PERFORMANCE, POWER, AND ENERGY TUNING USING HARDWARE AND SOFTWARE TECHNIQUES FOR MODERN PARALLEL ARCHITECTURES

by

Wei Wang

A dissertation submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science

Summer 2016

PERFORMANCE, POWER, AND ENERGY TUNING USING HARDWARE AND SOFTWARE TECHNIQUES FOR MODERN PARALLEL ARCHITECTURES

by

Wei Wang

Approved: _____

Kathleen F. McCoy, Ph.D.
Chair of the Department of Computer and Information Sciences

Approved: _____

Babatunde A. Ogunnaike, Ph.D.
Dean of the College of Engineering

Approved: _____

Ann L. Ardis, Ph.D.
Senior Vice Provost for Graduate & Professional Education

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

John Cavazos, Ph.D.
Professor in charge of dissertation

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Guang R. Gao, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

James Clause, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Allan Porterfield, Ph.D.
Member of dissertation committee

ACKNOWLEDGEMENTS

There are so many people that I am feeling thankful for. It would not have been possible for me to finish my graduate program without their help.

First and foremost, I would like to thank my advisor, Professor John Cavazos, for his guidance, encouragement, and support throughout my years pursuing the PhD degree. I am truly grateful to have such a considerate and caring advisor.

In the meanwhile, I would like to thank Professor Guang R. Gao, Professor James Clause, and Dr. Allan Porterfield for being on my committee and for their guidance and feedbacks on my research. I also want to specially thank Allan for being my mentor during three internships at RENCI. At RENCI, I learned a lot from Allan and built foundation for my dissertation.

Additionally, I would like to express my gratitude to Dr. Edgar Leon, who was my mentor while I was doing internship at LLNL, Professors Howie Huang and Matthew Kay for their guidance and collaboration on the first part of the dissertation, and Dr. Alexandra Jimborean for her help on understanding Decoupled Access-Execution model.

My graduate life would not be as enjoyable without my friends in and out of the lab. I would like to thank Eunjung Park, Lifan Xu, William Killian, Tristan Vanderbruggen, Robert Searles, Sameer Kulkarni, Scott Grauer-Gray, Sridutt Bhalachandra, Yuanfang Chen, Fan Yang, Yifan Peng, Xiaoran Wang, Xi Chen and Xin Cheng for their help in many aspects, from setting up experiments to substituting for lecturing, from baby sitting to job referral, etc.

I would like to thank my parents, my mother-in-law and father-in-law, and my two elder sisters for their constant love and support for me.

Last but not the least, I would like to thank my wife, Fan Zhang, for always standing by me, encouraging me, supporting me, and loving me, and my daughter, Evelyn Wang, for bringing me the joy of being a father. This dissertation is dedicated to them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

As the high-performance computing (HPC) community continues the push towards exascale computing, power consumption is becoming a major concern for designing, building, maintaining, and getting the most out of supercomputers. Energy efficiency has become one of the top ten exascale system research challenges. Meeting the goal of exascale performance with 20 Megawatts of power limit requires performance, power, and energy optimization techniques at all levels, from the hardware to the application. In the meanwhile, although advances in parallel architectures promise improved peak computational performance, the use of software tools to drive parallelism of the hardware still requires expertise that is not widely available. Domain scientists are faced with a challenge to efficiently port applications to new parallel architectures like Nvidia GPUs and Intel Many-Integrated Core (MIC) accelerators. The fact that an increasing number of supercomputers are going to contain accelerators poses more challenge to application developers, who will need to get their applications ready for the supercomputers.

In this dissertation, we began with the study on how to achieve both automatic parallelization using OpenACC and enhanced portability using OpenCL. We applied our parallelization techniques on GPUs as well as an Intel MIC-architecture accelerator to reduce the running time of 2D wave propagation simulations. The performance and programmability of CUDA, OpenCL, OpenACC, and OpenMP implementations of the wave propagation simulation are compared. Compared to CUDA and OpenCL, we believe that OpenACC is preferable for domain scientists because programmers can parallelize their code using simple directives, and therefore it speeds up the process of parallelizing applications. OpenACC is shown to be able to achieve comparable performance as CUDA and OpenCL on GPUs with much reduced coding effort. Our

OpenMP implementation outperforms OpenCL and OpenACC on the Intel MIC accelerator. Emerging software developments like the OpenACC facilitate exploiting application parallelism offered by evolving hardware architecture. Our method of using OpenACC, OpenCL, and OpenMP to achieve efficient and effective parallelization on different accelerators can be generally applied to benefit other domains.

For the energy tuning problem, we tackle the problem from using software techniques first. We integrated an energy measurement framework to an existing polyhedral transformation framework called PoCC. Loop transformations supported by PoCC have been shown to be effective in optimizing the performance of small kernels. However, there have been few studies on how these transformations affect the power and the energy. The energy measurement framework allows exploring the relationship between tuning for power/energy and tuning for performance. A high correlation of energy/performance in PoCC is observed but tuning for power is different from tuning for execution time. We constructed predictive models that achieved high prediction accuracy. In addition, we also demonstrate the potential of polyhedral transformations in optimizing the 2D cardiac wave propagation application for both performance and energy.

Then, we propose to minimize energy usage without impacting the performance of HPC applications from using hardware techniques. We developed energy optimization techniques that did not only reduce power, but also Energy-Delay Product (EDP) and in some cases even Energy-Delay-Squared Product (ED^2P). We took advantage of the low transition overhead of CPU clock modulation and applied it to fine-grained OpenMP parallel loops. The energy behaviour of OpenMP parallel regions is first characterized by the memory access density. By characterizing memory access density, the best clock modulation setting is determined for each region. Finally, different CPU clock settings are applied to the different loops within the same application. The resulting multi-frequency execution of OpenMP applications achieved better energy efficiency than any single frequency setting.

In the last chapter of this dissertation, we combined software and hardware

techniques to obtain better energy efficiency for HPC applications. In particular, on Intel Sandy Bridge architecture we applied concurrency throttling, i.e., reducing the number of threads needed by an OpenMP application, with CPU clock modulation and on IBM Power8 architecture we applied concurrency throttling with DVFS. In both cases we observed improved energy efficiency. Lastly, we combined polyhedral compilation techniques with CPU clock modulation and evaluated their interactions under a power-capped environment.

Chapter 1

INTRODUCTION

As the high-performance computing (HPC) community continues the push towards exascale computing, power consumption has become a major concern for designing, building, maintaining, and getting the most out of supercomputers and energy efficiency has become one of the top ten exascale system research challenges [93]. State-of-the-art technologies still have a big gap in achieving desired energy efficiency in terms of performance per Watt. Table 1.1 lists the performance (in TFLOPS), power (in KiloWatts), and energy-efficiency (in GFLOPS per Watt) of the three fastest supercomputers from the Top500 list [2] and the three most energy-efficient supercomputers from the Green500 list [1]. Also shown in the table is a goal system that attains exascale performance with the 20MW power limit set by the HPC community [9]. Meeting the goal of exascale performance and the goal of 20 Megawatts of power requires an energy efficiency of fifty GFLOPS/watt (billions of operations per second per watt). However the world's "greenest" Shoubu supercomputer from RIKEN has not yet surpassed seven GFLOPS/Watt. The immediate two runner-ups achieved just a little over six GFLOPS/Watt. The world's fastest supercomputer Sunway TailhuLight has reached 93 Petaflops but its energy efficiency is about six GFLOPS/Watt. The second fastest supercomputer Milkyway-2 has reached 33 Petaflops of sustained performance but is drawing the power at the rate of almost 20 Megawatts, resulting in an energy efficiency of only 1.901. Titan, the third fastest supercomputer is only about 10% more energy-efficient than Milkyway-2. Next-generation supercomputers step closer to the goal but continuous efforts are needed to further optimize the performance and power. Summit and Sierra, two supercomputers that are staged to replace Titan and Sequoia respectively, will reach more than 100 Petaflops (one tenth of 1 Exaflops) peak computation

Table 1.1: Performance, power, and energy efficiency of 6 top systems (as of June 2016) from the Top500 list and the Green500 list as well as a goal system that achieves exascale performance with 20MW of power.

System	*Performance (TFLOP/s)*	*Power (KW)*	*GFLOPS/W*
Exascale Goal System	1,000,000	20,000	50
Top500 NO.1 (Sunway-TaihuLight)	93,014.6	15,371	6.051
Top500 NO.2 (Milkyway-2)	33,862.7	17,808	1.901
Top500 NO.3 (Titan)	17,590.0	8,209	2.143
Green500 NO.1	1001.0	149.99	6.674
Green500 NO.2	290.5	46.89	6.195
Green500 NO.3	93,014.6	15,371	6.051

rate [36]. Summit will draw 10 MW of power, i.e. 10% more than Titan [67]. Addressing the power consumption issue requires developing energy efficiency techniques at all levels, from the hardware to the application. Optimizing HPC applications for energy has become as important as optimizing for performance. Controlling an application's energy use running on increasingly powerful compute nodes will continue to be an important concern.

Before tuning an application for energy efficiency, it is necessary to understand how energy consumption optimization is related to traditional performance (i.e., execution time) optimization. Knowledge of the relationship between performance and energy can guide the tuning effort on pre-exascale and exascale systems. For most scientific applications, nested loops consume a significant portion of the total running time. When tuning an application for better performance and energy usage, some combination of loop optimizations, including loop tiling, loop unrolling, and loop fusion, are usually performed on the program along with auto-parallelization. Determining which set of optimizations produces the best results is challenging. Polyhedral auto-tuning frameworks have shown promising results at simplifying that effort [70, 68, 69] for small computation kernels, such as the Polybench programs [77]. The foundation of our auto-tuning framework is a polyhedral compiler, capable of generating thousands of program variants with the same semantics as the original program. In order for programs to be auto-tuned using a polyhedral framework, the original program should contain Static

Control Parts (SCoPs). Polybench is a suite of programs containing SCoPs. We added energy measurement capability on top of an existing performance auto-tuning framework to study the relationship between optimization for performance and for energy. In this dissertation work, we use the Resource Centric Reflection daemon tool (RCRtool/RCRdaemon) developed at RENCI [74], to measure energy consumption at a fine granularity for any OpenMP program. Fine-grain measurements enable attribution of energy consumption to particular application regions and even to individual lines of codes. This allows an accurate study of the correlation between execution time and energy consumption of an application. If the correlation study reveals that there is no or little correlation between execution time and energy consumption of program variants, then the machine learning model for predicting the best performing variant will not be suitable for predicting the version that consumes the minimum amount of energy. On the other hand, if a strong correlation between execution time and energy consumption exists, then the auto-tuning framework for sifting out the best optimizations can be extended to select the program version that consumes the least amount of energy.

Although polyhedral compilers are powerful in generating various code versions for specially written small kernels like those found in the Polybench, their restrictive requirements (SCoPs only) usually limit their application in optimizing larger and realistic applications. One reason is that many applications do not contain SCoPs by default. It is not uncommon for scientific codes to include indirect memory accesses. However no indirect memory access is allowed in a SCoP therefore polyhedral compilers would not be capable of dealing with applications with such code characteristics. Fortunately there are realistic applications that are amenable to polyhedral compilers. Such applications perform complex simulations without using a lot of branching and indirect memory access. This dissertation will show the effectiveness of using polyhedral compilers to optimize a wave propagatioin simualation application for performance. This realistic application simulates the cardiac wave propagation in a 2D grid and is used for cardiac arrhythmia research. From our experience of polyhedrally optimizing

a realistic application, we hypothesize that a large number of kernels from realistic scientific applications can benefit from applying source-to-source transformations using polyhedral compilers.

Several current and future top supercomputers feature accelerators that represent the state-of-the-art parallel architecutre innovations. For example, the Titan supercomputer features hybrid parallel architecture consisting of multi-core AMD Opteron CPU and Nvidia Kepler K20x GPU. Two future supercomputers (Summit and Sierra) will both include IBM Power9 multi-core CPUs and Nvidia Volta GPUs. On Summit, more than 90% of floating point calculation power is expected to come from Volta GPUs. The Milkyway-2 supercomputer contains Intel Xeon Phi coprocessors, which are Many-Integrated-Core (MIC) architecture accelerators. Several future supercomputers like Aurora and Theta will include next-generation Xeon Phi processors [5]. It is worthwhile to prepare applications for getting good performance out of the GPUs or Xeon Phi accelerators found on supercomputers, using various appropriate tools. CUDA, OpenCL, and OpenACC are three major programming languages that target GPUs. New developments on the OpenMP standard include the "target" construct to support execution on GPUs. However, these new developments are still evolving, and they are not as mature as OpenACC. OpenMP code traditionally runs only on multi-core CPU architectures but the MIC architecture additionally supports OpenMP. CUDA and OpenCL are low-level GPU programming languages while OpenACC and OpenMP are high-level, directive-based language and language extension. Compared to CUDA and OpenCL, OpenACC and OpenMP are better for domain scientists because programmers can parallelize their code using simple directives, and therefore it speeds up the process of preparing applications for supercomputers. OpenCL and OpenACC are portable languages and extensions in that the code generated by these languages can run on both Nvidia GPUs and the Intel MIC architecure. OpenCL is known for its portability and an OpenACC compiler can generate OpenCL code that targets different platforms, including Intel Xeon Phi nodes. In this dissertation,

the aforementioned wave propagation simulation application is used to show how different programming languages and language extensions can help prepare applications for execution on modern parallel architectures, especially the newest HPC systems that contain accelerators. In addition, performance and programmability of CUDA, OpenCL, OpenACC, and OpenMP implementations of the cardiac wave propagation simulation are compared. OpenACC is shown to achieve comparable performance as CUDA and OpenCL on GPUs with much less code. Our OpenMP implementation outperforms OpenCL and OpenACC on an Intel Xeon Phi node.

