is pinned to the first core in a socket, avoiding any interruption to other cores. Each call comprises only handful of straightline instructions, limiting the overhead to be smaller than the run-to-run variation in performance and are not detectable.

`MPI_Init` and `MPI_Finalize` calls are intercepted to setup and clean the infrastructure. During program execution, ACR uses one of three policies discussed earlier to set the best effective clock rate between MPI calls. `MPI_Barrier` and `MPI_Allreduce` are intercepted in the current experiments, but other MPI collectives can be easily used as well.

4.3 Infrastructure

A thirty-two blade partition of the Shepard Advanced Systems Technology Test Bed at Sandia National Laboratories is used for all experiments. This development partition exposes at the user-level power and energy instrumentation as well as grants user control of clock frequency and modulation through *msr-safe* and other kernel modules.

4.3.1 System

All tests used a portion of Penguin blade cluster. Each node has two Intel(R) Xeon(R) E5-2698 V3 CPUs, each with 16 cores, 128GB of memory at 2.3GHz with hyperthreading enabled[3] and connected with Mellanox Fourteen Data Rate InfiniBand. The maximum turbo frequency for the CPU is 3.6GHz. The cluster runs Red Hat Enterprise Linux Server 6.8 (Santiago), is scheduled by Slurm 2.3.3-1.18chaos and runs a Linux 3.17.8 kernel. MPI version used is Mpich 3.2.

4.3.2 Measurement Techniques

All reported power, energy and temperature numbers are obtained with the Intel Running Average Power Limit (RAPL) interface. To allow user-level access to the RAPL values of interest, the Resource Centric Reflection (RCR) daemon (Porterfield et al. 2010) is used. The RCRdaemon has been extended to provide additional performance related metrics associated with frequency, instructions retired and cache accesses.

Modern processors have enough internal heterogeneity that execution times often vary by several percent run to run (Porterfield et al. 2013a). The average is taken over 12 test runs for each power and software setting. To avoid energy variation with temperature, each test script ignored results from the first several minutes until the system temperature was stable.

4.4 Results

The evaluation of ACR uses a set of DOE mini-apps that encompass a variety of computation/memory patterns. Measurements are reported for the entire execution and not restricted to single phases. The applications can be divided into two groups.

[3]The hyperthread enablement has no effect on our experiment as applications are run only on one thread per core and frequency is changed by same amounts for both logical threads.

- *Mini-apps*: Six Mini-applications - five from the Mantevo Suite (Heroux and Barrett 2012) (*MiniFE*, *MiniGhost*, *CloverLeaf*, *miniAMR*, *HPCCG*) and one from the NERSC-8/Trinity Benchmarks (tri arks), *AMG*

- *Production application*: One production DOE application, *ParaDiS* - a free large scale dislocation dynamics simulation code to study the fundamental mechanisms of plasticity. It was originally developed at the Lawrence Livermore National Laboratory (Bulatov et al. 2004).

4.4.1 Mini applications

ACR attempts to act where load imbalance exists and remain dormant when work is evenly partitioned. The potential gain realistically achievable with ACR should occur when evaluating several HPC benchmarks with unbalanced workloads. For evaluation, a number of DOE MPI mini-apps were selected to simulate variety types of loads on HPC systems.

Application	**Default Time (s)**	**Policy**	**Min Phase Limit (ms)**	**Max Phase Degradation(%)**
miniFE	182	DDCM	none	none
		DVFS	10	5
		Combined	none	none
miniGhost	68	DDCM	none	none
		DVFS	none	none
		Combined	none	none
miniAMR	81	DDCM	none	none
		DVFS	none	none
		Combined	none	none
CloverLeaf	90	DDCM	none	0
		DVFS	none	none
		Combined	50	10
HPCCG	125	DDCM	10	10
		DVFS	100	10
		Combined	none	0
AMG	133	DDCM	10	10
		DVFS	none	none
		Combined	none	none
Paradis	123	DDCM	none	none
		DVFS	none	none
		Combined	none	none

Table 4.1: Execution time and ACR parameters for all applications on 32 nodes

Table 4.1 gives the execution time for default run without ACR for all applications. It also lists the ACR parameters used for runs that use ACR on 32 nodes. It can be observed that the policy works well without changing any ACR user options in most cases (represented as "none" value). Better energy efficiency while using ACR user options is obtained for some cases either by enhancing power reduction or controlling performance degradation as explained in Section 4.4.2. The values chosen for the user options in our experiments, especially for minimum phase length, is obtained empirically. We recommend using user-annotations supported by ACR to skip startup or non-computation phases in practice. A brief description of the mini applications used can be found in Section 2.5.

4.4.2 Impact of ACR user options

The impact of the user options on the Clock-Frequency policy is measurable. The performance impact on *HPCCG* as the user options are added is demonstrated with per-core DVFS policy in Figure 4.3. The base policy (A) results in a slight improvement in performance of 0.5%. Power is marginally reduced by 6.1%, to see an energy improvement of 6.5%.

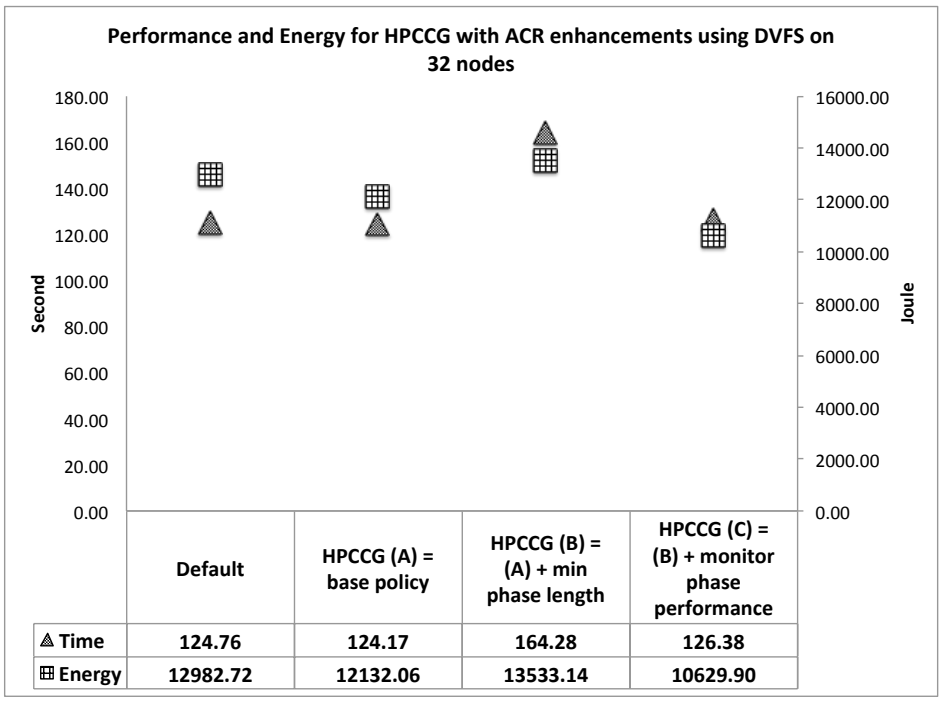


Figure 4.3: Execution metrics showing improved effectiveness of the policy through options in ACR

By forcing minimum phase length to be equal or greater than 100ms (B), power reduction is improved drastically to 20.8%. The increase in execution time though is extremely large at 31.7% that the energy consumption increases by 4.2%. For HPC applications high execution slowdown is problematic.

By limiting phase degradation (C) to 10%, performance slowdown is reduced to 1.3%. The power reduction remains very similar to (B) at 19.2%. With this option, the runtime attempts to (over-)react quickly at a phase change to prevent the critical core in the next phase from running at a clock frequency below 100%. If ACR detects a phase to run greater than 10% longer than the previous instance, it resets the core clock frequency to 100%. Even with aggressive clock frequency resets, the energy saved is still 18.1%.

4.4.3 Mini-application Results

The results for the mini-apps are in Figure 4.4. The best energy savings obtained for each application with ACR using either DDCM, DVFS or Combined is summarized in Table 4.2. The mini-apps fall into two broad categories. *miniFE* and *miniGhost* have significant imbalanced phases with a large number of memory references. DDCM reduces the clock frequency further than DVFS resulting in greater power savings. The effect of DVFS and DDCM on memory references was not studied. The combined policy provides the best results by allowing the voltage to also be lowered during the low clock frequency phases. *miniAMR* does not have a large number of memory references like the above two, yet it achieves highest energy savings with Combined due to large imbalanced phases.

The other mini-apps have better load balance and use DVFS's ability to lower the voltage resulting in lower energy usage than DDCM. The combined policy does not improve over DVFS for these mini-apps. The most likely cause is the overly aggressive use of DDCM during transition phases.

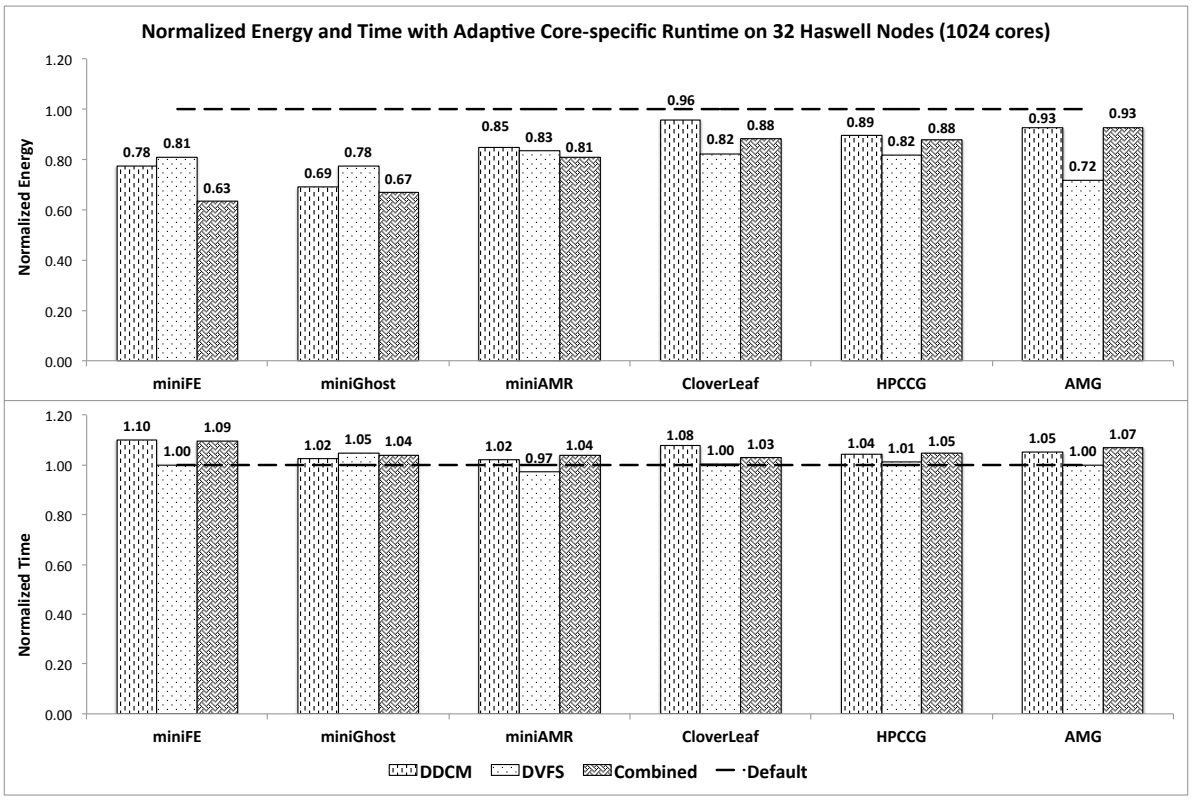


Figure 4.4: Energy consumption and execution times of ACR using DDCM, DVFS and both for mini applications

No.	Application	Power %	Energy %	Time %	Temperature diff	Avg. Frequency %	Policy
1	miniFE	58	63	109	-8	37	Combined
2	miniGhost	64	67	104	-6	45	Combined
3	miniAMR	78	81	104	-4	61	Combined
4	CloverLeaf	82	82	100	-4	84	DVFS
5	HPCCG	81	82	101	-4	79	DVFS
6	AMG	72	72	100	-1	72	DVFS

Table 4.2: The best energy savings obtained for each application with ACR using either DDCM, DVFS or Combined.