In our research, we found that performance optimizations that improve execution time save energy as a by-product of the improved performance. This is referred to as a "race-to-halt" strategy. Other than this "race to halt" strategy that focuses on performance improvement, research has employed Dynamic Voltage and Frequency Scaling (DVFS) techniques to save energy. Ge et al. divided applications into fixed intervals and determined a suitable frequency for each interval by using performance counters [28]. Other work looked at applying DVFS during communication phases or non-critical execution paths of MPI applications to lower power consumption without affecting performance [96, 94, 95, 25, 99]. All of these approaches applied DVFS to the coarse-grain phases of the application. For OpenMP code, parallel regions as identified by the "parallel" construct naturally divide an application into fine-granularity phases, especially for code that involves multiple time-steps. None of the aforementioned DVFS approach applied energy control on a per-loop basis. This was likely due to the overhead of switching the frequency using DVFS prevented it from being used for small code regions.

Another method to reduce energy consumption of CPUs is CPU clock modulation, also known as Duty Cycle Modulation, CPU throttling, or CPU clock skipping, which squashes cycles in the CPU clock without changing the real frequency [96, 37, 110]. Using this technique allows individual cores to have the effective clock frequency reduced, without changing the voltage or the memory system's clock. The advantage of CPU clock modulation over DVFS is the much lower transition overhead [76], since

no voltage stabilization must occur. In addition, CPU clock modulation can be applied on top of DVFS to provide more fine-grained energy control knobs that are otherwise available with DVFS. When the CPU has significant pipeline stalls or idle instructions, clock modulation reduces the energy required during the stalls.

Like DVFS, applying clock modulation to every parallel region can slow down execution and lead to increased energy consumption. By characterizing OpenMP parallel loops as compute-bound or memory-bound first, the appropriate clock modulation setting can be applied to each appropriate loop. Applications can have multiple parallel loops with different memory access densities. These types of applications have loops that prefer distinct frequency settings per loop that results in the minimum energy or energy-delay product (EDP). Applications with a variety of different classes of regions have the potential of benefitting from fine-grained per-loop clock modulation to conserve energy without performance degradation.

For loops with high/medium memory access density, reducing the frequency eliminates some of the energy during stalled pipeline cycles. For loops with low memory access density, setting the frequency to the maximum can avoid the unnecessary delay of computation. Using the best frequency for different loops can have a lower energy/EDP than any single frequency execution setting. A multi-frequency setting profits from saving energy during CPU stalls (e.g. during cache misses) and maximizing computation otherwise.

Another technique we investigated was concurrency throttling [49, 76], which reduces the number of hardware threads being used, thus saving energy when parts of the system are saturated. It can be combined with clock modulation to further reduce energy consumption for some application phases on Intel architectures. As mentioned earlier, Summit and Sierra supercomputers both feature IBM Power9 architecture and Nvidia's Volta GPU architecture. We believe it worthwhile to test the combined software techniques (concurrency throttling) and hardware techniques (energy control) on the Power architecture. At the time this dissertation research was performed, the IBM Power9 was not in production, therefore this dissertation investigated its predecessor,

the Power8 architecture. Similar to the Power9, the Power8 also featured 8-way simultaneous multithreading (SMT-8). However, since the Power8 did not support the CPU clock modulation hardware feature, we combined per-core DVFS with concurrency throttling[1].

Future data centers and supercomputers will have to address the challenge of running applications with limited power envelope due to budget concerns. This has prompted new power management features in new parallel architectures. As an example, power capping is a built in concept in the overall architecture design of IBM Power8 architecture [24]. Optimizing application performance on power constrained HPC systems arises as a new challenge. Power capping usually sets stage for power shifting which redistributes power resources to different components within a compute node [48]. In addition to realize intelligent balancing of critical power resource, power capping can be applied for thermal control. In this dissertation, we study how applications perform under a set power limit on Intel SandyBridge architecture. Unlike IBM Power architecture which relies an On Chip Controller (OCC) to cap the power consumption implicitly according to thermal control algorithms, power capping on Intel platforms can be done explicitly via privileged instructions. In addition, other power management techniques like DVFS and CPU clock modulation can be applied on top of power capping. Combined effects of power capping and clock modulation techniques are investigated for several benchmarks in this work. We also combined polyhedral compilation with clock modulation technique to find opportunities where polyhedral optimized code could be further optimized for energy and energy efficiency (e.g. EDP).

The main contributions of this dissertation are as follows. First, we parallelized and optimized a realistic scientific application on several parallel architectures using different programming tools. The lessons learned can benefit a broad range of developers who aim to make their applications performant and energy efficient on supercomputers involving accelerators like the Nvidia GPUs and the Intel accelerators. Second,

[1] On Intel Sandy Bridge architecture, DVFS is not per-core but per-socket.

to tackle the energy optimization issue, we studied the energy and time correlation of programs in a polyhedral autotuning framework. Tuning for time with polyhedral transformations can be used as a proxy for tuning for energy. A predictive model for energy is also developed and verified to have good accuracy. Then, we studied the energy characteristics of application loops in the context of being able to control machine power usage. Although a whole application might prefer "racing to halt" to get the best energy efficiency, the individual loops can have different preferences based on their memory density. Several mini-applications are executed in a multi-frequency fashion (i.e. different loops are executed at different frequencies using CPU clock modulation) to achieve the best energy-delay trade-off. Third, we show that combining hardware techniques like CPU clock modulation and DVFS with software techniques like concurrency throttling and polyhedral compilation achieves better energy and EDP improvement. Our experiences using IBM Power8 architecture to evaluate its DVFS features are encouraging for the OpenPower community. In addition, our experiments with power capping, CPU clock modulation, and polyhedral compilation provide insights on how applications would perform on future power constrained systems.

The rest of the dissertation is organized as follows. Chapter 2 gives the background necessary to understand the research in this dissertaiton. Chapter 3 discusses the parallelization and optimization of 2D wave propagation simulation on modern computational accelerators, including GPUs and Xeon Phi. Chapter 4 uses the polyhedral framework to evaluate execution time and energy consumption correlation. A predictive model for energy that drives polyhedral optimizations applied to parallel application is also covered. Chapter 5 shows how we utilize the energy control hardware methods to optimize applications for energy. Chapter 6 combines software techniques and hardware techniques used in previous chapters to further optimize benchmarks for energy on both Intel and IBM multicore architectures. Evaluation of benchmark performance in power-capped environment is also performed in this chapter. Chapter 7 concludes the dissertation.

Chapter 2

BACKGROUND AND RELATED WORK

This chapter introduces the tools and techniques used in this dissertation such as loop transformation tools, energy measurement tools, and energy control methods. State-of-the-art studies that used these tools are discussed.

2.1 Source to Source Compilers for Loop Transformations

The Polyhedral Compiler Collection (PoCC)[79], a source-to-source polyhedral compiler, was frequently used in this dissertation. This source-to-source compiler was used in Chapter 4 and Chapter 6 to generate program variants with different optimizations. It was also used in Chapter 4 to optimize application performance. Other compilers that can perform polyhedral source-to-source transformations include CHiLL and Orio.

2.1.1 PoCC and PolyOpt

PoCC requires that programs contain Static Control Parts (SCoP)[8, 23, 29] so that polyhedral transformations can be applied. A SCoP consists of a set of consecutive statements, which usually form a loop nest. The loop bounds, if statement conditionals, and array accesses in the loop nest should all be affine functions of the loop iterators and global parameters. For example, a valid affine expression for a loop bound in a SCoP with two loops iterators i,j and two parameters N,P will be of the form $a \cdot i + b \cdot j + c \cdot N + d \cdot P + e$, where a,b,c,d,e are arbitrary (possibly 0) integer numbers. The following are examples that break the SCoP property:

- Non-affine *for* initialization or test condition, e.g., $for(j = 0; j < i * i; ++i)$.

- if conditionals involving loop iterators that form a non-affine expression, e.g., if (i * j == 0).
- if conditionals involving variables that are not a loop iterator or a parameter, e.g., if (A[i][j] == 0).
- Non-affine array access, e.g., A[j % i] or A[B[i]].

In practice, code with excessive use of local variables, can make the polyhedral compiler take a very long time to transform programs.

Polybench is a collection of programs that contain SCoPs and thus can be optimized with a polyhedral compiler. Inside the compiler, each SCoP is represented by two matrices, which correspond to the loop nests' iteration domain as well as the statements' dependencies. Loop transformations on such programs are equivalent to manipulating such matrices. The transformed matrices can be converted back to semantically equivalent, but optimized code.

We used PolyOpt (a Polyhedral Optimizer for the ROSE compiler) [78] to automatically detect SCoPs in applications. PolyOpt is based on PoCC, but integrated into the ROSE compiler. Aside from its capability to extract SCoP regions in an automatic way, it fully supports polyhedral analysis and optimizations. PolyOpt supports loops fusion, loop tiling, thread-level parallelization and vectorization. PolyOpt has better support for side-effect free program features like math functions[8] and allows some function calls within a SCoP. PolyOpt, although a powerful polyhedral compiler, still may not be able to extract any SCoPs because of structural impediments of source codes. Changes to the program may be required to expose the SCoPs for PolyOpt, before loop transformations, parallelization, and vectorization can occur.

2.1.2 Other source-to-source compilers

Other source-to-source polyhedral compilers include CHiLL [15], Orio [61], PPCG [101], LooPo [31], PLUTO [12] and R-stream[87] etc. CHiLL takes in the original code and a transformation script as input and outputs a set of alternative implementations. The transformation script, also called transformation recipe, describes how to optimize the

code. For example, a CHiLL transformation script may include loop permutation, loop tiling, and loop unrolling etc. Orio has similar functionality in transforming programs. Instead of taking in a stand-alone transformation recipe, it takes in the original code with annotated text for loops of interest. PPCG is a polyhedral parallel code generator that translates sequential static control code for parallel execution on a modern GPU. LooPo and PLUTO both generate parallelized code for execution on modern shared-memory systems. PLUTO is included in POCC. R-stream is a commercial source-to-source compiler from Reservoir Labs. For our energy/performance correlation study, CHiLL, Orio, PPCG, and LooPo could potentially be used instead of PoCC. However, we chose PoCC with PLUTO because it has been extensively tested on our benchmarks of interest, including the Polybench.

2.2 Energy Measurement Tools

Traditional energy measurement tools require hardware instrumentation and provide coarse grained energy measurement. In contrast, software energy measurement is more fine-grained. The Intel Sandy Bridge and Haswell architectures allow users to track energy usage through the Running Average Power Limit (RAPL) interface [37]. Energy consumed by the chip can be tracked by a Model Specific Register (MSR), specifically the MSR_PKG_ENERGY_STATUS performance counter. This counter is frequently updated and counts the energy in 15.3 micro-Joule (i.e. $1/2^{16}$ Joule) units on Sandy Bridge architectures and 61.2 micro-Joule (i.e. $1/2^{14}$ Joule) units on Haswell architectures. Based on RAPL, different software tools were developed to monitor power usage of applications. In this dissertation work, we use RCRdaemon [75] to monitor application energy. Other tools that can measure application energy based on RAPL include LIKWID [100] and Intel Power Gadget[39]. The accuracy of these tools relies on the exactness of RAPL. Hähnel *et al.* reported the identical curve characteristics comparing RAPL with external measurement[32] while presenting their HAECER framework for short-term energy measurements using RAPL.