4.4.3.1 miniFE

For *miniFE* version using DDCM, the energy saving is 22.4% and program execution time is increased by 10.1%. With DVFS the energy savings is only 18.9%, but the slowdown is reduced to only 0.1%. When the combined policy is used energy savings increases to 36.7%. The energy reduction is achieved in spite of a 9.4% execution time increase through a 42.1% power reduction.

4.4.3.2 miniGhost

DDCM reduces energy on *miniGhost* by 31.0% with a 2.5% execution slowdown. DVFS increases execution time by 4.7% and lowers power by 25.9% resulting in an energy reduction of 22.4%. By combining the two policies, the performance penalty is only 3.8% and the energy savings increases to 33.1%.

4.4.3.3 miniAMR

miniAMR is an interesting example. With DVFS, execution actually speeds up slightly (2.7%). This combined with a power reduction of 14.4% results in it using 16.7% less energy. The speedup is consistent over multiple runs. It may result from the hardware moving power from the core saving energy to the core with the critical section, or it may be related to a better performance of the barrier when all processes arrive at nearly the same time (no thread is swapped out). The energy savings with DDCM and Combined are 15.3% and 19.1% with an execution time increase of 2.1% and 3.9% respectively.

4.4.3.4 CloverLeaf

This and the next two mini-apps with lower amounts of extreme imbalance all perform best with the DVFS policy. DVFS increases *CloverLeaf* execution by only 0.2%. This allows the energy reduction (17.6%) to effectively be equal to the power savings of 17.7%. In contrast, with DDCM the performance degradation is 7.7% results in only a 4.8% reduction in energy consumed. The combined policy does better than DDCM but still suffers a 2.9% execution time penalty and only reduces energy by 11.7

4.4.3.5 HPCCG

With DVFS, *HPCCG* is executed using 18.1% less energy. This savings is obtained with a time increase of only 1.3%. Both the DDCM and Combined policy see time increases of 4.5% and 4.9% respectively. The increased time results in energy savings of only 10.6% and 12.0% .

4.4.3.6 AMG

DVFS performs the best on *AMG*. Over a quarter of the energy is saved (28.2%), while only increasing execution time by 0.1%. The DVFS policy produced significant power/energy savings with a performance

impact less than typical run-to-run variation in execution time. DDCM and Combined policies were much less effective.

4.4.4 Production applications - ParaDis

With encouraging mini-app results, testing was expanded to a small real world (full) application (Figure 4.5) with parameters shown in the last row of Table 4.1. *ParaDis* does dislocation dynamics by introducing dislocation lines into a computational volume that interact and move in response to forces imposed by external stress and inter-dislocation interactions. The simulation run is "form_binaryjunc" with "fmctab.Ta.600K.0GPa.m2.t5.dat" correction table demonstrating the formation of a binary junction from two dislocation lines. There are 8x8x8 cells spread across 16, 8, 8 cores along x, y and z axes on 32 nodes. The discretization range is [5.000000e+01, 2.000e+02] and re-mesh method used is 3, with maximum steps equal to 100. With load balancing turned off, *ParaDis* provides an unbalanced small real-world application where number of timesteps can be adjusted to create short enough runs for extensive testing.

Initial testing with *ParaDis* on 1024 cores yielded encouraging results (Table 4.3). With the chosen number of time steps default case on average executed in 122.6 seconds. DDCM lowered power 19% but took on average 5% longer to execute, while DVFS also reduced 19% power but ran in 122.3 seconds showing no performance degradation. The Combined policy takes only 108.7 seconds on average. The optimization meant for power reduction, also *decreased* the execution time by 11%. When combined with the 31% reduction in power the best total energy savings is 42%.

Configuration	**Power (W)**	**Energy (J)**	**Time (s)**	**Average Frequency (MHz)**	**Metrics compared with default**			
Default	94.2	11541.1	122.6	2272.8	***Power %***	***Energy %***	***Time %***	***Frequency %***
DDCM	76.5	9865.5	129.0	1531.5	81	85	105	67
DVFS	76.3	9330.9	122.3	1837.7	81	81	100	81
Combined	65.0	7058.1	108.7	1321.7	69	61	89	58

Table 4.3: Execution metrics for ParaDis while using ACR with DDCM, DVFS and both on 32 nodes (1024 cores)

Upon closer inspection, a large amount of run-to-run variation is present in the 1024 core runs. Figure 4.6 graphs the performance of 12 runs for each of the energy policies. The default runs have a 30% run-to-run variation, from a low of 105 seconds to high of 136 seconds. When using the Combined policy, variation

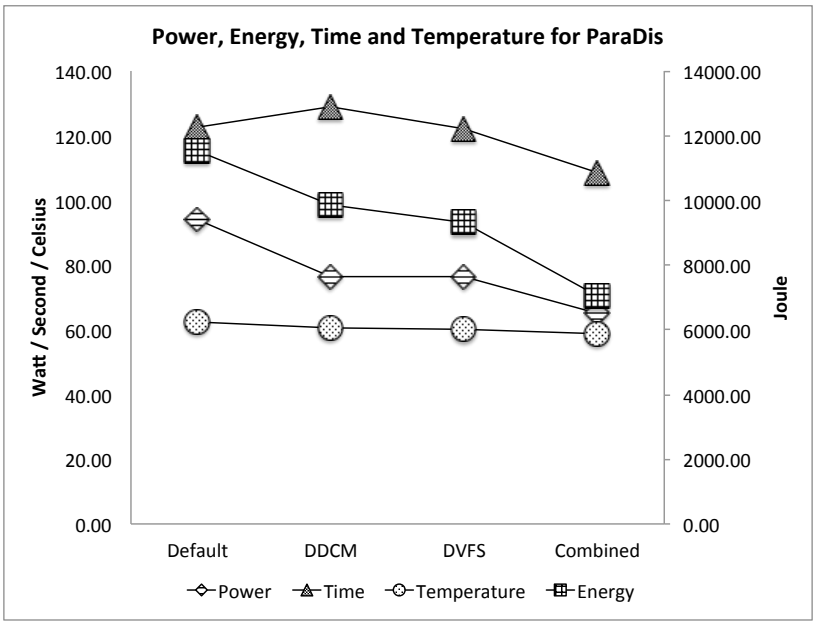


Figure 4.5: Execution metrics for ParaDis while using ACR with DDCM, DVFS and both on 32 nodes (1024 cores). Note that the values are discrete with links only serving as visual cues.

was reduced to about 5%, and execution times clustered around the fastest observed for default execution (between 105 seconds and 111 seconds).

4.4.5 Understanding performance improvement for ParaDis

To better understand performance improvement seen with *ParaDis* its critical path behavior for the Default (no ACR), DDCM and DVFS cases on 24 nodes (768 cores) is shown in Figure 4.7[4]. The single run chosen has the worst execution time out of 12 runs for each of the three cases. The values in subtitles denote average values across the entire execution of the run, while the values in legend are average values taken only across data shown in the plot. A core with the highest compute time per phase is considered as the critical core. Critical cores with compute times shorter than 0.1s are discarded to avoid large average frequency values computed using Intel APERF and MPERF counters. Only cores running at average frequencies greater than 2200MHz are considered to ensure that the critical cores run at the maximum possible frequency and do not experience any undue slowdowns (due to policy mispredictions). Consequently, a lower percentage (89%, 74%, and 70%) of the actual critical execution is captured in Figure 4.7.

[4]All 32 nodes in the partition were not available during profiling

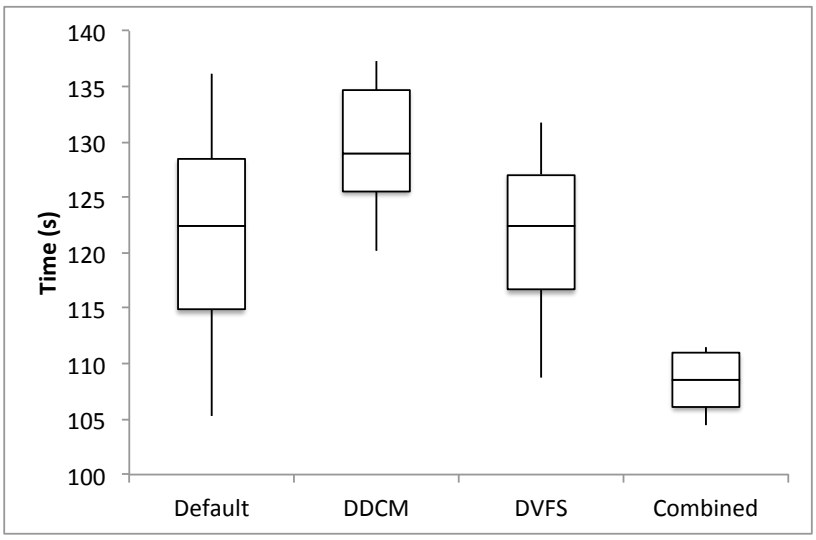


Figure 4.6: Variation in execution time of *Paradis* across all cores for 12 runs on 32 nodes

On 24 nodes *ParaDis* takes about four times longer to complete in the default case and shows a lot higher workload imbalance than on 32 nodes. As a result with DDCM, the execution time for *ParaDis* is reduced from 405.0s to 265.2s, a reduction of 34.5% (compared to 11% on 32 nodes). For the Combined case (not shown in Figure 4.7), the execution time is lowered by 35.4% (261.6s) and the power by 28.9% (65.5W) for a total energy savings of 54.1% (17127.8J). The DVFS case, though, shows only 1.3% performance improvement running for 399.7s. This indicates that the performance improvement for the Combined case is mainly due to DDCM, and not DVFS. As in the case of 32 nodes, the run-to-run variation is seen to be greatly reduced with ACR on 24 nodes to suggest conformity between the two execution profiles.

By analyzing the critical path behavior in Figure 4.7 the speedup for *ParaDis* can be explained using two key factors:

Reduction in run-to-run variation: The two dashed lines in each plot trace the means of a bimodal distribution of critical path times. In successive phases work appears to be similar, with occasional jumps between short and long critical paths. The consistency suits ACR as the frequency for the non-critical cores can be lowered to very low values for prolonged periods. Hence, non-critical cores do not compete with the critical core for resources during a phase. This alleviates any existing contentions to explain the reduced run-to-run variation. Further, the regular work pattern reduces mispredictions in all ACR policies.

Turbo mode: Lowering the frequency of non-critical cores for prolonged periods increases the available thermal headroom making critical cores with ACR using DDCM to run at higher turbo frequencies (2784.8MHz) compared to default (2507.4MHz). Because turbo frequencies are disabled when DVFS

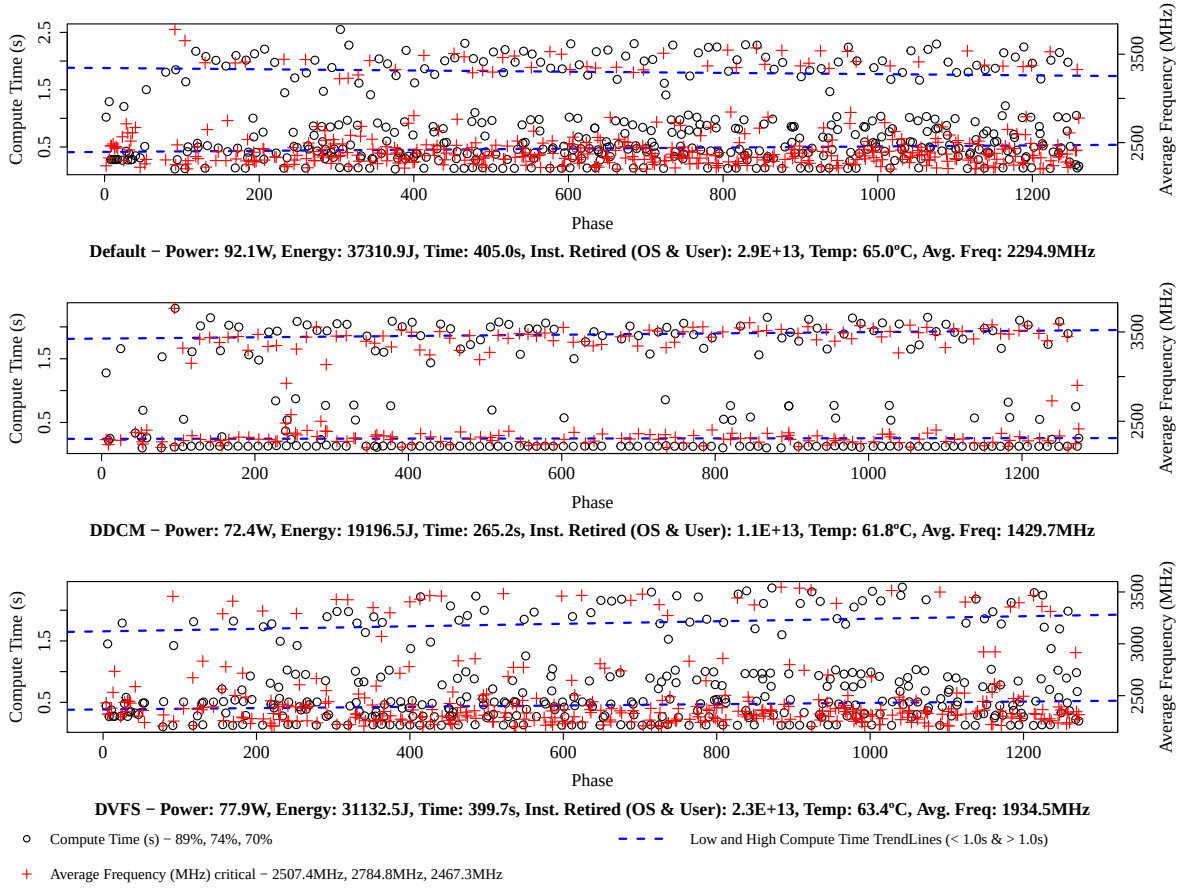


Figure 4.7: Critical Path Behavior for *ParaDis* running on 24 nodes (768 cores).

is in operation, critical cores run at much lower frequencies (2467.3MHz) resulting in low performance improvement, if any. Critical cores running at turbo also reduce the impact of policy mispredictions during phase transitions compared to DVFS (E.g. 75% of 2.6GHz (turbo) with DDCM $>$ 75% of 2.3GHz with DVFS). The average count of instructions retired at OS and User level by each core using DDCM (1.1E+13) is only one-third compared to default (2.9E+13) due to lowered busy waiting. The reduction in busy waiting again can be mostly attributed to critical cores running at turbo with DDCM as this effect is not seen in the case of DVFS (2.3E+13). Finally, even though the power is reduced substantially (21.4%) with DDCM and average frequency across cores is only 1429.7MHz the temperature is not reduced comparably (only 3.2°C reduction). This indicates work of turbo, as the heat dissipation is non-linear.

4.5 Discussion

Table 4.4 summarizes the energy savings and other related metrics obtained by each of the policies with mini applications and full application *ParaDis* on 1024 cores. The least performance degradation of 0.5% across all applications is obtained with DVFS. It also reduces power by 20.5% to achieve a commensurate average energy efficiency improvement of 20.2%. The best energy savings overall is achieved with Combined at 22.6% with a power reduction of 24.9%. However, the execution time increases to 2.9%. The energy improvement with DDCM is 15.1% with power reduction of 19.3% and execution time penalty of 5.3%.

Policy	**Avg %Power**	**Avg %Energy**	**Avg %Time**	**Avg Temp diff**
DDCM	-19.3	-15.1	5.3	-3.2
DVFS	-20.5	-20.2	0.5	-3.3
Combined	-24.9	-22.6	2.9	-4.2

Table 4.4: Summary of energy savings and other metrics obtained with each policy for mini applications and *Paradis*

The intent of the above comparison table is not to help the decision of choosing one policy over the other, but only to summarize the effectiveness of ACR with the three policies. Figure 4.4 and Figure 4.5 show unique characteristics of each policy depending on the nature of the application. For applications that show extreme workload imbalances and/or high memory references the Combined or DDCM policy work best. And in some cases (*ParaDis*) with improved performance. For applications showing more moderate imbalance, DVFS works better. The higher average time penalty seen by Combined and DDCM in Table 4.4 is mostly due to the higher performance deterioration observed with applications that are more stable.

By tailoring the frequency of each core to match its work, slack as well as the power wasted is greatly reduced. Reducing the clock frequency for hardware threads spending significant time at software barriers results in valuable energy savings and in most cases will be invisible to the users, as the execution delay is well below the variance already observed during execution. With the reduction in power, ACR shows a corresponding reduction in the chip temperature for all applications, reducing cooling requirements.

4.6 Conclusion

ACR uses hardware core-specific power control mechanisms and an adaptive software policy to achieve significant energy savings with minimal execution time penalties. It provides, for a number of DOE mini-apps

and small applications, 20+% energy savings with performance within the normal run-to-run variation. With no application code modifications, ACR provides significant energy savings with no user-visible effects. For one application case (*ParaDis*) a significant performance improvement is observed due to the reduction in run-to-run variation and execution of critical path cores at turbo frequencies with DDCM. This shows evidence in proving measurements and controls local to the core can on average reduce power at runtime with little performance impact.

As Exascale deploys over-provisioned systems that use per core power-limits in day-to-day operations, energy optimizations will be more important. Runtimes such as ACR will either allow more work to be run at one time by using less power or allow single applications to be run faster by allowing a higher power cap on critical cores than non-critical. On power-limited systems, power (and energy) optimizations will be critical. ACR demonstrates that adaptive dynamic control of power at runtime is possible.

In the future, a better understanding of the advantages and disadvantages of ACR is needed. At some scales, ACR results in a significant performance improvement. As a downside, at other configuration sizes (and user options) ACR results in slowdowns. A better understanding of when the chosen clock frequency is too low and how to correct it quickly is required before a system such as ACR can be deployed in a production environment.

CHAPTER 5: Improving Energy Efficiency in Memory-constrained Applications

5.1 Introduction

Disparity between CPU and memory speeds and the latency of accessing DRAM across the system bus contribute significantly to CPU cycling while waiting on memory and wasting power. Many memory operations are not visible to the operating system (OS) and are not sufficiently coarse-grained for the hardware circuitry to stall (or switch off) cores and reduce power as memory accesses are serviced. For certain classes of applications that are memory-bound, reducing the processor speed or using related approaches like CPU throttling for power savings has shown little adverse impact on performance, or in some cases slight speedup from reduced contention (Porterfield et al. 2013b, Wang et al. 2015). The last few chapters discussed techniques targeting computational workload imbalance, however waiting on memory and sometimes even I/O is another important problem faced by many HPC applications. This chapter focuses on detecting situations in which an application is memory-bound during execution and using core-specific power controls to reduce CPU power consumption.

Most research to regulate energy and performance in software has revolved around Dynamic Voltage and Frequency Scaling (DVFS) (Kimura et al. 2006, Kappiah et al. 2005, Rountree et al. 2009). Because of past hardware limitations, these previous DVFS methods impacted all cores on a multi-core processor and potentially slowed the critical path. Those research efforts focused on finding situations where the slowdown is greatly outweighed by the energy savings. The chip-wide effect of DVFS also made effective fine-grain control of performance difficult. With the introduction of per-core voltage regulators in Intel Haswell, each physical core (or pair of logical cores when using Hyper-Threading) can be independently controlled using software as discussed in this research to target slowing of only non-critical threads.

Recent advancements in memory technologies like the introduction of double data rate fourth-generation synchronous dynamic random-access memory (DDR4 SDRAM), among others, have attempted to close the gap between CPU and memory processing times. It is important to ascertain the severity of the memory bottleneck in modern HPC systems, to see if there exist opportunities to pursue energy efficiency research

addressing that bottleneck. We analyze the memory bandwidth and latency on modern CPU architectures like Intel Sandy Bridge and Haswell with different memory configurations. The study reveals bandwidth saturation and increased latency situations where CPU frequency can be reduced without affecting performance. Several on-board hardware counters are evaluated to determine a usable metric that can detect the above situations during execution, and form a basis for a runtime control policy. The metrics are then used to characterize HPC applications based on their memory activity. and a dynamic policy to lower frequency of individual cores using per-core DVFS on Haswell machines, is used to reduce power and save energy with minimal performance impact.

This chapter introduces the following main results:

- An experimental memory study on modern CPU architectures, Intel Sandy Bridge and Haswell, with different memory configurations.
- A memory characterization of HPC applications using on-board hardware counters to identify metrics that guide application of power controls.
- A dynamic policy with coarse and fine-grained application modes that uses the TORo_core metric to direct per-core DVFS during program execution. Policy validation is performed using six mini-apps (*HPCCG, AMG, CloverLeaf, miniFE, miniGhost, miniMD*). The evaluation shows the best energy savings with coarse and fine-grained versions is 32.1% and 19.5% respectively. Also, the performance is seen to improve in a few cases, more often with the fine-grained version.

5.2 Memory performance on modern HPC systems

The memory study undertaken is similar to the one in our previous work (Mandal et al. 2010) on Dual and Quad-socket AMD Opterons and Intel Nehalem. Recent HPC systems are evaluated to better understand the improvement in memory performance over previous generations. The goal is to determine when memory bottlenecks exist.

5.2.1 PCHASE Benchmark

PCHASE (Pase 2008) is specifically used to test memory throughput under carefully controlled degrees of concurrent access. Each thread executes a loop with a controllable number of independent “pointer

chasing" operations per iteration. Each sequence of pointer addresses is pseudo-random and designed to defeat hardware prefetching while limiting TLB misses. A wrapper script around PCHASE is added that iterates over different numbers of memory reference chains (thereby controlling memory concurrency) and threads for each PCHASE run. PCHASE was also modified to control thread placement on multi-socket, multi-core systems. This forces sequential memory accesses within a thread. References from different chains can be issued concurrently. A PCHASE experiment specifies the number of miss chains followed by each thread. The chains within a thread are independent and interleaved, so their references can be resolved concurrently. An experiment also specifies the number of threads that access memory concurrently. By varying the number of chains, number of threads and placement of threads, contention in most parts of the downstream path and components can be measured.

Threads that run on different cores will use different paths to memory and may use different memory components. While concurrent references within a thread all use the same paths, references from threads on distinct cores and sockets use distinct on-core components, but will interleave and contend with one another in the downstream paths and components of the system.

Each experimental run of PCHASE is parametrized by (a) memory requirement for each reference chain, (b) number of concurrent miss chains per thread and (c) number of threads. The other parameters (page size, cache line size, iterations, access pattern etc.) are kept fixed.

5.2.2 Experimental Setup

The memory performance is obtained by running PCHASE on Sandy Bridge (SB16) and Haswells (HW20, HW32).

SB16 is an M620 Dell Blade with two Intel Xeon E5-2680 CPUs at 2.7GHz containing eight cores each and 64 GB of main memory. It has 20MB cache per CPU (40MB total). Each DDR3 DIMM is 4GB in size, with 8 out of the 12 available slots for each CPU populated. This configuration allows the memory to run at its maximum speed of 1600MHz.

HW20 is an Dell PowerEdge R730 with two Intel E5-2650v3 CPUs at 2.3GHz containing 10 cores. The LLC size is 25MB per CPU (50MB total). The DDR4 memory with a chip size of 4GB is placed in 4 out of the 12 slots available for each of the CPU (32GB total). The maximum memory speed is 2133MHz. Hyperthreading has been disabled on SB16 and HW20, and both run Linux kernel 2.6.32.

The 32-core Haswell machine (HW32) runs Linux kernel 3.17.8. It has two Intel(R) Xeon(R) E5-2698 V3 CPUs, each with 16 cores at 2.3GHz and hyperthreading enabled. The total memory is 128GB and cache size is 50MB on each. The DDR4 16GB chip is populated in 4 out of the 8 slots available and clock at maximum speed of 2133MHz.