2.2.1 RCRdaemon

The RCRdaemon runs at supervisor level to access the hardware counters through the MSR performance counters. For each monitored counter, it writes the current value of the counter at least 1000 times a second into a shared-memory data structure. This "blackboard" structure provides a hierarchical view of the system where various current performance information is stored. The storage is only 8KB. The hardware MSR_PKG_ENERGY_STATUS counter is only 32 bits and can overflow in as little as a couple of minutes. The RCRdaemon detects the overflow and supplies a 64 bit value with the upper 32 bits being the number of overflows since RCRdaemon instantiation. The stored information is in shared memory and is available to any OpenMP applications through a simple API that delineates a code region for measurement with a start and end call. Each region is identified by its file name and line number. If a region is executed multiple times, the energy is summed across all executions. All energy information is available during application shutdown. The execution time increase from continuous monitoring of a 400-second OpenMP application is observed to be less than 8%.

2.2.2 LIKWID

LIKWID contains a set of command-line lightweight performance tools. It offers likwid-powermeter to access RAPL energy counters and query Turbo mode steps on Intel processors. It does not export an energy library for direct application invocation.

2.2.3 Intel Power Gadget

Like LIKWID, the Intel Power Gadget can also output current energy information by querying RAPL energy counters. RCRdaemon differs from both the Intel Power Gadget and LIKWID in that it manages energy information in a way that provides access to all applications This is supported by the daemon querying the counters frequently and calculating the energy as well as the APIs granting all applications access to the energy information.

2.3 Power Control Methods

Modern Intel CPU architectures provide three mechanisms to control power usage. They are Dynamic Voltage and Frequency Scaling (DVFS), Duty Cycle Modulation (DCM, also called Clock Modulation), and Power Capping. Similar to the APIs that get the energy consumption information, energy control API calls are developed and inserted into the source code to control the voltage and frequency for DVFS, the clock modulation for DCM, and the power cap for Power Capping.

2.3.1 DVFS

DVFS refers to changing the frequency along with the paired voltage to put machines into low-power states. For a given machine, only a limited set of frequency/voltage pairs are supported. Users can invoke DVFS by writing the desired frequency value to a file on the file system. The driver, i.e., acpi-cpufreq, is responsible for detecting this change of frequency and for making the transition. Note that even though the users only specify the change of frequency, voltage values will be automatically changed. The frequency/voltage transition latency as well as the available set of frequencies can be looked up from files on the file system when the DVFS driver is enabled. As an example, to set all CPU cores to 1.2GHz, a user can invoke the following command that writes the frequency value (1200000 KHz) to the files (scaling_setspeed) associated with each cpu core.

```
echo 1200000|tee /sys/devices/system/cpu/cpu*/cpufreq/scaling_setspeed
```

The above method of changing DVFS utilizes a shell script and is relatively coarse-grained. A faster method of enabling DVFS changes is to open the scaling_setspeed file and overwrite its contents.

In order to achieve dynamic change of voltage and frequency scaling, i.e. dynamically decide what frequency should be used, "cpufreq governors" are usually set. In Linux kernel, there are five CPUfreq governors: conservative, ondemand, userspace, powersave, and performance. Powersave and performance set the CPU statically to the lowest and the highest frequency, respectively. The userspace governor allows the

user to set the desired frequency. Ondemand and conservative dynamically change the frequency by sampling the CPU usage.

DVFS has a global effect on Sandy Bridge architecture, which means all the cores in the same CPU must run at an identical frequency. Newer architectures like Intel Haswell [46, 33, 47] and IBM Power8 [24] support separated voltage/frequency domains therefore DVFS can be applied to cores individually and cores can operate at different frequency levels.

2.3.2 Duty Cycle Modulation

The hardware implementation of clock modulation involves setting the stop-clock internal signal to regulate the CPU's normal "heartbeat". Intel Xeon Processors support software-controlled clock modulation. The stop-clock cycle is controlled through the IA32_CLOCK_MODULATION model specific register (MSR). By controlling the number of cycles to be skipped for every 16 clock cycles, various effective frequencies can be achieved.[1] For example, skipping eight cycles changes the effective frequency to be half the original. There are 16 available frequencies ranging from 6.25% to 100% of the maximum frequency, with a 6.25% interval. Once the "MSR" kernel module is loaded (via modprobe), users can write the desired value of IA32_CLOCK_MODULATION to the MSR device file (found under /dev/cpu/cpu*/msr). Root privilege is required to control the power level as *modification* of MSR device files are protected. The Linux kernel has been locally modified to support a clock modulation system call. Changing the frequency via CPU clock modulation is a light-weighted process, requiring only a MSR write. The software controlled clock modulation can be applied on a core by core basis. The core-specific control provided by clock modulation

[1] Our tested Intel Sandy Bridge architecture was enabled to support sixteen frequencies. In general, non-extended software controlled clock modulation supports eight frequencies, with an interval of 12.5%. Extension to software controlled clock modulation is supported only if CPUID.06H:EAX[Bit 5] = 1. Also the CPU must have ACPI feature to support both extended and non-extended software controlled clock modulation, i.e. CPUID.01H:EDX[Bit 22]=1.

allows for a much finer control over energy of an application. Modulating the CPU clock does not change the actual frequency or voltage and can be combined with DVFS. This permits an effective frequency well below the minimum DVFS frequency, providing more energy optimization options for applications. Table 2.1 shows the values to use to set the desired frequency.

Table 2.1: Values to Write into IA32_CLOCK_MODULATION MSR to Achieve Duty Cycle Modulation.

Duty Cycle Level	*Binary*	*Decimal*	*Hexadecimal*	*Effective Frequency*
1	10001B	17	11H	6.25%
2	10010B	18	12H	12.5%
3	10011B	19	13H	18.75%
4	10100B	20	14H	25%
5	10101B	21	15H	31.25%
6	10110B	22	16H	37.5%
7	10011B	23	17H	43.75%
8	11000B	24	18H	50%
9	11001B	25	19H	56.25%
10	11010B	26	1AH	63.5%
11	11011B	27	1BH	69.75%
12	11100B	28	1CH	75%
13	11101B	29	1DH	81.25%
14	11110B	30	1EH	87.5%
15	11111B	31	1FH	93.75%
16	00000B	0	00H	100%

In DVFS, the transition to lower frequencies takes much longer than clock modulation due to the overhead of switching between the supported Voltage/Frequency pairs using the on-chip voltage regulator. Energy savings are greater with DVFS (due to combined frequency and voltage reduction, the power dissipation of a CPU is proportional to the frequency and the voltage-squared) but the change of frequency overhead is much lower with clock modulation, requiring only a MSR write. By setting the clock modulation, no frequency and voltage transition must be involved, yet the effective frequency is reduced. This advantage makes it more suitable for fine-grained energy control of applications.

2.3.3 Power Capping

Power Capping refers to putting a limit on the average power usage of the package. It is done by writing the power limit value into yet another RAPL MSR – MSR_PKG_RAPL_POWER_LIMIT. The power control is per package, i.e. over time the power consumption of the whole package is capped by the power limit.

2.4 DVFS-Based Energy Saving Techniques

To optimize parallel applications for energy efficiency, the potential of DVFS has been extensively studied. One scenario of applying DVFS takes advantage of the work-load imbalance (slack) in MPI applications. Such systems include Jitter [41], Adagio [85], and Green Queue [99]. Kappiah et al., use a runtime system Jitter to predict the appropriate clock frequencies for upcoming iterations based on observations of previous iterations. In Adagio, Rountree et al., [85] present a runtime system that accurately predicts slack. Tiwari et al., [99], in Green Queue, combine inter-node DVFS approach and intra-node DVFS approach and scale the DVFS technique to 1024 cores. The CPU clock modulation technique discussed should be applicable to all this related work.

Another common use of DVFS is to divide applications into phases first and then apply appropriate DVFS setting to each phase. Freeh et al., [26] choose different frequency settings according to application profile to be applied to different MPI phases. The MPI application is executed using multiple energy "gears" to realize energy savings. Livingston et al. [51] present REST to dynamically detect the memory intensive and compute intensive phases of MPI applications. Machine frequency is changed via DVFS. Other work utilizes performance counters [50, 28] to build power/energy models and use these built models to predict the optimal frequency setting for an application. In this dissertation, a simple memory access related counter is used to categorize OpenMP loops. Different energy settings were applied in fine granularity using the clock modulation technique. Other DVFS use cases include combining concurrency throttling to optimize for energy on multi-core platforms [19]. In contrast,

concurrency throttling is combined with clock modulation to realize energy savings in this work. Eyerman et al., studied the potential of fine-grain DVFS in saving energy for memory-intensive applications [22]. In absence of the hardware that supports fine-grained DVFS, our work favors software clock modulation over DVFS.

2.5 Clock-Modulation-Based Energy Saving Techniques

While DVFS has been extensively studied, only a handful of work (including our previous work [104]) has looked at employing clock modulation techniques to save application energy. Sundriyal et al., [96] apply clock modulation in inter-node communication phases of MPI applications to achieve significant energy savings with low overhead. They also apply the same technique in point-to-point communications [94] and collective communications [95]. Cicotti et al., [17] present Efficient Speed (ES), a library and run-time that controls the speed of processor while minimizing the performance impact. They achieved 16% energy decrease with less than 5% performance loss for MPI applications. We present an equivalent infrastructure for power measurement and control targeting OpenMP applications that achieves 10% energy savings with less than 1% performance loss for many kernels. In other energy optimization work [110], clock modulation techniques have been used to realize resource utilization management on servers.

2.6 Concurrency Throttling

Concurrency throttling is a software mechanism to modulate the amount of concurrency for regulating application runtime performance on systems with multi-core processors. By reducing the number of threads used for executing the applications, the concurrency level is regulated. This technique was used with DVFS for predicting the best configuration setting, i.e., the frequency and concurrency level, on HPC systems [19]. It has also been applied to realize energy savings for OpenMP benchmarks, such as LULESH and BOTS [76]. In that work, Porterfield et al., developed a run time system that automatically adjusted the concurrency level based on hardware

performance counters, achieving energy reductions without performance lost. In this dissertation, we combine clock modulation with concurrency throttling to achieve even more energy/EDP improvements. Li et al., combined DVFS with dynamic concurrency throttling (DCT) to achieve improved EDP in hybrid (OpenMP/MPI) programming models [49]. In that work, they presented models and algorithms for energy efficient execution of hybrid MPI/OpenMP applications. After characterizing energy-saving opportunities in these hybrid applications, they applied DCT and DVFS to leverage these energy-saving opportunities without performance loss. In this dissertation, we focus on applying concurrency throttling techniques with power saving techniques for OpenMP applications on different architectures, including the Intel Sandy Bridge architecture (with CPU clock modulation) and the IBM Power8 architecture (with DVFS).

2.7 Existing Performance-Energy Correlation Studies

Our work is not the first to show that "race to halt" can be the most energy efficient [102]. Yuki *et al.,* [109] developed a high-level energy model of power consumption under Dynamic Voltage and Frequency Scaling (DVFS) and found it best to run as fast as possible to completion. They pointed out that the constant power of current machines were significant enough to render DVFS useless in saving energy. Before them, Cho and Melhem[16] identified that DVFS might not help if the fraction of total power unaffected by DVFS is large. We evaluated the energy effects of compiler optimizations, rather than DVFS, by measuring the energy consumptions of hundreds to thousands of program variants. In most cases where power saving techniques like CPU clock modulation, power capping, and DVFS are not applied, program variants trying to "race to halt" consumed the minimum amount of energy and that optimizing for execution can be used as a proxy to optimizing for energy. Rahman *et al.,*[81] studied the impact of application level optimizations from both the performance and power efficiency perspective of various applications. They found that optimizing for performance did not guarantee better power consumption. We observed tuning for performance or tuning for power were both equally effective. Garcia *et al.,*[27] studied the

energy consumptions of applications and proposed models characterizing application energy consumption footprints.

2.8 Autotuning Frameworks Applied to Execution Time

In solving the problem of picking the right optimization sequence to apply to computational kernels, Park et al. [70] built a predictive model that accurately finds the optimization sequence which results in the best performance. The model was initially built from extracting dynamic program features consisting of various hardware performance counters. The model was later improved by replacing the fixed-length dynamic performance counter features with variable length control-flow graphs [68]. The inclusion of the structural similarity between programs increased the accuracy of the prediction models. The prediction model construction mainly involves the following steps:

1. Extract control flow graphs from the original, unoptimized code version of the benchmarks.
2. Annotate the nodes of the control flow graphs with histograms of instruction counts in the code blocks.
3. Represent compiler transformations using fix-length feature vectors.
4. Calculate the graph similarities using the Shortest Path graph kernel (a graph comparison algorithm).
5. Calculate the similarities between the compiler transformation sequences (represented as feature vectors).
6. Combine the graph similarities and compiler transformation sequences to derive similarities of program variants.
7. Run Support Vector Machine (SVM) machine learning algorithms to construct a prediction model.

Given a new test program and its compiler optimization sequences, the model predicts the execution time for each optimization sequence. In this work, we extend the model to predict the energy savings of different compiler optimizations over the baseline program (without optimizations). We investigated energy savings prediction

models because we empirically determined that execution time prediction models are not adequate when used with power management techniques. When not considering power management techniques, such as CPU clock modulation, the compiler transformations that achieve the minimum execution time usually lead to the minimum energy consumption. However, once we consider varying the CPU frequency, the minimum energy may be achieved from optimized variants of the program that do not achieve a minimal execution time because the power consumption can be significantly reduced without affecting performance much. In contrast to the execution time prediction model which does not contain power information, the energy prediction model is constructed from application energy consumption, which reflects the impact on power consumption when a power saving technique is used. Therefore, the addition of power management techniques is transparent to the energy model and does not change how the model works. In other words, the energy model just views power management to be yet another optimization and how the model works remain unchanged.

2.9 Related Work on Power-Capped Environments

There has been an increasing number of research on power capped environments. Bhalachandra et al., focused on combining CPU clock modulation and power capping to achieve application speedup via MPI runtime, taking advantage of CPU clock modulation's per-core energy control [10]. They modulated the clock for different processes from estimating the slack time and then investigated the performance under a power bound. In MPI, slack time refers to how long processes not on the critical paths are waiting for other processes on the critical paths. In this dissertation, based on our previous work [105], polyhedral compilation, CPU clock modulation, and power capping are combined to evaluate how compiler transformations interact with modulated CPU frequency under a capped power limit. Marathe et al., developed a run-time system called Conductor for power-capped system [54]. Conductor was capable of distributing power across nodes and cores within a HPC system. It intelligently

selected thread concurrency levels and DVFS states under a power bound and balanced power provisioning between nodes according to critical path analysis. Bailey et al., predicted the upper-bound for applications under power-constraint system configurations using linear programming formulation [6]. Again, DVFS state and the number of OpenMP threads form the configuration space. Our work in this dissertation combines the polyhedral transformation work and power management techniques like CPU clock modulation and power capping, into an energy measurement framework. We are among the first to study the relationship between power management techniques and compiler transformations.

Chapter 3

OPTIMIZING 2D WAVE PROPAGATION SIMULATION ON MODERN COMPUTATIONAL ACCELERATORS

3.1 Introduction

Recent developments in the field of high performance computing have greatly expanded the computational capabilities and application of Graphics Processing Units (GPUs). Using GPUs to perform computations that are typically handled by a CPU is known as General Purpose computation on GPUs (GPGPU). To name a few, GPUs are now used in the fields of bioinformatics [92], signal processing [13], astronomy [90], weather forecasting [57], and molecular modeling [20]. In addition to GPUs, Intel's new Many Integrated Core (MIC) architecture also provides a powerful parallel platform for complex computations. The Intel Xeon Phi is the first accelerator based on the MIC architecture and is expected to accelerate oil exploration, climate simulation, and financial analyses, as well as other applications [56]. While new accelerators promise improved computational performance, the use of software tools (like CUDA programming language) to drive parallelism of the accelerators still requires expertise that is not widely available. In addition, CUDA code will only run on NVIDIA GPUs, thereby limiting its portability to other accelerators.

In this dissertation, we sought to overcome the limitations of CUDA by studying how to achieve both automatic parallelization using OpenACC (a directive based language extension similar to OpenMP) and enhanced portability using OpenCL. We applied our parallelization schemes on GPUs as well as an Intel MIC-architecture accelerator to reduce the running time of wave propagation simulations. For results in this chapter, we used a well-established 2D cardiac wave propagation model as a specific case-study.

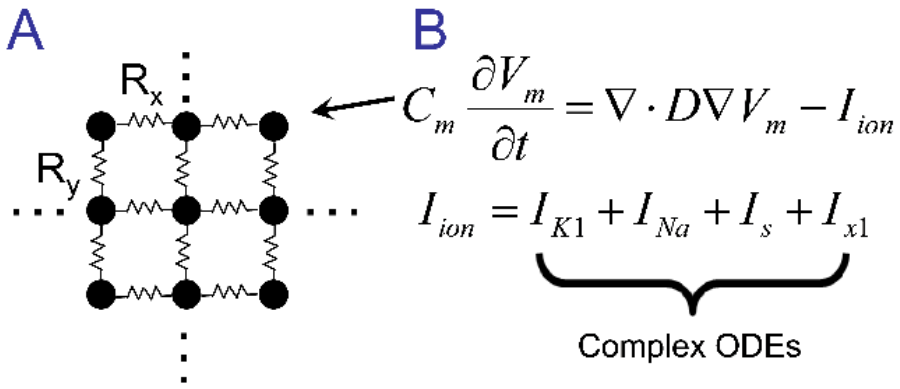


Figure 3.1: (A): Cardiac tissue is modeled as a large geometrical network of nodes that are electronically coupled (B): The electrical potential of the cardiac cell membrane at each node is represented as a large set of partial differential equations

An overview of the cardiac wave propagation model is shown in Figure 3.1. Section 3.2.3 describes the details of the model. Models of cardiac wave propagation often require a large number of computations at each model node at each time step to compute the value of numerous ionic currents at the node. A computational approach that uses parallel processing on GPUs or millions of cores provides a huge performance increase over any sequential approach. For example, Neic et al., accelerated cardiac biodomain propagation simulations using a cluster of GPUs which achieved up to 16.3× speedups over parallelized CPU implementations [59]. Mirin et al., simulated thousands of heartbeats at a resolution of 0.1mm using more than one million cores [58]. They were able to simulate human heart function over 1200 times faster compared with any published results in the field [84]. Unlike prior work that focused on either using one programming language for parallelization on GPUs or multiple programming models targeting CPUs [58, 84, 60, 59, 72, 71], we studied multiple parallelization approaches for running 2D cardiac wave propagation simulations.

We first used OpenACC to automatically generate CUDA and OpenCL GPU code that runs on NVIDIA GPUs. This allows programmers to develop using high-level programming constructs and letting the OpenACC compiler automatically generate

the lower level parallelization details of CUDA or OpenCL. In addition, we developed low-level CUDA and OpenCL implementations for parallelization on NVIDIA GPUs, as well as an OpenMP implementation of the same model for parallelization on Intel CPUs, including Intel multicore processors and the Intel Xeon Phi. In this way we could compare the performance of different parallelization techniques and parallel architectures. By altering the number of threads in the OpenMP implementation, we were able to achieve good speedups on Intel Xeon Phi accelerator.

The contributions of this work are two fold: 1) We parallelized a 2D cardiac wave propagation model using both manual and automatic parallelization paradigms using CUDA, OpenCL, and OpenACC. In particular, auto-parallelizing our model using OpenACC is an excellent example of achieving parallelism in an efficient and effective way. 2) We applied OpenCL, OpenACC, and OpenMP to the problem of simulating cardiac wave propagation so that parallelism from different architectures could be explored. We found that this approach provided excellent speedups of the model on NVIDIA GPUs and the Intel MIC-architecture accelerator. Our results show that OpenACC, OpenMP, and OpenCL are very powerful tools for solving computational models of wave propagation in multi-dimensional media using newly-available computational accelerators.

3.2 Methods

3.2.1 High Performance Computing on GPUs

GPUs are massively parallel multi-threaded devices capable of executing a large number of active threads concurrently. A GPU consists of multiple streaming multiprocessors, each of which contains multiple scalar processor cores. For example, NVIDIA's Fermi architecture GPU card Tesla C2050 contains 14 multiprocessors, each of which contains 32 cores, for a total of 448 cores. In addition, most GPUs have several types of memory, most notably the main device memory (global memory) and the on-chip memory shared between all cores of a single multiprocessor (shared memory).

GPUs achieve high-performance computing through the massively parallel processing power of hundreds or even thousands of compute cores. There are two popular low-level programming languages for GPUs. The first language is CUDA (Compute Unified Device Architecture) [18], a parallel programming model that delivers the high performance of NVIDIA's graphics processor technology to general purpose GPU computing. Applications written using CUDA can run on a wide variety of NVIDIA GPUs. The second language is OpenCL (Open Computing Language) [64], a framework that is similar to CUDA. However, applications written in OpenCL can be executed across heterogeneous platforms not just NVIDIA GPUs. Specifically, OpenCL applications can run on AMD GPUs, NVIDIA GPUs, AMD CPUs, Intel CPUs, and Intel coprocessors, like the XEON PHI. Recent development of high-level directive-based GPU programming allows the programmer to target GPU by placing pragmas in sequential code and the compiler will generate either CUDA or OpenCL parallel code [30]. For example, OpenACC [63] is a directive-based programming extension that can be used to parallelize applications.

3.2.2 Many Integrated Core Architecture

A first-generation Intel MIC-architecture accelerator card (codenamed Knights Corner) contains 60 or 61 cores, and each core supports 4 hardware threads (i.e., a 4-Way SMT). A second-generation Intel Xeon Phi accelerator card (codenamed Knights Landing) contains up to 72 cores (288 hardware threads). One notable feature is that both generations of the MIC cards feature 512-bit wide SIMD vectors processing units (VPUs) which provide fine-grained vectorization parallelism. A single instruction can operate on 8 adjacent double-precision floating point data or 16 single-precision floating point data. Intel MIC accelerators achieve high-performance computing through the hardware threads and the wide vector registers. In contrast to GPUs, the MIC accelerator does not require a host processor to execute the entire application. On the other hand, GPU applications consist of a CPU part that orchestrates the execution of

sections of the application, often referred to as threads that run on the GPU. The programming languages for MIC accelerator include OpenMP, OpenCL, and MPI. In this work, we use OpenMP and OpenCL to exploit the parallelism of one first-generation MIC card.

3.2.3 Cardiac Wave Propagation Model

Our goal was to study the computational speedups for simulating cardiac electrical wave propagation achieved by multiple hardware platforms and several programming languages. We chose to work with a relatively straight-forward 2D implementation of a well-known cardiac action potential model (Beeler-Reuter).[1] This model simplified porting the code between hardware platforms and programming schemes. Although the model is not as complex as models used in state-of-the-art simulations [60, 59], we used it as a case-study that could provide a simplified, systematic approach for comparing modern parallel programming tools. Our experience in parallelizing this model can provide insights for others seeking to parallelize other more complex cardiac models [97] and other propagation models, such as convection and diffusion models [91, 111], seismic wave propagation models [98], and tumor growth and drug transport models [107].

In our cardiac wave model, as shown in Figure 3.1, cardiac tissue is modeled as a large geometrical network of nodes that are electrically coupled. The electrical potential of the cardiac cell membrane at each node is represented as a set of partial differential equations (shown in Equation 3.1).

$$C_m \frac{\partial V_m}{\partial t} = \nabla \cdot D \nabla V_m - I_{ion} \tag{3.1}$$

Transmembrane potential (Vm) at each node in a rectilinear 2D grid ($Nx \cdot Ny$) is computed using a continuum approach with no-flux boundary conditions and finite difference integration [4, 43]. In diffusion models, flux is the amount of a chemical

[1] Action potential is the change in electrical potential associated with the passage of an impulse along the membrane of a cardiac muscle cell.

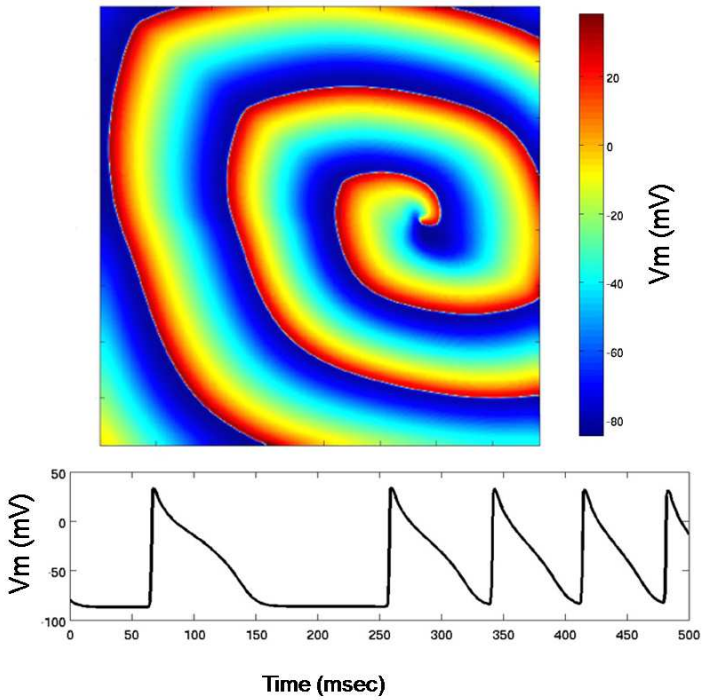


Figure 3.2: Simulation of a single rotor. Top: An image showing cardiac electrical wave propagation as spatial fluctuations of transmembrane potential. A single rotating wave (rotor) is shown. Bottom: Action potentials at one node are shown as the temporal variation of the transmembrane potential at a node.