5.2.3 Results

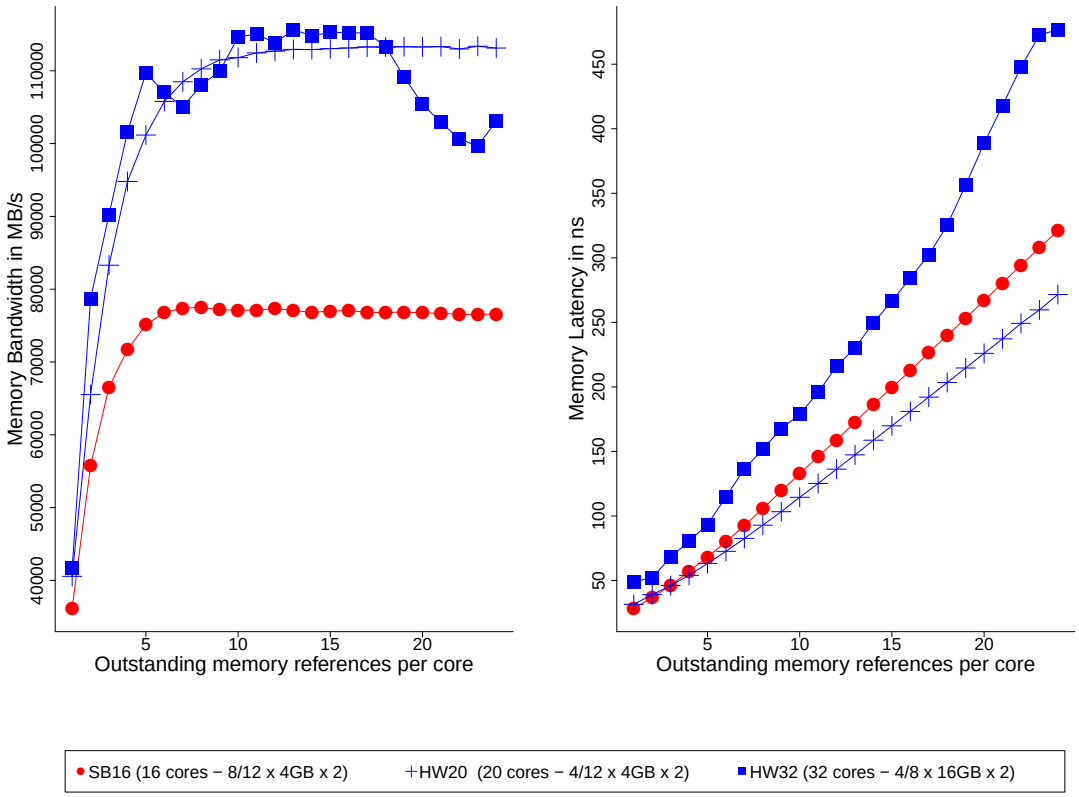


Figure 5.1: Memory Bandwidth & Latency using PCHASE on Sandy Bridge (SB16) and Haswell (HW20, HW32)

The memory utilization as the number of outstanding requests change per core is measurable. Figure 5.1 shows total memory bandwidth (in MB/s) and effective latency (in ns) with varying numbers of outstanding memory references. For each system, the lowest bandwidth and latency is found with one memory reference per core. Bandwidth increases almost linearly without increasing latency as the system becomes better utilized at two references per core. As the number of references per core increases beyond two, queuing and contention within the system increase latency.

The reason to include a Sandy Bridge (older) machine in this work is to understand the improvements leading to Haswell. A considerable improvement in bandwidth is observed from SB16 to HW20 & HW32 emphasizing the benefit of DDR4 over DDR3 memory. Better bandwidth does not always guarantee better latency, which is subject to core count on chip (Figure 5.1). HW20 is seen to be the fastest in terms of latency above three outstanding references per core. The gap increases slowly as the number of outstanding references increases. In contrast, HW32 is not only slower than HW20, but is also slower than SB16. It is slower at all concurrency levels and the disparity increases as contention increases. Looking back at the figure reveals that both the Haswell systems have similar effective latencies for an equal total number of references. For example, the effective latency for HW20 with 16 outstanding memory references per core (320 in total) is similar to that of HW32 with 10 outstanding memory references per core. Core count and type/number of DIMMs is still an important factor when determining the overall memory performance. An poorly provisioned newer chip can have higher latency than an older chip.

Peak bandwidth on all three machines is achieved with 5-7 outstanding memory references per core. Beyond this, the memory bandwidth no longer increases, but the effective latency continues to rise. This is because the references queue up after the bandwidth saturation thereby increasing the stress on the memory system. For example, at 24 memory references per core the latency (476.83ns) on HW32 is about 10 times that of at one memory reference (49.13ns). Even though hyperthreads weren't used, a non-uniform memory bandwidth curve is seen for HW32 beyond 5 memory references. The low level at which memory bandwidth reaches saturation and consequent increased latency presents an opportunity to improve energy efficiency by reducing cpu frequency when the offered memory load exceeds saturation level. This motivates use of uncore metrics to determine memory load and dynamically control power and frequency of CPUs.

5.3 Infrastructure

The modern Intel architectures can monitor performance of both core and uncore components. The uncore sub-system of Intel architecture consists of caching agent (CBo), power controller unit (PCU), integrated memory controller (iMC) and home agent (HA) among other components. It facilitates per-component performance monitoring (PMON) through sets of counter registers. In this chapter, we use the counter registers that are part of the CBo to monitor all core transactions that access the Last Level Cache (LLC).

The transaction monitoring is supported using 44-bit wide counters (**Cn_MSR_PMON_CTR{3:0}**) (Intel 2015b). We track the Table of Requests (TOR) queue for pending CBo transactions.

All reported power, energy and temperature numbers are obtained with RAPL through the RCRdaemon. RCRdaemon has been extended to provide additional performance related metrics associated with frequency, instructions retired and cache accesses. The average is taken over 12 test runs for each power and software setting. To avoid energy variation with temperature, each test script ignored results from the first several minutes until the system temperature was stable.

A number of DOE MPI mini-apps from the Mantevo suite (Heroux and Barrett 2012) and NERSC-8/Trinity benchmarks (tri arks) were selected to simulate a variety of loads on HPC systems. The description for these can be found in Section 2.5.

5.3.1 System

A thirty-two blade partition of the Shepard Advanced Systems Technology Test Bed at Sandia National Laboratories is used for all memory characterization and policy experiments. This development partition exposes power and energy instrumentation along with control of clock frequency and modulation through *msr-safe* and other kernel modules. The Penguin nodes are connected with Mellanox Fourteen Data Rate InfiniBand and have configuration similar to HW32. The hyperthread enablement has no effect on our experiment as benchmarks are run only on one thread per core and frequency is changed by same amounts for both logical threads. The cluster runs Red Hat Enterprise Linux Server 6.8 with Mpich 3.2 and is scheduled by Slurm 2.3.3-1.18chaos.

5.4 Characterizing memory behavior

As uncore performance monitoring can capture numerous events in various components, choosing an appropriate metric that detects memory saturation and latency increase presents a challenge. With shared LLC, each core has to create requests for memory locations not in its private cache into the Table of Requests (TOR). Hence, it was reasonable to begin the search to find our metrics by analyzing the events associated with LLC, particularly TOR. From the many events monitored in the CBo, three memory metrics captured using **Cn_MSR_PMON_CTR{3:0}** were chosen. The metric value from each core is summed for the entire

length of the application and then divided by the number of memory updates that happened in the RCRdaemon during this period.

The first metric, TOR occupancy (TORo) captures all valid requests in TOR, including those that reside even for a short time, such as LLC Hits that do not need to snoop cores or requests that get rejected and have to be retried through one of the ingress queues.

$$TORo = \frac{\sum_0^t \sum_{i=0}^n (TOR\_Occupancy_i / Clock\ cycles_i)}{Number\ of\ Memory\ updates}$$

The second metric, RR occupancy (RRo) is the average number of entries in the Ingress Request Queue (IRQ) on Address (AD) Ring. The AD Ring is associated with core read/write requests and Intel QPI Snoops. It also carries Intel QPI requests and snoop responses from core to Intel QPI. This metric is supposed to be a subset of TORo such that applications with high R×R_Occupancy generally have higher TORo.

$$RRo = \frac{\sum_0^t \sum_{i=0}^n (RR\_Occupancy_i / Clock\ cycles_i)}{Number\ of\ Memory\ updates}$$

The final metric, LLCv is the average number of lines that are victimized on a LLC fill.

$$LLCv = \frac{\sum_0^t \sum_{i=0}^n (LLC\_Victims_i / Clock\ cycles_i)}{Number\ of\ Memory\ updates}$$

In the above two equations; *n* is the number of cores per socket, *t* is the running time of the application. The memory value in the RCRdaemon is updated every 1ms.

The results for memory characterization of the chosen applications on 32 Haswell nodes (1024 cores) is shown in Figure 5.2 along with average power consumption and temperature. TORo, RRo and LLCv are dimensionless ratios.

The metrics TORo and RRo classify the applications along a spectrum with cache/memory bound applications on the left of the plot and compute bound applications to the right. For *miniGhost* and *miniMD* that are completely compute intensive applications the two metric values are 0. LLCv is not a viable metric for characterizing application as compute or memory bound as even for highly memory constrained applications – *CloverLeaf, HPCCG and AMG* the value for LLCv is 0. Overall both TORo and RRo are able to support characterization/classification of the applications. However, TORo has better resolution for applications in

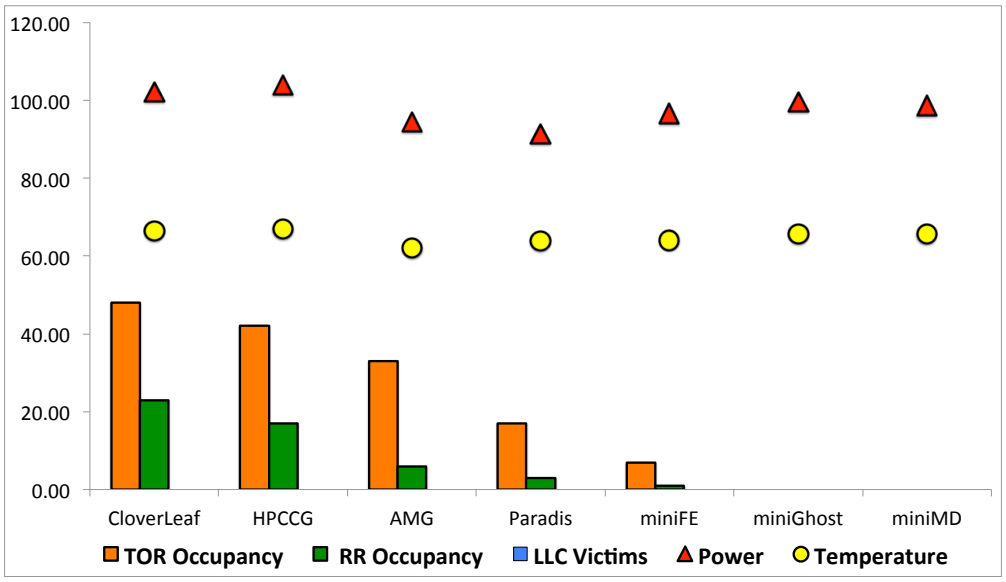


Figure 5.2: Memory characterization of applications on 32 Haswell Nodes (1024 cores)

the middle of the spectrum. TORo is the best metric for use in the policies to capture the opportunities seen in the evaluation with PCHASE.

5.5 Runtime Policy

The insights from characterization were used to formulate a dynamic policy based on TORo to control power. The policy is implemented within the MPI profiling interface (PMPI) and requires no application code changes. Calls to `MPI_Init`, `MPI_Finalize` are intercepted to start collection of power, memory and other metrics using RCRdaemon, while other MPI calls like `MPI_Send`, `MPI_Recv` among others are intercepted to collect TORo values and make policy decisions. Since the policies are core-specific, the TORo value collected by the RCRdaemon at a core level between the MPI calls is used (denoted by *TORo_core* hereafter) eliminating the need for communication with other MPI ranks. The policies, as in the previous chapters, are embedded in a shim library that uses PMPI.