(e.g., potassium) that passes near a point in space per unit area per unit time. No-flux boundary condition means the chemicals do not go through the walls, i.e., the boundaries in our cardiac model. Standard Euler time-stepping was used. Cardiac membrane ionic current kinetics (I_{ion}, $\mu A/cm^2$) were computed using the Drouhard-Roberge formulation of the inward sodium current (I_{Na}) [21] and the Beeler-Reuter formulations of the slow inward current (I_s), time independent potassium current (I_{K1}), and time-activated outward current (I_{x1}) [7]. These currents are represented as complex

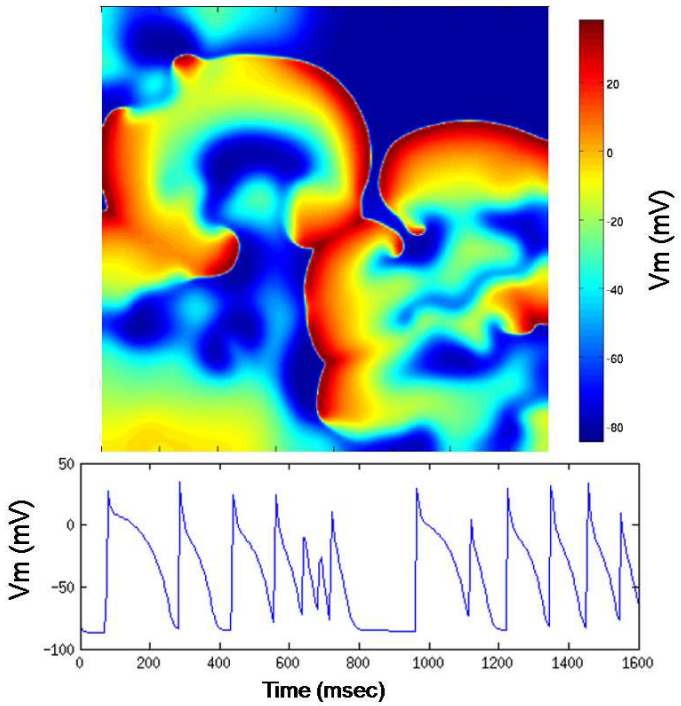


Figure 3.3: Simulation of rotor breakup and fibrillatory activity. Top: An image of transmembrane potential showing complex wave activity. Bottom: Action potentials at one node are shown as the temporal variation of the transmembrane potential at a node.

ordinary differential equations. We simulated functional wave reentry (one rotor), and rotor wave breakup with subsequent fibrillatory wave activity. Figure 3.2 shows an example of simulating reentrant activity with one rotor, and Figure 3.3 shows rotor breakup and fibrillatory activity. For wave reentry, the fiber orientation was typically set at 33 *degrees*; diffusion coefficient along fibers was $0.00076 cm^2/ms$ and diffusion across fibers was $0.00038 cm^2/ms$. For rotor breakup and fibrillatory wave activity, the fiber orientation was set to 0 *degrees*; diffusion coefficient along fibers and across fibers were both $0.00055 cm^2/ms$. All simulations were checked for accuracy and numerical

stability by comparing with the sequential implementation's simulation results.

The general approach for solving the model is shown in Algorithm 1. The differential equations were evaluated independently at each grid node at each time step. Therefore, within each time step there is no data inter-dependency, which fits the modern parallel architectures quite well. There is potential for programming tools that exploit data-level parallelism to provide dramatic computational speedups.

3.2.4 GPU Implementations

Algorithm 1 Generalized code block that is evaluated at each time step to compute transmembrane potential (Vm). Xstep (Ystep) is for iterating nodes in X (Y) dimension of the 2D grid.

```
for Xstep = 1 → Nx do
  for Ystep = 1 → Ny do
    brgates(); //update ionic gating equations
    brcurrents(); //update ionic currents
  end for
end for
bcs(); // set boundary/ghost nodes' potentials
for Xstep = 1 → Nx do
  for Ystep = 1 → Ny do
    Vmdiff(); //update diffusion terms for next time step
  end for
end for
```

As shown in Algorithm 1, the general algorithm loops through each node in a 2D grid. Xstep is the coordinate of the X direction and Ystep is the coordinate of the Y direction. Inside the loop, the same set of functions is evaluated at each node. The temporal loop is outside the nested spatial loops. Because of the sequential structure of the program, total computational time is proportional to the domain area, which means that large spatial domains require significantly longer computational times.

Parallel implementations of N by N dimensional cardiac models are relatively straightforward because once diffusion currents have been computed there is no data dependency between neighboring nodes for a particular timestep. Because of this,

the differential equations that represent myocyte electrophysiology (the brgates() and brcurrents() functions) can be evaluated at each node, in almost any sequence.

CUDA and OpenCL Implementations.

We parallelized the cardiac model using CUDA and OpenCL. In CUDA and OpenCL, multiple threads execute the same instructions but the data processed by these threads might be on different nodes. Take CUDA, for example, a GPU device is conceptualized as a grid containing a large number of equally-shaped blocks, into which the threads are grouped. The parameters of dimGrid and dimBlock define how the blocks (threads) align in the grid (block). Each thread block in the grid executes on a multiprocessor and threads in the block execute on multiple cores inside the multiprocessor. The simulation input set is a 2D grid containing $Nx \cdot Ny$ nodes (Nx columns and Ny rows of nodes). We mapped one GPU thread to one node. To achieve maximum parallelism, the optimal setting for block size must be identified. As mentioned before, inside each GPU there are many multiprocessors and each contains multiple cores. When the GPU code is executing on a GPU, all of the blocks are evenly assigned to the multiprocessors. If a multiprocessor contains 32 cores and a block has 64 threads, the first 32 threads will start executing on 32 cores first, then the next set of 32 threads will be swapped in at some later stage. Assigning large number of threads in one block may increase the occupancy of a multiprocessor, thereby keeping all cores busy. On the other hand, due to the limited resources like registers and shared memory that are available in a GPU, a larger block may increase the pressure on these resources. To find the best block size for our model, we tested our code using various block sizes (i.e. different BlockX and BlockY values) and node numbers (i.e. different Nx and Ny values) for different GPUs. We found that the sweet spot for our cardiac model is 128×1 for both CUDA and OpenCL implementations running on both GPU cards.

The CUDA implementation of our cardiac model is computationally efficient and provides substantial speedups compared to a sequential CPU implementation.

However, CUDA code can only run on supported NVIDIA GPUs. To address this limitation we developed an OpenCL version of the model to support GPU parallelization across platforms. In this work, the OpenCL implementation was tested on the Intel MIC accelerator card in addition to NVIDIA GPUs. The general architecture of our OpenCL implementation is the same as the CUDA implementation. One thread was used to solve the equations at one node in the model.

OpenACC Implementation.

Since we identified the most computationally expensive part of the sequential program, we can add OpenACC directives to offload the code block to run on accelerators like GPUs and other accelerators. The code with OpenACC pragmas is displayed in Algorithm 2 with OpenACC pragmas.

In the code block with OpenACC pragmas, "#pragma acc data" specify which data should be copied to the accelerator (using "copyin") and to/from the accelerator

Algorithm 2 The sequential program with OpenACC pragmas.

```
#pragma acc data copyin(constarr,D,Dp,Afield) copy(datarr)
for T = 0 → final do
  #pragma acc loop independent vector(32) worker(2) gang(256)
  for Xstep = 1 → Nx do
    #pragma acc loop independent vector(32) worker(2) gang(256)
    for Ystep = 1 → Ny do
      brgates(); //update ionic gating equations
      brcurrents(); //update ionic currents
    end for
  end for
  #pragma acc parallel vector_length(1) num_workers(1) num_gangs(1)
  bcs(); // set boundary/ghost nodes' potentials
  #pragma acc loop independent vector(32) worker(2) gang(256)
  for Xstep = 1 → Nx do
    #pragma acc loop independent vector(32) worker(2) gang(256)
    for Ystep = 1 → Ny do
      Vmdiff(); //update diffusion terms for next time step
    end for
  end for
end for
```

(using "copy"). The "#pragma acc loop" specifies the loop that should be parallelized by the OpenACC compiler. The "vector", "worker", and "gang" constructs specify the thread configurations similar to block size and grid size in OpenCL/CUDA. For example, Algorithm 2 is configured to have a grid size of 256 (gang) and a block size of 32×2 (vector,worker). Note bcs function needs to be executed in serialized manner, thus the thread configuration specifies the vector length, the number of workers, and the number of gangs to be 1. The pragmas shown in Algorithm 2 are about all that were needed to drive the generation of efficient parallel code.

3.2.5 OpenMP Implementation

We parallelized the sequential CPU code using OpenMP [65] to provide more perspectives in showing the speedups of parallel GPU implementations. We have two versions of the OpenMP implementation with different number of threads configured for CPUs and the MIC accelerator, respectively. Using OpenMP directive, we can specify as many threads as the number of cores in a single machine so that the best speedup results would be achieved (on this single machine). For the OpenMP implementation running on the CPU, we used a machine that contained 8 cores. In this case, we created 8 CPU threads, and these threads divided among themselves all the computations. With these multiple threads, one of the spatial loops shown in Algorithm 1 could be parallelized. In the case of the MIC card, we created as many threads as the number of hardware threads available. In exploiting the vectorization power of the MIC card, we performed loop unswitching optimization to the OpenMP code. The optimization was used because the array accesses involved $step\%2$ in every statement of the computation region. Loop unswitching yielded better vectorized code in this case.

3.3 Results

In this section, we report the results of experiments with our CUDA, OpenCL, and OpenACC GPU implementations. We also report the results of comparing performances of the GPU implementations with OpenMP implementation. In the end of

this section, we report the speedups from the Intel MIC accelerator.

3.3.1 CUDA and OpenCL Implementations

Hardware

Our GPU implementations of the model were tested on two different GPU cards: a Fermi-architecture GPU card Tesla C2050 and a Kepler-architecture GPU card Tesla K20. In the following sections, we refer to them as Fermi GPU and Kepler GPU respectively. CPU-based implementations were also tested on the machines that hosted each GPU card. The machines hosting the Fermi GPU had 2 Intel E5530 2.4GHz Quad Core Nehalem processors (8 cores total) and 24GB memory. The machine hosting the Tesla K20 GPU card had 2 AMD Opteron 6320 2.8GHz eight core processors (16 cores total) and 16GB memory.

The Fermi GPU had 14 multiprocessors, each with 32 cores (448 cores total) clocked at 1.15GHz and 3GB global memory. The Kepler GPU card had 13 multiprocessors, each multiprocessor containing 192 cores (2496 cores total) clocked at 706MHz. It had 4800MB global memory. The peak double precision floating point performance for Fermi GPU and and Kepler GPU was 515GFlops and 1.17TFlops respectively. These hardware configurations were also used for the experiments comparing against OpenMP implementations.

Scalability and Performance

The computational performance of the GPU implementations (running with the best block size configuration 128×1) was studied by increasing the grid size and measuring total run time on each hardware platform. Simulations of reentrant activity (one rotor) and rotor breakup and fibrillatory activity were studied. Our GPU implementation accommodates large grid size for both simulations. We tested grid sizes up to a maximum of 2048×2048. To ensure appropriate comparisons, the same model parameters were used for simulations running on CPUs and GPUs for each model. The CPU results were obtained from running on the machine that hosted the Fermi GPU.

For the single rotor simulations, we studied grid sizes ranging from 256×256 to 2048×2048. A total time of 100 milliseconds was simulated and dt was varied from 0.025 milliseconds to 0.003125 milliseconds, requiring from $4,000$ to $32,000$ steps for simulations to finish. The number of nodes visited per second ranged from 1024 to 655350. Square grids were used with an edge size of $10cm$.

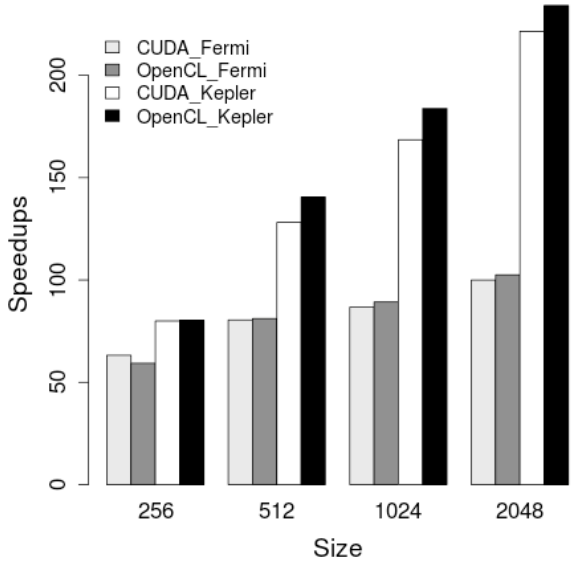


Figure 3.4: Speedups from running on the Fermi-architecture Tesla C2050 GPU and the Kepler architecture Tesla K20 GPU using OpenCL and CUDA for reentrant activity (one rotor) simulation.