Figure 5.3 describes actions taken at each MPI call by the dynamic policy in coarse and fine-grained versions. The basic idea in both the versions is that when the memory load associated with a core is high at a MPI call, it is better to slow down that core as the performance is no longer dependent on the speed of core, but on the memory access time. In contrast to the policies in Chapter 3 and 4, here the decision on processor speed is taken at the beginning of the MPI call rather than at the end. For the coarse-grained policy,

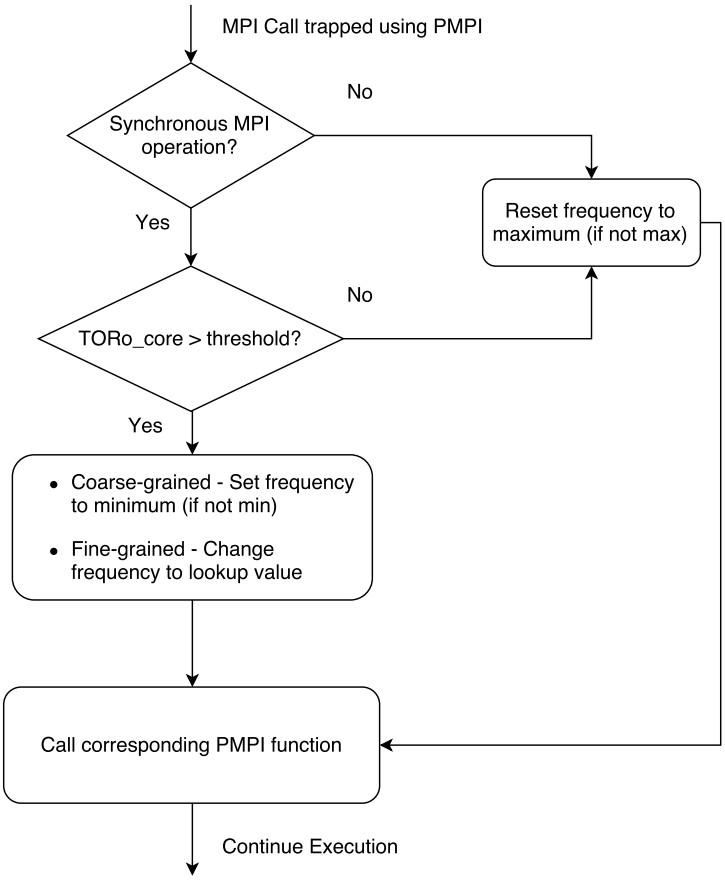


Figure 5.3: Flowchart of the dynamic policy using TORo_core as the metric to throttle power using per-core DVFS

the threshold value is empirically obtained and set at the beginning of execution. The fine-grained dynamic policy uses multiple threshold values to change the core frequency.

5.5.1 Coarse-grained application

The coarse-grained dynamic policy (CDP) targets applications that are well-known to be memory intensive like *HPCCG* (Figure 5.2). For empirically determining the threshold value for evaluation, HPCCG and miniMD were run on HW32 with different TORo_core threshold values as shown in Figure 5.4. Initially, the applications run in their default (D) configuration with power, execution time and energy consumption recorded. Cores for which the TORo_core value is above the specified value are controlled to run at their lowest frequency (1.2GHz). The threshold value is varied from 0 (almost all cores run at lowest frequency for the entire length of execution) to 10 (almost all cores run at max frequency).

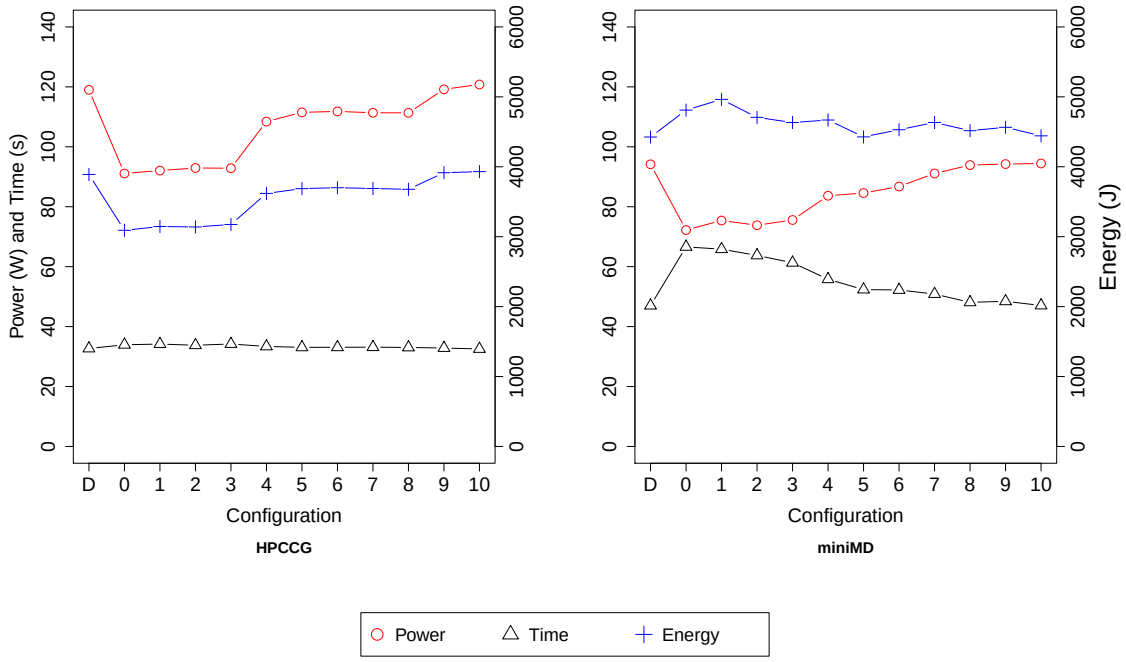


Figure 5.4: Effect of using different TORo_core threshold values on HPCCG (memory constrained) and miniMD (compute bound) application energy and execution time within coarse-grained dynamic policy (CDP). Note that the values are discrete with links only serving as visual cues.

For *HPCCG*, which is heavily memory constrained (Figure 5.2), there is very little performance impact when running the application at lowest frequency for the entire duration. Consequently, at TORo_core=0, the application power is at its lowest point at 91.1W (30.6% reduced) along with the energy consumption (3091.4J – 25.8% reduction) as the increase in execution time is marginal at 3.7% (33.94s). At TORo_core=4, the application power consumption increases drastically to 108.4W and the execution time increase at this point is reduced to 1.6%. This suggests that for a highly memory-bounded application like *HPCCG* , the majority of its memory references are concentrated at a value of less than four on the TORo_core scale. Between a TORo_core value of five to eight, the power and performance is almost flat followed by drastic increase in power at TORo_core=9 where the values are comparable to default. This indicates there exists a second level of concentration for the memory references.

The compute-bound application (Figure 5.2), *miniMD*, sees a steep increase of about 16% in execution time at TORo_core=0. Even though the power is reduced by 30.4%, the energy increases by 8%. Each increase in threshold level thereafter is seen to reduce the performance impact with increasing power consumption. At TORo_core=10, the values match that of the default configuration. The prevailing phenomenon is

indicative of the direct impact of frequency reduction on performance of a compute bound application. For highly compute bound applications like *miniMD*, the TOR_core scale is not suited to improve energy efficiency, and such applications are best left alone by memory-centric schemes like the ones discussed here. Computational-centric policies discussed in Chapter 3 and Chapter 4 may be more suitable for such applications.

For HPCCG the lowest energy point is TORo_core=0. The PCHASE evaluation (Figure 5.1) shows that the memory bandwidth is no longer linear beyond two outstanding memory references. As the energy consumption curve is almost flat up to TORo_core=3, we choose a value of 1.0 (one outstanding request in the TOR per core). The value of TORo_core=1.0 also allows some headroom for the frequency decisions avoiding overfitting. However, the value for TORo_core can be set to any desired value at runtime using environment variables.

5.5.2 Fine-grained application

Figure 5.4 reveals that memory references are concentrated on multiple levels of the TORo_core scale with steps at TORo_core=4 and TORo_core=9. The fine-grained dynamic policy (FDP) aims to take advantage of this information to have a much finer control on frequency to make the TOR_core-scale based policy broad. This further helps the policy to target applications that lie between the extremes of the memory characterization spectrum by controlling degradation (Figure 5.2). The memory references for a memory intensive application are spread across different values on the TORo_core scale with regions of concentration. The DVFS operational range levels are mapped to TORo_core values and the frequency is set to the value at a particular TORo_core value during execution. The effective average TORo_core value at which the core has to be slowed down is determined during execution. For example, in our evaluation between a TORo_core value of 0 to 0.5 the mapped core frequency is 2.3GHz (max non-turbo frequency supported). Beyond a TORo_core value of 9.0, the mapped core frequency is 1.2GHz. During execution the core frequency ranges from 1.2 to 2.3 GHz. The effective frequency at which the core needs to run for an application is adaptive, making the performance deterioration lower. In contrast, with **CDP** the frequency reduction is fixed and is set at the beginning of execution.

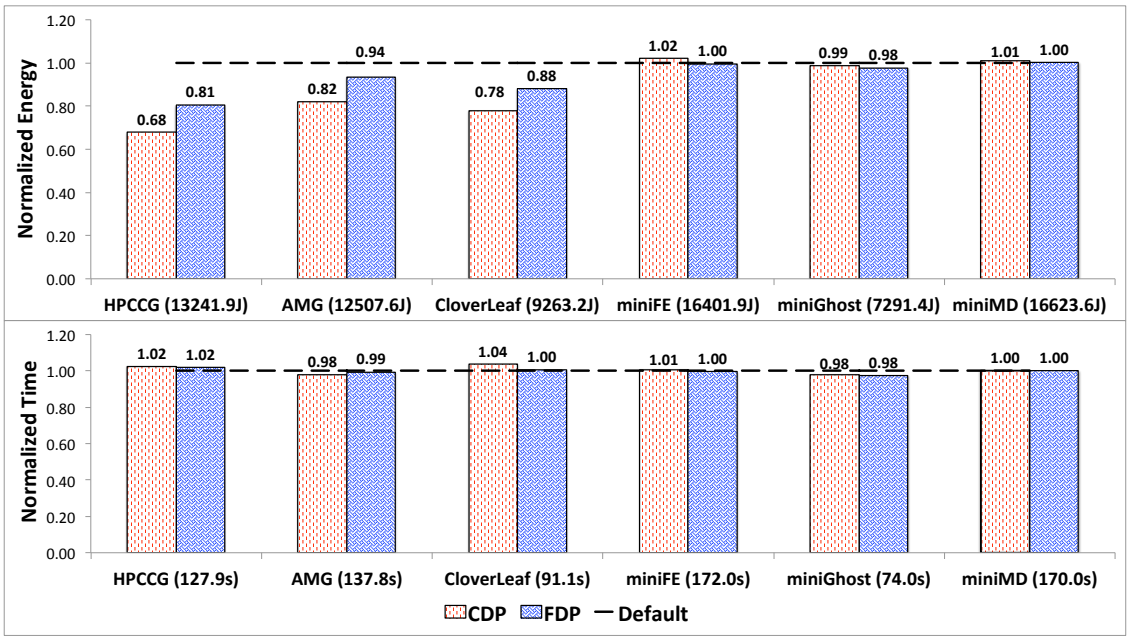


Figure 5.5: Energy consumption and execution times using coarse (CDP) and fine-grained (FDP) dynamic policy for mini applications

5.6 Results

The result for mini-applications with the two policies is shown in Figure 5.5. The energy consumptions and execution times are normalized and the default values for each application is provided. At the onset, the policies are seen to target applications that are only memory constrained and not compute-bound. The most memory constrained application – HPCCG is seen to have largest reduction in energy with both policies. While the least memory constrained – miniGhost and miniMD are mostly untouched. This validates TORo_core as the right metric for controlling the behavior to improve energy efficiency seen in Figure 5.1.

For *HPCCG* using FDP, the energy saving is 19.5% and program execution time is increased by 1.9%. With CDP the increase in execution time is slightly more at 2.4%, but the reduction in energy is 32.1% owing to large power reduction of 33.7%. The large energy efficiency improvement with CDP can be mostly attributed to its high memory-intensive nature.

CDP not only reduces energy consumption (17.8%) for *AMG*, but also improves performance by 2.2%. Even with FDP the performance improves by 0.6% saving 6.4% in energy. These cases demonstrate that energy optimization improves performance in certain scenarios.

CloverLeaf is an interesting example where we see that even though its average TOR occupancy is the highest (Figure 5.2), the energy savings is much lesser than *HPCCG*. It reduces the power with CDP by up to 24.7%, however the execution time is increased by 3.6% resulting in energy savings of 24.7%. Using FDP, we are able to reduce the performance degradation up to 0.4% saving 12.0% in energy. This shows the benefit of FDP to control performance deterioration within permissible levels of run-to-run variation. Profiling *CloverLeaf* revealed a large presence of asynchronous calls (`MPI_Isend` and `MPI_Irecv`). The frequency at which these calls run determines execution time. To overcome this, both policies were slightly modified to run the asynchronous calls at 1.8GHz instead of 2.3GHz. The energy reduction without slowdown in case of FDP suggests that even though asynchronous calls are able to hide memory latency to an extent, there exist further opportunities to improve energy efficiency.

Applications *miniFE*, *miniGhost* and *miniMD* are mostly cache/compute bound (Figure 5.2). Neither policy reduces power or adversely impacts performance, as necessary for a policy meant to only target memory-constrained scenarios.

miniFE sees a slight increase in energy (2.1%) owing to increase in power (1.4%) and execution time (0.7%) using CDP. With FDP the increase in execution time is nullified and the execution time and power is close to default. It is important to note here that even though the load_imbalance=100, there is no change in its memory footprint. It continues to remain compute-bound and is unaffected by the dynamic policy.

miniGhost performance improves with both CDP (2.0%) and FDP (2.5%). The power slightly increases with CDP (0.6%) therefore the energy saving is 1.4%. With FDP power is on par with default and energy savings commensurate with speedup (2.5%), showing yet another scenario showing energy optimization leading to performance improvement.

For *miniMD* using CDP, power is increased slightly by 0.8% and execution time by 0.1% to increase energy consumption by 0.9%. The FDP runs at same speed as that of default and consumes similar power.

Overall, **FDP** saves considerable amount of energy at lower performance degradation even with smaller power reduction compared to **CDP**. It is more suitable for applications whose memory behavior is not well known, as CDP may adversely impact performance. The best energy savings with CDP and FDP are 32.1% and 19.5% for *HPCCG*.

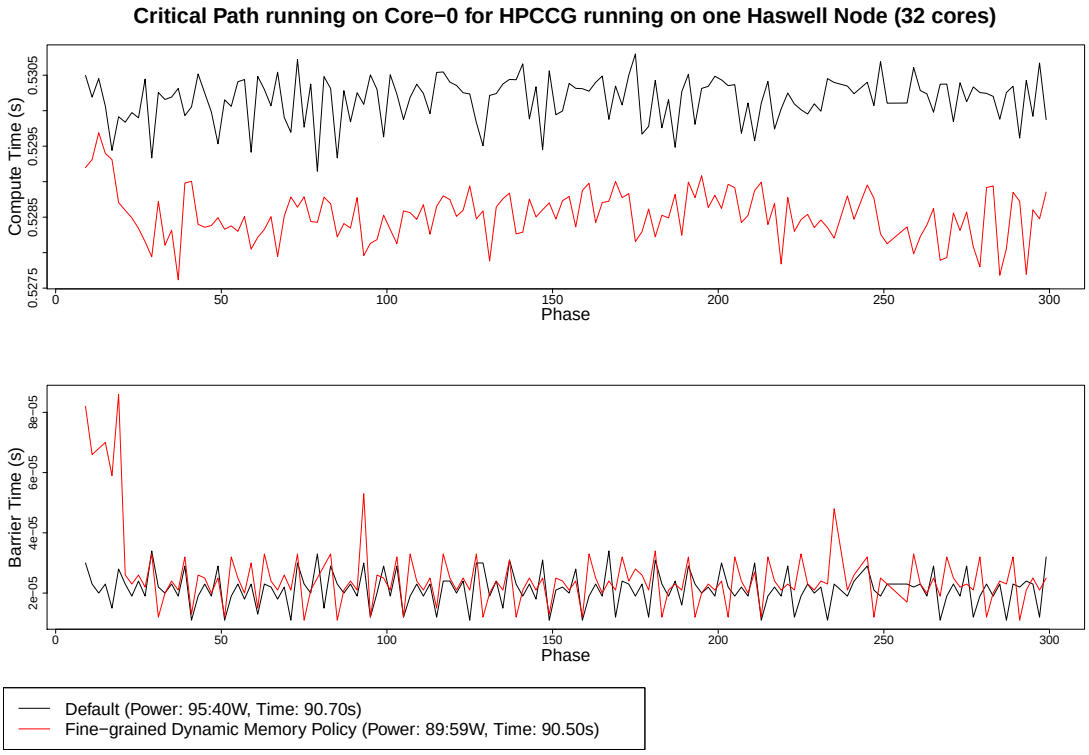


Figure 5.6: Compute and barrier times for HPCCG using fine-grained dynamic policy (FDP) on one Haswell node (32 cores). Power consumption and total execution time for default and FDP are also shown.

5.6.1 Understanding the effects of FDP on application performance

To better understand the effects of FDP on application performance, especially where the performance is improved, we looked at the critical path behavior for HPCCG (application with largest energy savings) on a single node. The experiment was repeated multiple times to ensure reproducibility of the compute and barrier times on a particular node and the results from one such node is shown in Figure 5.6. It can be observed that HPCCG runs slightly faster with FDP, in addition to reducing power by about 6.9% for an overall energy savings of 6.3%. This shows that FDP is also effective on single node runs.

The critical path for HPCCG on Core-0 (phases where Core-0 has the longest compute time) of a single Haswell node is shown. Core-0 was chosen as it was on the critical path for 142 out of the 302 HPCCG phases. Every phase runs 1 to 2% faster with almost no impact on the closing barrier time. The timescales however are very small as the improvement in execution time with the policy is only 0.2s.

Multiple metrics like the number of sets and resets by the policy, instructions retired and frequency were further analyzed to explain improvement in performance or lack of performance degradation. There is about a 10% reduction in the overall instructions retired both at the user and OS-level across the 32 cores. This is

mostly due to elimination of busy-waiting instructions for cores not on the critical path with their frequency reduced. We found that the length of the phases as seen in the figure are not large enough to support reliable gathering of frequency related information to aid our understanding. Further, our attempt to use Intel's Precision Event Based Sampling (PEBS) (Intel 2015a) was not useful as the sampling information is not accurate with changing frequency as the clock timing is affected. Thus, we infer that the cases where the policies have shown to improve performance may be attributed to lowering of frequency decreasing resource contention and eliminating busy-wait instructions or adding thermal headroom for the critical core.

5.7 Conclusion

Despite faster memory transfer rates and simultaneous transfers through multiple memory controllers, the memory bottleneck remains a frequent limitation in HPC. As *Big Data* and other memory-intensive applications become more main stream in HPC, the bottleneck is likely to stay. There is likely to remain an opportunity to save energy in memory-bound applications. The TORo_core metric presented identifies memory behavior exhibited by the applications conducive to constructing runtime energy saving policies. This along with the policies in the previous chapter further supports the thesis claim that local core measurements and controls are effective in improving energy efficiency without performance degradation.

Further understanding of advantages and disadvantages of the proposed metric and policy is needed. This can involve finding memory-constrained applications for assessment, and identifying when and where each of the two policies is better suited. Other metrics like instructions per cycle may also aid the policy decisions to further improve energy efficiency.

CHAPTER 6: Conclusions

In recent times, the fundamental drive in supercomputing to increase peak performance by adding an increasing number of power hungry components has waned, with the shift in focus to energy efficiency to mitigate large operating costs and carbon footprints in future supercomputing centers. This dissertation explored and evaluated methods to improve energy efficiency of HPC applications at runtime by dynamically adapting power consumption to the workload characteristics utilizing underlying machine power controls. Toward this end, runtime optimization techniques that perform measurements and control local to a processor core are presented that improve energy efficiency in several varieties of HPC applications. These techniques address many of the challenges faced by prior energy efficiency techniques in HPC. The techniques do not involve static analysis or require code changes in the application. The use of core-specific power control overcomes the global slowdown effect experienced by techniques that use chip-wide power controls. The techniques have been validated on several HPC mini and full applications. Several experiments have been performed to study the effect of DDCM techniques on power-limited systems. Our results prove that *measurements and controls local to each core in a multi-core system can on average reduce power consumed by large supercomputing applications at runtime while having little or no adverse impact on execution times.*

Research over the last 10-15 years has focused on software techniques to save energy. The techniques save energy but increase execution time. HPC has to date, ignored the results because very high machine depreciation costs make absolute performance critical. This dissertation introduced techniques that on average reduce power consumed while having little or no adverse impact on execution times. With exascale performance goals, changes in system design like over-provisioning will make software energy saving techniques particularly relevant to HPC. Many exascale applications will have heterogeneous processor load, and with power-limits most exascale systems will have heterogeneous performance. The techniques presented in this dissertation save energy by dynamically setting core-specific clock rates. Since the hardware uses less power for cores doing less work, in the presence of favorable conditions (Chapter 4) or when power-limited (Chapter 3), it is free to allocate additional power to cores running the critical threads. This improves overall performance, suggesting that saving energy will also be a performance optimization.

6.1 Summary of Results

An adaptive runtime DDCM policy is presented in Chapter 3 to reduce power in unbalanced MPI applications. In contrast with prior chip-wide DVFS based studies, this is a core-specific approach that slows down only non-critical cores to prevent performance degradation. As the policy computation is all local, it is seen to scale well from a single node to 16 nodes both on conventional and power-limited systems. This makes the demonstrated policy attractive for large HPC systems. Optimizing for energy is often also seen to be a performance optimization. In the presence of a power limit, the policy is seen to reduce energy along with improving performance. DDCM is thus shown to be an alternative to save energy in HPC, and improve performance in power-capped environments.

Secondly, an Adaptive Core-Specific Runtime for energy efficiency allowing processors with core-specific power control to reduce power with little performance impact by dynamically adapting core frequencies to workload characteristics is introduced in Chapter 4. This work extends the DDCM policy to DVFS and also introduces a method to combine the benefit of larger power reduction with DVFS owing to reduction in both voltage and frequency, and the ability of DDCM to lower the frequency beyond the operating range of DVFS. This framework is transparent to the application in the form of a MPI library and allows use of multiple power controls (DDCM, per-core DVFS or both). A large improvement in energy efficiency is obtained with a full application through a combination of speedup and power reduction. The average improvement in performance seen is a direct result of the reduction in run-to-run variation and running at turbo frequencies. This further makes a case for energy optimization as a performance optimization.

Lastly, metrics to detect applications that are constrained by memory are identified in Chapter 5 to be used in adaptive runtime policies to reduce energy. Our experimental memory study on modern CPU architectures, Intel Sandy Bridge and Haswell, identifies opportunities to reduce CPU frequency. The presented metric detects bandwidth saturation and increased latency in the memory system, and is used in a dynamic policy to modulate per-core power controls. The policy is evaluated when applied at coarse and fine-grained levels on MPI mini-applications. The policies do not adversely affect performance whether compute or memory-bound, but lower energy if memory-bound. In a few cases, using the policy makes the execution faster, which may be attributed to lowering of frequency decreasing resource contention or adding thermal headroom for non-throttled cores.

6.2 Limitations

In this section, we discuss the limitations of the showcased work. The general limitation for all of these techniques and the framework is that they need to have access to controls and counters at protection ring-0 of the operating system (OS) requiring root privileges. This can be a challenge on production machines as operating at root level can expose system vulnerabilities. In recent times however, the machine vendors and other vested entities have shown interest in coming up with interfaces to overcome this limitation (e.g. msr_safe). As these interfaces become more mainstream, policies and frameworks like ours may see more wide-spread adoption. Also, another minor side-effect of using our techniques is that applications need to link against our MPI library.

Our policies that target computational workload imbalance assume temporal patterns in the application. Though this is largely true, there can be departures from this. We do introduce rudimentary approaches to detect phase change and performance deterioration, however being able to ensure an upper-bound on performance degradation in the future will be ideal. As certain supercomputing centers also need to operate within an energy budget, this will also ensure that an energy budget is never violated due to high performance degradation.

The policies that target memory-constrained applications only work reliably for synchronous calls. The asynchronous calls are merely bypassed by resetting back the frequency. It will be beneficial however to see if the presented metric is still useful once latency hiding comes into play. One way to achieve this could be to use additional metrics to aid the policy decisions to make it effective for applications with majority asynchronous calls.