Figure 4.5 shows the speedups achieved by the Fermi GPU and the Kepler GPU over the sequential CPU implementation. In the figure, the X axis is the grid size from 256×256 to 2048×2048, and the Y axis is the relative speedup value, computed as CPU time/GPU time. The four bars represent the combinations of runs: OpenCL implementation running on two GPUs and CUDA implementation running on two GPUs.

For both CUDA and OpenCL GPU implementations, the Fermi GPU provided speedups of at least $80\times$ when running large grid sizes, such as 1024×1024 and $2048 \times$

2048. The Kepler GPU provided more than 220× speedups for the 2048×2048 grid size. For each GPU card we observed that larger grid sizes provided larger speedups. Comparing the performance of the Fermi GPU with that of the Kepler GPU, we could see Kepler GPU was more than 2× faster than the Fermi GPU. The increase of the number of cores from 448 to 2496 and the improvement of double-precision performance were among the main factors contributing to the speedups. Since the Kepler results were obtained by running exactly the same code that ran on the non-Kepler GPUs, optimizations specific to Kepler GPU would have given even more speedups. Although the GPU kernel functions were equivalent for the OpenCL and CUDA implementations, OpenCL implementation was slightly faster than CUDA implementation on the two GPUs. One reason is that our OpenCL implementation predefined the total number of threads to be one while executing a kernel function that needed to be serialized; however, our CUDA implementation needed a conditional statement to allow only one thread to run that kernel function which incurred overhead. Another reason may be a difference in the runtimes of OpenCL and CUDA. Overall, both CUDA and OpenCL GPU implementations provided very good speedups on two different GPUs. It took the GPU implementations about 15 (to 30) minutes to finish the largest simulated grid size (2048×2048) on Kepler GPU or Fermi GPU – the sequential CPU implementation took more than 2 days to finish. We noticed that although we copied back from GPU to CPU the transmembrane voltage values every constant timesteps (5 milliseconds of simulation), the IO overhead was insignificant, especially for large grid size. So we excluded the IO time from the total running time for GPU implementations. The transmembrane voltage values for all nodes reside in GPU global memory. There is no need to copy the voltage values back to CPU in every single timestep. Fortunately, the GPUs we used contained enough global memory to hold the voltage values for each node even for the largest grid containing 2048×2048 nodes. For larger grid sizes, values will have to be copied back to and from the CPU as the entire grid is computed.

Rotor breakup simulations were used to study performance during greater computational loads. In these tests, we set the model parameters to simulate the breakup

of a single rotor into multiple rotors, resulting in electrical activity that is much more complex than that of a single rotor (shown in Figure 3.3). The computational load for such simulations is typically higher because more nodes are active per unit time. Despite the fact that the rotor breakup simulation is much more complex, we observed similar significant speedups from GPUs.

3.3.2 OpenACC Implementation

The hardware used to test our OpenACC implementation was a machine hosting a Kepler GPU. To compile OpenACC code, we used the HMPP 3.3.3 compiler from CAPS.[2] The compiler transforms OpenACC code to either CUDA or OpenCL code as specified by the user.

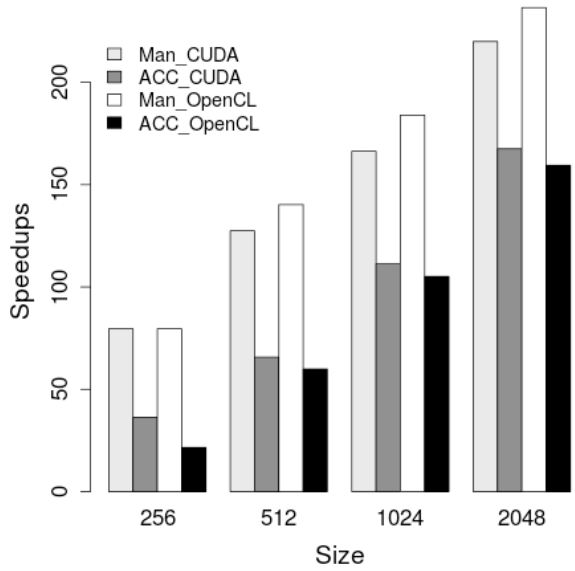


Figure 3.5: Speedups of hand-written GPU code (Man_CUDA, Man_OpenCL) over the sequential baseline vs. speedups of OpenACC targeting CUDA and OpenCL (ACC_CUDA, ACC_OpenCL) over the same baseline. All GPU codes were run on the Kepler GPU.

[2] As of June 2016, the compiler from CAPS is not available because CAPS went out of business. PGI compilers should be used for reproducing the experiments.

The speedups of automatically generated GPU code (CUDA/OpenCL) and hand-written GPU code (CUDA/OpenCL) are shown in Figure 3.5. The tests were run on Kepler GPU and the same reentrant activity as above was simulated. We can see from the figure that OpenACC code targeting CUDA and OpenCL achieved more than 150 times speedup over the sequential implementation with the largest input size and over 100 times speedup with the second largest input size. Although automatically generated GPU code did not achieve the same speedup as hand-written GPU code, the amount of modification to the original sequential code from OpenACC was trivial and significantly less than the hand-written counterpart. We believe OpenACC to be a quick way of exposing parallelism of potential applications and could serve as a transition between sequential CPU implementations and a more performant, but harder to code GPU implementation.

3.3.3 OpenMP Implementation

To get a deeper perspective on the speedups achieved by our GPU implementation, we conducted additional tests with a parallel CPU implementation of the model using OpenMP. The OpenMP implementation was run on the hardware platform that hosted Fermi GPU and with the same reentrant activity (one rotor) simulation input files. We achieved an average speedup of 7.15 over sequential CPU implementation using 8 CPU threads, which is almost 90% of full occupancy on each core.

Figure 3.6 shows the speedup results over OpenMP for hand-written CUDA and OpenCL implementations running on the Fermi GPU and the Kepler GPU. All GPU implementations provided more than 10 times speedup over the multi-core OpenMP implementation running over large input size on Fermi GPU. The speedup for both CUDA and OpenCL implementations increased quickly from about 12 times to more than 30 times running on Kepler GPU with the increase of problem size. The maximum speedup of 31 over OpenMP was achieved by running OpenCL implementation on the Kepler GPU.

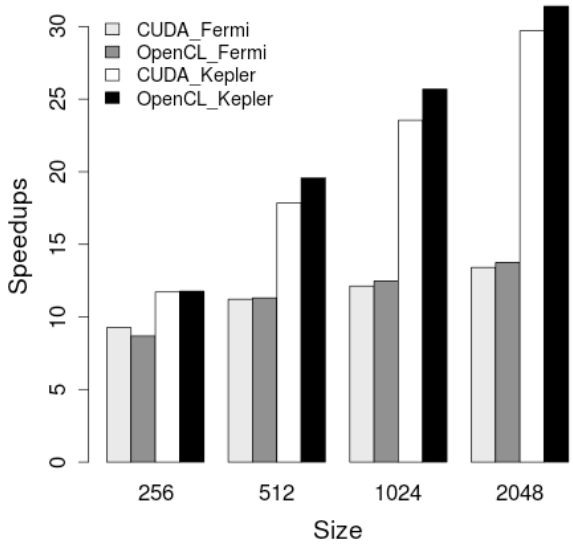


Figure 3.6: Speedups on Fermi GPU (NVIDIA C2050) and Kepler GPU (NVIDIA K20) over the 8 core OpenMP implementation. Reentrant activity (one rotor) was simulated.

3.3.4 Implementations on MIC Architecture

We used an Intel Xeon Phi 5110P coprocessor to test our hand-written OpenCL code, automatically generated OpenCL code (by OpenACC), and the OpenMP code written for the Intel MIC architecture. The coprocessor contained 60 cores and supported 240 threads. The memory of the coprocessor was 8GB. The machine hosting the coprocessor had 32 GB of memory and the CPU was Intel Xeon E5 clocked at 2.63GHz. We used OpenCL 1.2 library (provided by Intel ICC compiler v14.0.0) for OpenACC and OpenCL implementations. The compiler we used to compile our hand-written OpenCL code was GCC. We used the CAPS compiler to compile OpenACC code to generate OpenCL code that ran on the Intel MIC card. For OpenMP implementation, we used the Intel ICC compiler versioned 14.0.0.

Figure 3.7 shows the speedups achieved on MIC accelerator from the three implementations. The OpenMP implementation was the fastest of the three. It achieved

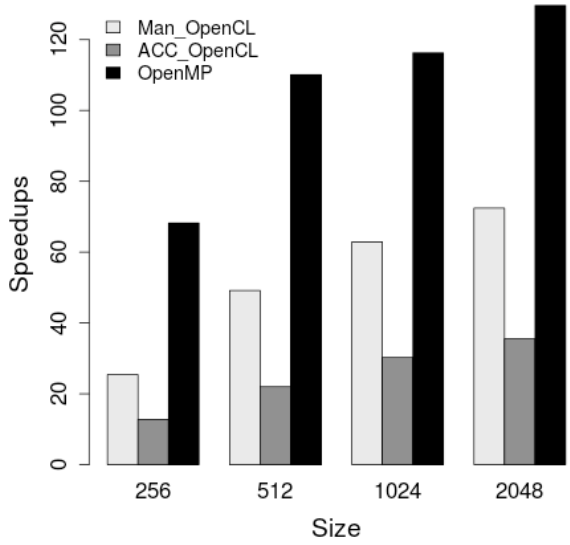


Figure 3.7: Speedups on MIC-architecture Xeon Phi coprocessor using hand-written OpenCL, OpenACC-generated OpenCL, and OpenMP implementation over the sequential implementation. Reentrant activity (one rotor) was simulated.

more than 120× speedup for the largest simulation size. For smaller simulation size like 512 and 1024, it achieved more than 100× speedup. The MIC OpenMP implementation was vectorized with a factor of at least four because if we disable the vectorization, the speedup obtained was four times less. The MIC OpenCL implementation remained identical to the implementation tested on GPUs, thanks to its portability. It provided decent speedups on the MIC while offering good portability. For the largest simulation size, more than 70× speedup was achieved. As the OpenCL library improves, we expect the speedup number to get better. In contrast to the GPUs, OpenACC only provided a maximum of 35× speedup on the MIC. For these experiments, OpenMP was better than OpenACC for achieving good performance on the MIC.

3.4 Discussion

In this section, we first summarize the optimizations applied to different implementations. Then we compare different programming languages with various metrics. In the end, we compare with the related work and summarize this work.

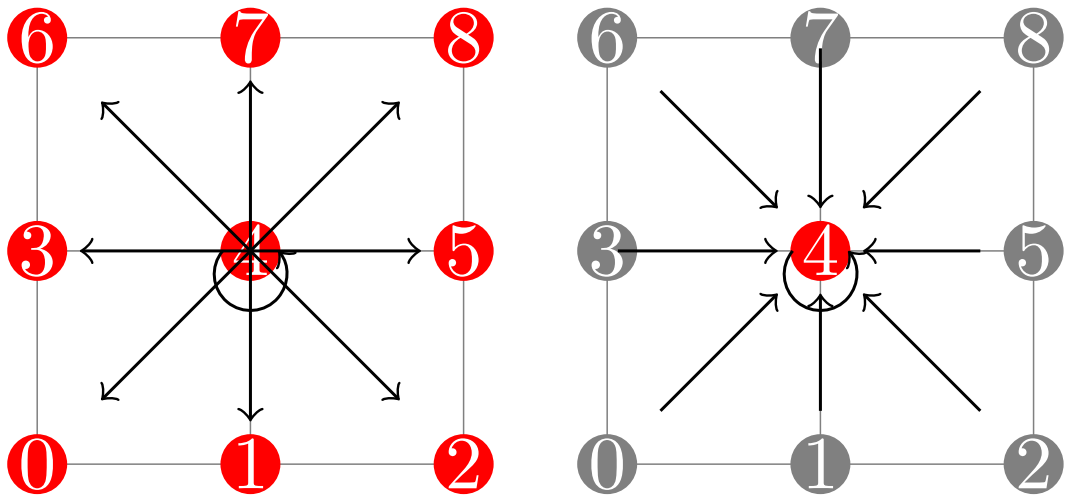
3.4.1 Summary of Effective Optimizations for Different Implementations

Although it was straightforward to port the cardiac model to run on accelerators, optimizing the code was not trivial.

For CUDA and OpenCL implementations on GPU, we applied two other effective optimizations before optimizing the block size:

1) Eliminating atomic operations. A direct port of the sequential code contained 9 equations that represent the updates (writes) from a current node to the 8 neighboring nodes and itself, as shown in Figure 3.8(a). When the ported GPU code is executed, multiple GPU threads will update the same model node, causing a race condition. Before optimization, atomic operations (in CUDA) and kernel isolations (in OpenCL) were used to avoid racing updates. Atomic operations are expensive and kernel isolations bring synchronization overhead. To address this limitation, we came up with a neighbor-update-free strategy, as shown in Figure 3.8(b). In this optimization strategy, each node "collects" (reads) update requests from the neighbors and performs the update to the node itself. No two GPU threads will update the same memory location, therefore only the center node is updated by the GPU thread that is mapped to the node. 2) Coalescing memory accesses. Coalesced memory accesses on GPUs means the data requests (e.g. by 32 threads) consisting of contiguous (e.g. 256 bytes for double) aligned memory space could be fulfilled by one memory access. More than one memory access will be needed otherwise. Every node of our model is associated with 13 attributes. These attributes of all model nodes (N by N, for example) can be stored in two ways: A[N][N][13] or A[13][N][N]. Storing attributes using A[N][N][13] is like storing a 2D array (A[N][N]) of structures containing 13 struct-members. In contrast, storing attributes using A[13][N][N] is like storing a structure of 2D arrays.

We achieved coalesced memory access by changing Array of Structures (AoS) to Structure of Arrays (SoA). Without coalesced memory access, the speedup numbers would get cut by at least a half.



(a) This figure shows that a GPU thread (mapped to the center node Node 4) writes to all the neighboring nodes and Node 4. A red node means its value is changed by the GPU thread.

(b) This figure shows that a GPU thread (mapped to the center node Node 4) reads values from all neighboring nodes and updates Node 4. A red node means its value is changed by the GPU thread.

Figure 3.8: This figure shows two ways of executing a critical piece of code of the model on GPUs. Suppose a GPU thread is mapped to the center node. This code involves updating each model node's potential with its neighboring nodes' potentials from a previous iteration.

We obtained the OpenACC implementation from adding pragmas to a sequential implementation that incorporated the above two optimizations. Added with the optimization for gang, worker, and vector configuration, we achieved good speedups from OpenACC on GPUs.

For the MIC OpenMP implementation, we applied a loop unswitching optimization to the dominant time-step loop in addition to the above optimizations so that the vectorization power of a MIC card could be better exploited. The loop unswitching strategy improved the speedup results by approximately 40%.

For the sequential and OpenMP version that run on CPUs and serve as the baselines, we passed O3 optimization flag to the compiler so that the implementations were well optimized to be good baselines.

3.4.2 Code Metrics to Compare CUDA, OpenCL, OpenACC, and OpenMP

Table 3.1: Comparison of multiple metrics between different parallel programming implementations for the cardiac wave propagation model.

Language	Code-Change	Estimated Time to Program	Platforms
CUDA	500	Weeks	NVIDIA GPUs
OpenCL	500	Weeks	GPUs, CPUs, MIC accelerator
OpenACC	10	Days	GPUs, CPUs, MIC accelerator
OpenMP	10	Days	CPUs, MIC accelerator

In this section, we compare metrics like lines of source code change to the original implementation, portability, time taken to program, of each CUDA, OpenCL, OpenACC, and OpenMP implementation. Table 3.1 describes information for each of the different programming languages and extensions.

The Code-Change (second column) reports the estimated number of lines of code change for the kernel computation functions from the sequential C code. There are about 220 lines of computation code in the sequential CPU implementation. The CUDA and OpenCL implementation needed to replace almost every statement of the original CPU implementation. For CUDA and OpenCL, we need to include GPU initialization and data preparation code. This includes initialization of GPU, data initialization for GPU, and data movement between GPU and CPU. The code change was mostly addition and replacement of statements. The OpenACC implementation only required 10 additional statements to the sequential code. For OpenACC, all data movement and initialization are automatically handled in the generated CUDA and/or OpenCL code. The programmer only needed to figure out the correct pragmas and associated pragma attributes to add. The OpenMP implementation was also obtained by only placing a few pragmas around the parallel code regions. Considering the difficulty

of programming using the languages (third column), The difficulty of programming in either CUDA or OpenCL is about the same, and would take an experienced programmer weeks to get the program correct and optimized for GPUs. In our case, it took us months to to implement and tune our best CUDA version. In contrast, OpenACC and OpenMP programming mostly involved figuring out where to place the pragmas and what parameters should go with these pragmas. It did not require the programmer to be an experienced GPU programmer, in contrast to CUDA programming. It took us less than a week to get efficient OpenACC and baseline OpenMP implementations. We also compare the portability of these implementations (fourth column). The CUDA implementation can only run on NVIDIA GPUs while OpenCL can run across different architectures include GPUs, CPUs and accelerators like Intel Xeon Phi coprocessor. Since an OpenACC implementation can be compiled to OpenCL code, it can target the same architectures as OpenCL can. The OpenMP program can run on CPUs and Intel's Xeon Phi coprocessor (accelerator). In addition, the most recent OpenMP specification [66] introduced the standards of a new "target" directive for GPU execution.

Combining the performance results and the above metrics, we can see the OpenCL implementation achieved the best speedups and portability on GPUs; the OpenACC implementation, taking the minimum amount of effort to program, also achieved very good speedups on GPUs and the same portability as OpenCL implementation did. For Intel MIC architecture, the OpenMP implementation was the performance champion. We attribute this partially to the performance differences between the compilers and the (OpenCL) libraries used to generate the executables. As OpenACC compilers improve, we expect the performance of OpenACC applications will be on par to coding in lower level languages to implement parallelism.

3.4.3 Related Work

The first published simulation of 2D wave propagation in cardiac arrhythmias model using GPU hardware was performed on an Xbox 360 and resulted in a speedup

of more than 20× for specific model parameters [86]. Our work, based on our previous work [103, 106], is the first to report the simulation results of the 2D wave propagation using CUDA, OpenCL, and OpenACC to manually (CUDA/OpenCL) and automatically (OpenACC) take advantage of the power of modern computational accelerators like GPUs and Intel Xeon Phi coprocessor. Wienke et al. were the first to report the experiences with real-world applications using OpenACC [108]. Their OpenACC implementation achieved a fraction of 80% of the best performance offered by OpenCL implementation for one application and only 40% for a more complex medical program. As shown in Figure 3.5, our OpenACC implementation achieved about 70% of the best hand-written CUDA/OpenCL implementation. In this work, we also showed that the OpenACC implementation outperformed a non-optimized hand-written OpenCL code on the Intel MIC accelerator (Figure 3.7). Hart et al. first showed that OpenACC could be used in massively-parallel, GPU-accelerated supercomputers [34].

Work that compared OpenACC, OpenCL, and CUDA include the acceleration of hydrocodes [35] and the acceleration of financial applications [30]. Our case study is focused on a different codebase (2D wave propagation). While the work of accelerating hydrocodes showed that OpenCL performed the worst, we reported that the OpenCL implementation could achieve both good performance and portability. In fact, manual OpenCL often outperforms manual CUDA implementations. Different from the work of accelerating financial applications, we went a further step to compare the performance of the OpenCL, OpenACC, and OpenMP implementation on the Intel MIC accelerator. There is also related work that focused on comparing the different implementations of the OpenACC directive-based language itself [83, 82]. Oliveira et al. compared the CUDA, OpenCL, and OpenGL implementations of the cardiac monodomain equations [62]. In their work, the OpenCL implementation was slower than the CUDA implementation. Our work showed that both the hand-written OpenCL code and the OpenACC generated OpenCL code are competitive with CUDA code while achieving portability.

3.5 Summary

This work presents our efficient GPU implementations of a wave propagation model that achieve good scalability on modern computational accelerators. More importantly, we auto-parallelized the sequential code version of the model using OpenACC, the speedup of which was impressive–as much as 150× faster than the sequential version. By using OpenACC, modifications to the original sequential program were kept to a minimum. We have also compared the performance of parallel GPU computations with parallel CPU computations (using OpenMP). Our GPU implementation was as much as 200× faster than the original sequential CPU code. The OpenMP code achieved 7.15× speedups over the sequential CPU code on average as displayed. Comparing with the OpenMP code, our GPU implementation reached more than 30× speedups on the Kepler GPU. We also tested with different implementations on Intel MIC architecture accelerator where the OpenMP implementation achieved the best speedup of any OpenMP implementation of more than 120× and the portable OpenCL implementation achieved more than 70× speedup.

We conclude that emerging software developments, like the OpenACC directive-based programming language, facilitate exploiting application parallelism offered by evolving hardware architecture. Our method of using OpenACC, OpenCL, and OpenMP to achieve efficient and effective parallelization on different accelerators can be generally applied to benefit other domains using wave propagation.

Chapter 4

ENERGY TUNING USING THE POLYHEDRAL APPROACH

Understanding the impact of energy consumption and execution time that compiler optimizations have is important for the tuning of tune applications. To answer the question whether one can simultaneously optimize for energy and performance using polyhedral compilers, we performed experiments using both the light-weight fine-grained RCRdaemon energy measurement tool and the Polyhedral Compiler Collection tool, or PoCC. Using this setup, we can measure both the energy consumption and the performance of a large number of program variants constructed by applying various optimizations. The studies were performed on both an Intel Sandy Bridge architecture system and a Xeon Phi architecture system. We studied the impact of optimizations on energy and performance on a variety of benchmarks. We also studied energy and performance of the cardiac wave propagation application we used in Chapter 3.

4.1 Methods

PoCC could generate thousands of parallelized OpenMP program variants with different optimizations of the same benchmark. We used a version of the ROSE source-to-source compiler [80] that was modified and used to find OpenMP parallel regions and add RCRtool API calls around them. ROSE tracks original file names and source line number, allowing simple parallel region identification. The overhead of the RCRdaemon is negligible on both Sandy Bridge and Xeon Phi architectures. It enables us to measure the energy consumption of the application with a granularity of about one millisecond. When the program finishes execution, the elapsed time, the amount of energy used (in Joules), and the average computed power (in Watts) of each parallel

regions and the whole application are output. Additional information such as processor temperature is also available after application shut-down.

The Intel MIC architecture in the Intel Xeon Phi chips is a recent addition to the Intel processor offering. The Xeon Phi works as a coprocessor typically on a host server with architectures like Sandy Bridges. The Phi accelerator cards we had available contain 61 cores, each core supports 4 hardware threads. One notable feature is the 512-bit wide SIMD vectors providing fine-grain vectorization and high floating-point performance for each thread. With the wide vector registers, a single instruction can operate on 8 adjacent double-precision floating point data or 16 single-precision floating point data. The cores, threads and vector unit combine to achieve well over a Teraflop from a single socket.

RCRdaemon can collect power information either natively on the accelerator or on the host. Users can track power usage natively in microWatts through the sysfs system file (/sys/class/micras/power), which is updated every 50 millisecond. RCRtool monitors the power at user level and computes the energy consumption over time. The RCRTool provides a consistent API on the Xeon Phi as well as the Sandy Bridge. On the host, RCR collects power information of the coprocessor using the MICAccessSDK API provided by Intel. The granularity is the same 50 milliseconds.

4.2 Benchmarks and Experimental Setup

We evaluated three kinds of programs for energy auto-tuning with polyhedral framework: the Polybench benchmark suite, the LULESH program [42], and the cardiac wave propagation application developed and frequently used by our collaborators [43]. PoCC was used to optimize Polybench and, PolyOpt was used to optimize LULESH and the cardiac simulator.

4.2.1 Polybench

Previous work has obtained significant speedups with the polyhedral framework for the Polybench programs[70, 68, 69]. We extended this work to examine whether the

best tuned variants are also the most energy efficient. Using PoCC, program variants were generated using different combinations of optimizations from the following five groups:

- Loop fusion: smartfuse, maxfuse, nofuse
- Loop unrolling factor: 1, 2, 4, 8
- Loop tiling: 1, 16, 32, 64. Note that the number of different flags depends on the level of nested loops
- Loop vectorization: on, off
- Loop parallelization: on, off

The Tiling Hyperplane method[11] is used to perform loop tiling. Loop fusion is performed to minimize loop overheads. Depending on reuse patterns, fusion can increase or decrease locality. Similar to Park et al., [69] 1) nofuse results in no loop fusion 2) smartfuse only fuses statements that carry data reuse and are at similar nesting levels 3) maxfuse performs all legal loop fusion. Good loop fusion optimizations improve temporal locality. All three fusion types are evaluated. For a triply nested loop, applying all possible combinations of the above optimizations generated 5135 program variants. The ROSE source-to-source compiler was used to add energy profiling calls to each variant automatically. GCC(4.4.6) generated the final executable. During the execution of an application, periodic queries to the RCRtool blackboard provide energy consumption information. Figure 4.1 gives the workflow for measuring energy consumption of Polybench programs using the polyhedral compiler framework enhanced with energy measurement capability. We used the ROSE source-to-source compiler to add the APIs because phase entries and exits are very clear in the source code level for OpenMP applications. Note that binary instrumentation can alternatively be used to add energy profiling instrumentation, as in work done by Cicotti et al. [17].