6.3 Future Work

There are many avenues of future work. First and foremost, we want to develop a more improved and encompassing ACR that can target applications with both unbalanced and memory-constrained phases by combining the two sets of policies. This is because many HPC applications exhibit both workload imbalance and memory-constrained phases as evinced by the results in Chapters 3, 4 and 5. This new framework will not be just combining of the two individual shim libraries, but will include a policy that would be first able to ascertain if a phase is compute or memory-bound, and if compute bound whether there is any workload

imbalance dynamically at runtime. Next, it would be able to use the appropriate approach with the correct power control. There are certain challenges to achieving this in a real world setting.

- New boundary conditions to couple the solutions in the two areas with an upper-bound on the performance degradation are required.
- A thorough understanding of the different power controls to ascertain what control is most appropriate for a particular scenario.
- A comprehensive evaluation and validation of the policy is required on large system using several full HPC application for it to be adoption ready on production systems.

For policies targeting unbalanced works loads a more detailed understanding of when to use each of the power controls will be very useful. It is observed that DDCM and combined policies work well for extreme workload imbalances. A scale to classify the extent of workload imbalance will be highly useful in guiding a scheme that can choose a power control dynamically at runtime. In our evaluation of the above policies and the framework we use several HPC applications with unbalanced workloads to understand the potential gain realistically achievable. It would be interesting to compare the improvements with our policies to using dynamic load balancers. This would enable deeper understanding of lower resource contention and turbo boost due to increased thermal headroom on performance of an HPC application.

In the case of our policy for memory-constrained applications, we would want to compare it with Dynamic Concurrency Throttling (DCT) that involves changing the number of processes on which an application runs to lower energy. Traditionally, memory-bound applications have been shown to scale well on lower number of cores running at full speeds when compared to compute-bound applications (Curtis-Maury et al. 2006, 2008, Li et al. 2010, Porterfield et al. 2013b). This comparison will help us answer if running a memory-bound application on more cores at lower speeds has more merit than the traditional approach. Further, an analysis to see the improvement with our policy when compared to running all cores for a memory bound application at minimum frequency will help us determine the efficiency of our approach.

APPENDIX A: RENCI WORKLOAD IMBALANCE MICRO-BENCHMARKS CODES

```
/* Repeating Unequal */

#include<stdio.h>
#include "mpi.h"
#include<stdint.h>

#define SCALE 100000000
#define REPEAT 10
#define CORES 16

int main(int argc, char** argv)
{
  MPI_Init(NULL, NULL);

  int i,j,rank;
  int64_t result;
  int chunk;

  coresPerNode = CORES;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  chunk = rank % coresPerNode;

  for(i = 0; i < REPEAT; i++)
  {
    result = 0;
    for(j = 0; j < chunk * SCALE; j++)
    {
      result += j;
    }
    MPI_Barrier(MPI_COMM_WORLD);
  }

  MPI_Finalize();

  return 0;
}
```

Listing A.1: Repeating Unequal benchmark in C/MPI

```
/* Equally Shifting Load */

#include<stdio.h>
#include "mpi.h"
#include<stdint.h> //int64_t

#define SCALE 100000000
#define REPEAT 10
#define CORES 16

int main(int argc, char** argv)
{
  MPI_Init(NULL, NULL);

  int i,j,rank;
  int64_t result;
  int chunk;

  coresPerNode = CORES;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  for(i = 0; i < REPEAT; i++)
  {
    chunk = ((rank % coresPerNode) + i) % coresPerNode;
    for(j = 0; j < chunk * SCALE; j++)
    {
        result += j;
    }
    MPI_Barrier(MPI_COMM_WORLD);
  }

  MPI_Finalize();

  return 0;
}
```

Listing A.2: Equally Shifting Load benchmark in C/MPI

```c
/* Randomly Shifting Load */

#include<stdio.h>
#include "mpi.h"
#include<stdint.h> //int64_t
#include<stdlib.h> //rand

#define SCALE 100000000
#define REPEAT 10
#define CORES 16

int main(int argc, char** argv)
{
  MPI_Init(NULL, NULL);

  int i,j,rank;
  int64_t result;
  int chunk;
  int randomValue;

  coresPerNode = CORES;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  srand(rank % coresPerNode);

  for(i = 0; i < REPEAT; i++)
  {
    randomValue = rand()% 3 - 1;
    chunk = (chunk + randomValue) % coresPerNode;
    if(chunk < 0)
    {
      chunk = 0;
    }
    for(j = 0; j < chunk * SCALE; j++)
    {
        result += j;
    }

    MPI_Barrier(MPI_COMM_WORLD);
  }

  MPI_Finalize();

  return 0;
}
```

Listing A.3: Randomly Shifting Load benchmark in C/MPI

BIBLIOGRAPHY

(2008). Perfomance counters for linux. http://lwn.net/Articles/310176/.

(2012). Bios and kernel developers guide (bkdg) for amd family 15h models 00h-0fh processors. Technical report, AMD.

(NERSC-8/Trinity Benchmarks). http://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/.

Atkinson, J., Wamsley, T., Westerink, J., Cialone, M., Dietrich, C., Dresback, K., Kolar, R., Resio, D., Bender, C., Blanton, B., et al. (2008). Hurricane storm surge and wave modeling in southern Louisiana: A brief overview. In Spaulding, M., editor, *Estuarine and Coastal Modeling X*. ASCE.

Bader, D., Berry, J., Kahan, S., Murphy, R., and Riedy, E. (2010). The graph 500 list. graph500. org.

Bhalachandra, S., Porterfield, A., Olivier, S. L., and Prins, J. F. (2017a). An adaptive core-specific runtime for energy efficiency. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*, pages 947–956. IEEE.

Bhalachandra, S., Porterfield, A., Olivier, S. L., Prins, J. F., and Fowler, R. J. (2017b). Improving energy efficiency in memory-constrained applications using core-specific power control. In *Proceedings of the 5th International Workshop on Energy Efficient Supercomputing*, E2SC'17, pages 6:1–6:8, New York, NY, USA. ACM.

Bhalachandra, S., Porterfield, A., and Prins, J. F. (2015). Using dynamic duty cycle modulation to improve energy efficiency in high performance computing. In *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, pages 911–918. IEEE.

Blanton, B. (2008). North Carolina Coastal Flood Analysis System: Computational System. Technical Report TR-08-04, Renaissance Computing Institute, The University of North Carolina at Chapel Hill.

Blanton, B., McGee, J., Fleming, J., Kaiser, C., Kaiser, H., Lander, H., Luettich, R., Dresback, K., and Kolar, R. (2012). Urgent computing of storm surge for North Carolina's coast. *Procedia Computer Science*, 9(0):1677 – 1686. Proceedings of the International Conference on Computational Science, ICCS 2012.

Blanton, B., Seim, H., Luettich, R., Lynch, D., Werner, F., Smith, K., Voulgaris, G., Bingham, F., and Way, F. (2004). Barotropic tides in the South Atlantic Bight. *J. Geophys. Res.*, 109(C12024).

Brooks, D., Tiwari, V., and Martonosi, M. (2000). *Wattch: A framework for architectural-level power analysis and optimizations*, volume 28. ACM.

Bulatov, V., Cai, W., Fier, J., Hiratani, M., Hommes, G., Pierce, T., Tang, M., Rhee, M., Yates, K., and Arsenlis, T. (2004). Scalable line dynamics in paradis. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 19. IEEE Computer Society.

Butts, J. A. and Sohi, G. S. (2000). A static power model for architects. In *Microarchitecture, 2000. MICRO-33. Proceedings. 33rd Annual IEEE/ACM International Symposium on*, pages 191–201. IEEE.

Choi, K., Soma, R., and Pedram, M. (2004). Dynamic voltage and frequency scaling based on workload decomposition. In *Proc. of the 2004 Intl. Symposium on Low Power Electronics and Design*.

Curtis-Maury, M., Blagojevic, F., Antonopoulos, C. D., and Nikolopoulos, D. S. (2008). Prediction-based power-performance adaptation of multithreaded scientific codes. *Parallel and Distributed Systems, IEEE Transactions on*, 19(10):1396–1410.

Curtis-Maury, M., Dzierwa, J., Antonopoulos, C. D., and Nikolopoulos, D. S. (2006). Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 157–166. ACM.

David, H., Gorbatov, E., Hanebutte, U. R., Khanna, R., and Le, C. (2010). Rapl: memory power estimation and capping. In *Low-Power Electronics and Design (ISLPED), 2010 ACM/IEEE International Symposium on*, pages 189–194. IEEE.

Dennard, R. H., Gaensslen, F. H., Rideout, V. L., Bassous, E., and LeBlanc, A. R. (1974). Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268.

Dietrich, J. C., Bunya, S., Westerink, J. J., Ebersole, B. A., Smith, J. M., Atkinson, J. H., Jensen, R., Resio, D. T., Luettich, R. A., Dawson, C., Cardone, V. J., Cox, A. T., Powell, M. D., Westerink, H. J., and Roberts, H. J. (2010). A high-resolution coupled riverine flow, tide, wind, wind wave, and storm surge model for southern louisiana and mississippi. Part II: Synoptic description and analysis of hurricanes Katrina and Rita. *Mon. Wea. Rev.*, 138(2):378–404.

Dighe, S., Vangal, S. R., Aseron, P., Kumar, S., Jacob, T., Bowman, K. A., Howard, J., Tschanz, J., Erraguntla, V., Borkar, N., et al. (2011). Within-die variation-aware dynamic-voltage-frequency-scaling with optimal core allocation and thread hopping for the 80-core teraflops processor. *Solid-State Circuits, IEEE Journal of*, 46(1):184–193.

Ebersole, B., Westerink, J., Bunya, S., Dietrich, J., and Cialone, M. (2010). Development of storm surge which led to flooding in St. Bernard Polder during Hurricane Katrina. *Ocean Eng.*, 37(1, Sp. Iss. SI):91–103.

Edwards, R. G. and Joó, B. (2005). The chroma software system for lattice QCD. *Nucl. Phys B1*, 40 (Proc. Suppl.). arXiv:hep-lat/0409003.

Esmaeilzadeh, H., Blem, E., St Amant, R., Sankaralingam, K., and Burger, D. (2011). Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365–376. IEEE.

Etinski, M., Corbalan, J., Labarta, J., and Valero, M. (2010a). Optimizing job performance under a given power constraint in hpc centers. In *Green Computing Conference, 2010 International*, pages 257–267. IEEE.

Etinski, M., Corbalan, J., Labarta, J., and Valero, M. (2010b). Utilization driven power-aware parallel job scheduling. *Computer Science-Research and Development*, 25(3-4):207–216.

Etinski, M., Corbalan, J., Labarta, J., and Valero, M. (2012). Parallel job scheduling for power constrained hpc systems. *Parallel Computing*, 38(12):615–630.

Eyerman, S. and Eeckhout, L. (2011). Fine-grained dvfs using on-chip regulators. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(1):1.

Fan, X., Weber, W.-D., and Barroso, L. A. (2007). Power provisioning for a warehouse-sized computer. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 13–23. ACM.

Feng, X., Ge, R., and Cameron, K. W. (2005). Power and energy profiling of scientific applications on distributed systems. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 34–34. IEEE.

Fleming, J., Fulcher, C., Luettich, R., Estrade, B., Allen, G., and Winer, H. (2008). A real time storm surge forecasting system using ADCIRC. In Spaulding, M., editor, *Estuarine and Coastal Modeling X*. ASCE.

Fluhr, E. J., Friedrich, J., Dreps, D., Zyuban, V., Still, G., Gonzalez, C., Hall, A., Hogenmiller, D., Malgioglio, F., Nett, R., et al. (2014). 5.1 power8 tm: A 12-core server-class processor in 22nm soi with 7.6 tb/s off-chip bandwidth. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*, pages 96–97. IEEE.

Freeh, V. W. and Lowenthal, D. K. (2005). Using multiple energy gears in MPI programs on a power-scalable cluster. In *PPoPP 2005: Proc. of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.

Fu, H., Liao, J., Yang, J., Wang, L., Song, Z., Huang, X., Yang, C., Xue, W., Liu, F., Qiao, F., et al. (2016). The Sunway TaihuLight supercomputer: System and Applications. *Science China Information Sciences*, 59(7):072001.

Ge, R., Feng, X., and Cameron, K. W. (2005). Performance-constrained Distributed DVS Scheduling for Scientific Applications on Power-aware Clusters. In *SC05: Proc. of the 2005 ACM/IEEE Conference on High Performance Networking and Computing*. IEEE Computer Society.

Ge, R., Feng, X., Feng, W., and Cameron, K. (2007). CPU miser: A performance-directed, run-time system for power-aware clusters. In *ICPP 2007: 36th Intl. Conference on Parallel Processing*. IEEE.

Gholkar, N., Mueller, F., and Rountree, B. (2016). Power tuning hpc jobs on power-constrained systems. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, pages 179–191. ACM.

Goel, B. and McKee, S. A. (2016). A methodology for modeling dynamic and static power consumption for multicore processors. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pages 273–282. IEEE.

Gropp, W., Lusk, E., Doss, N., and Skjellum, A. (1996). A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828.

Gustafson, J. L. (1988). Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533.

Hackenberg, D., Schöne, R., Ilsche, T., Molka, D., Schuchart, J., and Geyer, R. (2015). An energy efficiency feature survey of the intel haswell processor.

Hammarlund, P., Martinez, A. J., Bajwa, A. A., Hill, D. L., Hallnor, E., Jiang, H., Dixon, M., Derr, M., Hunsaker, M., Kumar, R., et al. (2014). Haswell: The fourth-generation intel core processor. *IEEE Micro*, 34(2):6–20.

Hench, J. and Luettich, R. (2003). Transient tidal circulation and momentum balances at a shallow inlet. *J. Phys. Ocean.*, 33(4):913–932.

Heroux, M. and Barrett, R. (2012). Mantevo project homepage.

Hoefler, T. and Moor, D. (2014). Energy, memory, and runtime tradeoffs for implementing collective communication operations. *Journal of Supercomputing Frontiers and Innovations*, 1(2):58–75.

Hsu, C. and Feng, W. (2005). A power-aware run-time system for high-performance computing. In *SC05: Proc. of the 2005 ACM/IEEE Conference on High Performance Networking and Computing*. IEEE Computer Society.

Huang, S. and Feng, W. (2009). Energy-efficient cluster computing via accurate workload characterization. In *CCGrid 2009: Proc. of the 9th IEEE/ACM Intl. Symposium on Cluster Computing and the Grid*. IEEE Computer Society.

Intel (2015a). Intel 64 and ia-32 architectures software developers manual volume 3 (3a, 3b & 3c): System programming guide.

Intel (2015b). Intel xeon processor e5 and e7 v3 family uncore performance monitoring reference manual.

Kamil, S., Shalf, J., and Strohmaier, E. (2008). Power efficiency in high performance computing. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8. IEEE.

Kandalla, K., Mancini, E. P., Sur, S., and Panda, D. K. (2010). Designing power-aware collective communication algorithms for infiniband clusters. In *Parallel Processing (ICPP), 2010 39th International Conference on*, pages 218–227. IEEE.

Kang, J. and Ranka, S. (2010). Dynamic slack allocation algorithms for energy minimization on parallel machines. *Journal of Parallel and Distributed Computing*, 70(5).

Kappiah, N., Freeh, V. W., and Lowenthal, D. K. (2005). Just in time dynamic voltage scaling: Exploiting inter-node slack to save energy in mpi programs. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 33. IEEE Computer Society.

Kimura, H., Sato, M., Hotta, Y., Boku, T., and Takahashi, D. (2006). Emprical Study on Reducing Energy of Parallel Programs Using Slack Reclamation by DVFS in a Power-scalable High Performance Cluster. In *CLUSTER 2006: Proc. of the 2006 IEEE Intl. Conference on Cluster Computing*. IEEE.

Kogge, P., Bergman, K., Borkar, S., Campbell, D., Carson, W., Dally, W., Denneau, M., Franzon, P., Harrod, W., Hill, K., et al. (2008). Exascale computing study: Technology challenges in achieving exascale systems.

Kurd, N., Chowdhury, M., Burton, E., Thomas, T. P., Mozak, C., Boswell, B., Mosalikanti, P., Neidengard, M., Deval, A., Khanna, A., et al. (2015). Haswell: A family of ia 22 nm processors. *IEEE Journal of Solid-State Circuits*, 50(1):49–58.

Leverich, J., Monchiero, M., Talwar, V., Ranganathan, P., and Kozyrakis, C. (2009). Power management of datacenter workloads using per-core power gating. *Computer Architecture Letters*, 8(2):48–51.

Li, D., De Supinski, B. R., Schulz, M., Cameron, K., and Nikolopoulos, D. S. (2010). Hybrid mpi/openmp power-aware computing.

Lin, N. a. K. E., Smith, J., and Vanmarcke, E. (2010). Risk assessment of hurricane storm surge for New York City. *J. Geophys. Res.*, 115(D18121).

Lively, C. W., Wu, X., Taylor, V. E., Moore, S., Chang, H.-C., Su, C.-Y., and Cameron, K. W. (2012). Power-aware predictive models of hybrid (MPI/OpenMP) scientific applications on multicore systems. *Computer Science - R&D*, 27(4).

Livingston, K., Triquenaux, N., Fighiera, T., Beyler, J. C., and Jalby, W. (2014). Computer using too much power? give it a rest (runtime energy saving technology). *Computer Science-Research and Development*, 29(2):123–130.

Luettich, R., Carr, S., Reynold-Fleming, J., Fulcher, C., and McNinch, J. (2002). Semi-diurnal seiching in a shallow, micro-tidal lagoonal estuary. *Cont. Shelf Res.*, 22:1669–1681.

Luettich, R., Westerink, J., and Scheffner, N. (1992). ADCIRC: an advanced three-dimensional circulation model for shelves coasts and estuaries, Report 1: Theory and methodology of ADCIRC-2DDI and ADCIRC-3DL. Dredging Research Program Technical Report DRP-92-6, USACE/ERDC, Waterways Experiment Station, Vicksburg, MS.

Mämmelä, O., Majanen, M., Basmadjian, R., De Meer, H., Giesler, A., and Homberg, W. (2012). Energy-aware job scheduler for high-performance computing. *Computer Science-Research and Development*, 27(4):265–275.

Mandal, A., Fowler, R., and Porterfield, A. (2010). Modeling memory concurrency for multi-socket multi-core systems. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 66–75. IEEE.

Marathe, A., Bailey, P. E., Lowenthal, D. K., Rountree, B., Schulz, M., and de Supinski, B. R. (2015). A run-time system for power-constrained hpc applications. In *International Conference on High Performance Computing*, pages 394–408. Springer.

Michalakes, J., Dudhia, J., Gill, D., Henderson, T., Klemp, J., Skamarock, W., and Wang, W. (2004). The weather research and forecast model: software architecture and performance. In *Proceedings of the 11th ECMWF Workshop on the Use of High Performance Computing In Meteorology*, volume 25, page 29. Reading, UK.

Mucci, P. J., Browne, S., Deane, C., and Ho, G. (1999). Papi: A portable interface to hardware performance counters. In *Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10.

Niedoroda, A., Resio, D., Toro, G., Divoky, D., Das, H., and Reed, C. (2010). Analysis of the coastal Mississippi storm surge hazard. *Ocean Eng.*, 37(1):82–90.

Pase, D. (2008). The pchase benchmark page.

Patki, T., Lowenthal, D. K., Rountree, B., Schulz, M., and de Supinski, B. R. (2013). Exploring hardware overprovisioning in power-constrained, high performance computing. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 173–182. ACM.

Porterfield, A., Fowler, R., Bhalachandra, S., Rountree, B., Deb, D., and Lewis, R. (2015). Application runtime variability and power optimization for exascale computers. In *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers*, page 3. ACM.

Porterfield, A., Fowler, R., Bhalachandra, S., and Wang, W. (2013a). Openmp and mpi application energy measurement variation. In *Proceedings of the 1st International Workshop on Energy Efficient Supercomputing*, page 7. ACM.

Porterfield, A., Fowler, R., and Lim, M. Y. (2010). Rcrtool: Design document version 0.1. Technical report, Tech. Rep. RENCI Technical Report TR-10-01.

Porterfield, A. K., Olivier, S. L., Bhalachandra, S., and Prins, J. F. (2013b). Power measurement and concurrency throttling for energy reduction in openmp programs. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 884–891. IEEE.

Rahman, S. M. F., Guo, J., Bhat, A., Garcia, C., Sujon, M. H., Yi, Q., Liao, C., and Quinlan, D. (2012). Studying the impact of application-level optimizations on the power consumption of multi-core architectures. In *Proceedings of the 9th conference on Computing Frontiers*, CF '12, pages 123–132.

Rountree, B., Ahn, D. H., de Supinski, B. R., Lowenthal, D. K., and Schulz, M. (2012). Beyond dvfs: A first look at performance under a hardware-enforced power bound. In *26th IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum, IPDPS 2012, Shanghai, China, May 21-25, 2012*.

Rountree, B., Lowenthal, D. K., de Supinski, B. R., Schulz, M., Freeh, V. W., and Bletsch, T. K. (2009). Adagio: Making DVS practical for complex HPC applications. In *ICS '09: Proc. of the 23rd Intl. Conference on Supercomputing*.

Sarood, O., Langer, A., Gupta, A., and Kale, L. (2014). Maximizing throughput of overprovisioned hpc data centers under a strict power budget. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 807–818. IEEE Press.

Sarood, O., Langer, A., Kalé, L., Rountree, B., and De Supinski, B. (2013). Optimizing power allocation to cpu and memory subsystems in overprovisioned hpc systems. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pages 1–8. IEEE.

Shoga, K., Rountree, B., Schulz, M., and Shafer, J. (2014). Whitelisting MSRs with msr-safe. In *3rd Workshop on Exascale Systems Programming Tools, in conjunction with SC14*.

Snowdon, D. C., Sueur, E. L., Petters, S. M., and Heiser, G. (2009). Koala: a platform for OS-level power management. In *Proc. of the 2009 EuroSys Conference*.

Snyder, L. (1986). Type architectures, shared memory, and the corollary of modest potential. *Annual review of computer science*, 1(1):289–317.

Sundriyal, V. and Sosonkina, M. (2011). Per-call energy saving strategies in all-to-all communications. In *Recent Advances in the Message Passing Interface*, pages 188–197. Springer.

Tiwari, A., Laurenzano, M., Peraza, J., Carrington, L., and Snavely, A. (2012). Green queue: Customized large-scale clock frequency scaling. In *CGC '12: Proc. of the 2nd Intl. Conference on Cloud and Green Computing*.

Venkatesh, A., Vishnu, A., Hamidouche, K., Tallent, N., Panda, D. D., Kerbyson, D., and Hoisie, A. (2015). A case for application-oblivious energy-efficient mpi runtime. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 29. ACM.

Vishnu, A., Song, S., Marquez, A., Barker, K., Kerbyson, D., Cameron, K., and Balaji, P. (2010). Designing energy efficient communication runtime systems for data centric programming models. In *Proceedings of the 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, pages 229–236. IEEE Computer Society.

Wang, W. (2016). *Performance, power, and energy tuning using hardware and software techniques for modern parallel architectures*. University of Delaware.

Wang, W., Porterfield, A., Cavazos, J., and Bhalachandra, S. (2015). Using Per-Loop CPU Clock Modulation for Energy Efficiency in OpenMP Applications. In *Proceedings of the 2015 44th International Conference on Parallel Processing (ICPP)*, pages 629–638. IEEE Computer Society.

Weste, N. H. and Harris, D. (2015). *CMOS VLSI design: a circuits and systems perspective*. Pearson Education India.

Westerink, J., Luettich, R., Feyen, J., Atkinson, J., Dawson, C., Roberts, H., Powell, M., Dunion, J., Kubatko, E., and Pourtaheri, H. (2008). A basin- to channel-scale unstructured grid hurricane storm surge model applied to Southern Louisiana. *Mon. Weather Rev.*, 136:833–864.

Westerink, J., Luettich, R., and Muccino, J. (1994). Modeling tides in the western North Atlantic using unstructured graded grids. *Tellus*, 46a:125–152.

Youssef, A., Anis, M., and Elmasry, M. (2006). Dynamic standby prediction for leakage tolerant microprocessor functional units. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pages 371–384. IEEE.

Zijlema, M. (2010). Computation of wind-wave spectra in coastal waters with SWAN on unstructured grids. *Coastal Eng.*, 57(3):267–277.