4.2.2 LULESH

LULESH[42] is a mini-version of Arbitrary Lagrangian-Eulerian 3D (ALE3D), a multi-physics numeric simulation software tool. We used the OpenMP implementation

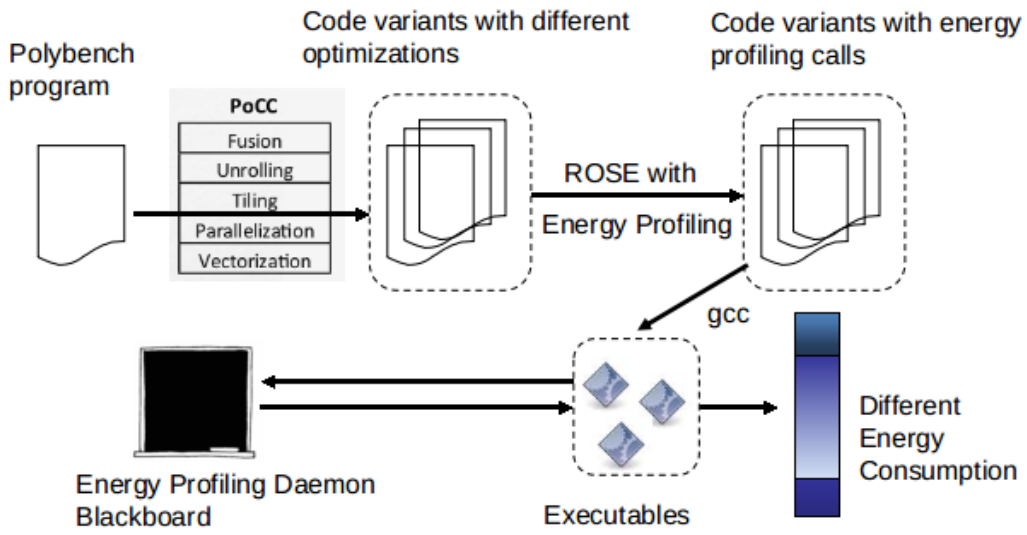


Figure 4.1: The workflow of obtaining energy consumption of polyhedral optimized Polybench programs.

v1.0 for our experiments. The original LULESH uses a block structured mesh accessed via an indirect reference pattern[42]. To make LULESH amenable to transformations by a polyhedral compiler, we modified it to resolve all indirect array accesses. Although doing this oversimplified LULESH, it allowed us to study the energy and time relationship of polyhedral compilation techniques with LULESH.

LULESH OpenMP implementation contains 30 parallel regions, 6 of which take up more than 60% of the total application time[74]. We manually converted the two most significant parallel regions to two SCoPs so that they could be passed to our polyhedral framework. Static Control Parts (SCoP) refer to code regions that are amenable to polyhedral transformations. A SCoP should conform to certain constraints. For example, indirect memory access and complex function calls are not allowed in a SCoP. The largest SCoP that we obtained from manual conversion contained too many dependences and we found it hard for the polyhedral compiler to finish transformation and parallelization: when all the temporary variables were eliminated from the most computationally intensive loop to create a SCoP, the resulting code was greatly expanded

and required hours of compilation to generate just one variant. In this work, optimizing the 2nd (largest) SCoP of LULESH was the focus. 200 program variants were produced by applying loop fusion (maxfuse and smartfuse), loop tiling, vectorization, and parallelization. The execution time and energy of each was measured.

4.2.3 Cardiac Wave Propagation

In addition to the Polybench programs and LULESH, the 2D monodomain cardiac wave propagation simulation application (named *brdr2d*) from Chapter 3 was used as a benchmark for experiments in this chapter. Recall that its model involves solving a set of ODEs and PDEs and is well-known in the computational cardiac modeling community[43]. The sequential C implementation was more than 1K lines.

One loop nest took more than 90% of the total application execution time. This dominate loop nest is an ideal situation for the polyhedral compiler. The loop nest was inside a while loop and was executed many times. This code structure is not unique to cardiac wave propagation simulation. Computationally dominate loops are often found inside either while loops that execute until some termination condition or inside a simulation time-step loop. LULESH also falls into this category with multiple loop nests within a time-step loop.

While PolyOpt sometimes cannot extract any SCoPs from a program, it does output information useful to the user so that they can manually transform the application to modify loops into SCoPs. For the application brdr2d, PolyOpt generated enough information so that we were able to transform two loops into SCoPs. To expose the SCoPs the following changes were required. The computation part of *brdr2d* was fully inlined removing all function calls. Then, all array indexes were changed to be affine functions of the loop iterators. This involved loop unswitching. Loop unswitching moves a conditional inside a loop nest outside and duplicates the loop body for the if and else clauses of the conditional. Here, loop unswitching is used to to specialize modular operations like *step* % 2. Finally, the number of dependencies was reduced

```
2
3 /** Original Version *****************/
4 double abfun(double a) {
5  return exp(a+0.1)*0.2;
6 } //end abfunc
7
8 //begin computation
9 step = 0;
10 while (step < T) {
11   for (i=0; i<M; i++) {
12     for (j=0; j<N; j++) {
13       double temp0,temp1,temp2,v;
14
15       temp0 = abfun(a[step%2][0][i][j]);
16       temp1 = abfun(a[step%2][1][i][j]);
17       temp2 = abfun(a[step%2][2][i][j]);
18
19       v = (temp0 + temp1) / temp2;
20     }
21   }
22
23   step++;
24
25 }
26 //end computation
27
```

```
2
3 /** Transformed Version ****************/
4 step = 0;
5 while (step < T) {
6   if (step % 2 == 0) {
7     for (i=0; i<M; i++) {
8       for (j=0; j<N; j++) {
9         double v;
10        v = (exp(a[0][0][i][j]+0.1)*0.2  +
11             exp(a[0][1][i][j]+0.1)*0.2) /
12             exp(a[0][2][i][j]+0.1)*0.2 ;
13      } //end for
14    }  //end for
15  } //end if
16  else {
17    for (i=0; i<M; i++) {
18      for (j=0; j<N; j++) {
19        double v;
20        v = (exp(a[1][0][i][j]+0.1)*0.2  +
21             exp(a[1][1][i][j]+0.1)*0.2) /
22             exp(a[1][2][i][j]+0.1)*0.2 ;
23      } //end for
24    } //end for
25  } //end else
26  step++;
27 } //end of while
```

Figure 4.2: A simplified version of the original LULESH loop nest and the same loop nest transformed using loop unswitching.

by forward substitution of temporary variables. After these changes, PolyOpt automatically detected the code region and applied various transformations to the SCoPs. Figure 4.2 shows a simplified version of the original and the transformed loop nest.

Program variants were generated to explore data locality and parallelism using loop fusion (smartfuse/maxfuse), different tiling sizes, vectorization and auto-parallelization. For those variants when parallelization was "on", OpenMP pragmas were automatically generated for each variant. The original sequential C implementation had all required OpenMP pragmas manually added to serve as a baseline. Four different input files for *brdr2d* were used to study how the performance of the program variants is impacted by different input sizes.

4.2.4 Experimental Setup

The tests ran on a 2-socket 8-core Intel Xeon E5-2680 processor with 20MB (40MB total) L3 cache. PoCC v1.2 was used to generate program variants. The extra large data set (specified in Polybench) was used. A few modifications were made to

ROSE (version timestamped 1370387370) to insert the energy API calls. GCC v4.4.6 was the backend compiler. Every executable was compiled with the -O3 optimization flag. To protect against low start-up energy/power measurements, the system was warmed up with a computational intensive program before any test was executed.

We also performed experiments on a Xeon Phi coprocessor. The Phi architecture accelerator card contained 61 cores clocked at 1.09GHz. Each core had 512KB of L2 cache. Each program variants was compiled with the ICC v14.0.0 compiler to generate the final executable, producing OpenMP programs that ran natively on the Phi. The polyhedral framework was first used to examine the energy usage (and the execution time) of the Polybench programs and LULESH on the Intel Sandy Bridge architecture. In particular, LULESH was rewritten to allow easy manipluation in a polyhedral compiler framework. A more realistic application, *brdr2d*, was then studied, which was run on both the Intel Sandy Bridge and the Intel Xeon Phi architecture.

4.3 Execution Time and Energy Consumption Correlation

The experiments in this section verify the relationship between execution time and energy consumption.

4.3.1 Polybench

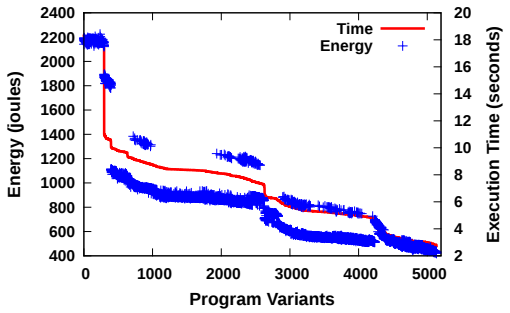
The Polybench v3.2 benchmark suite contains 30 programs. Because of the large number of variants created, the energy consumption of three (*covariance* benchmark, *2mm* benchmark, and stencil *seidel-2d* benchmark) were chosen for closer examination. Figure 4.3(a) shows the relationship between the execution time and the energy usage for the 5135 variants of the *covariance* benchmark, sorted by execution time. The left y-axis shows the energy consumption (in joules) and the right y-axis shows the execution time (in seconds).

There is clear correlation between the time and the energy in this graph. The energy line (blue plus and mostly below the execution time) generally follows execution time. The best optimized program variant for *covariance* for time (bottom right in the

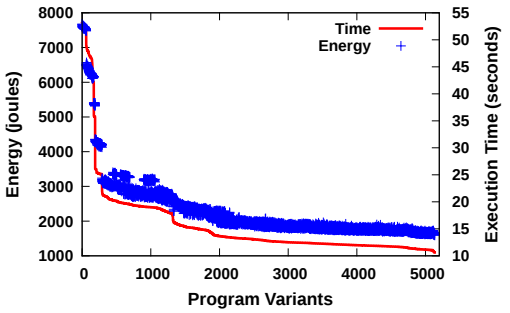
figure) consumed the least amount of energy. The energy line has many places where 2 runs that take the same amount of time consume significantly different energy. These appear as spikes in the graph. Examining the covariance optimization results closer, we noticed that the higher energy usage value always had the "maxfuse" flag set. The last jump in Figure 4.3(a) is at variant 4236, above which all executions have "maxfuse" set. The executable with "maxfuse" requires significantly more power than with either "smartfuse" or "nofuse". For the executables where no performance improvement is gained, this has noticeable energy costs. However, when the polyhedral framework finds the correct tiling size, the "maxfuse" flag produces a significantly faster executable (note the change in the execution curve that occurs at variant 4236). We believe that "maxfuse" exposes more instruction-level parallelism to the hardware resulting in faster execution. For poor tile sizes this results in an increase in the number of concurrent memory accesses and an increased power demand by the application. With the proper tile size execution time and energy is minimized. The "smartfuse" and "no fuse" options produce executables that run at lower power and may be beneficial if peak power usage is a constraint, but longer execution times (up to 2×) result in significantly higher overall energy costs. The interaction between optimizations can have significant impact on their effectiveness for both time, energy and power.

We can see a similar correlation between execution time and energy occurs for the *2mm* benchmark (as shown in Figure 4.3(b)). No single optimization has a large effect on power as "maxfuse" did in the previous example. The spikes that do occur (especially the left side) are from poor tiling configurations. Power is approximately constant for all the runs, so energy consumption is a function of execution time.

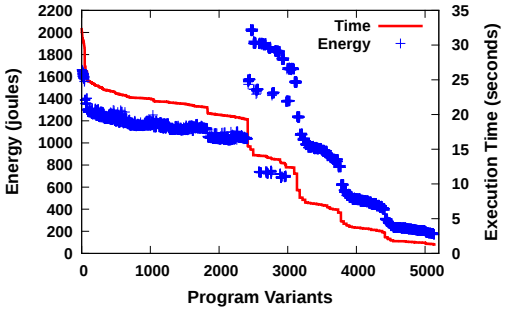
For the stencil benchmark *seidel-2d*, Figure 4.3(c) shows when the execution time becomes lowest, the energy consumption is also minimal. The jump in the energy curve occurs for all variants with parallelization turned on. Power for non-parallel variants is less than 60 Watts. Power for the parallel variants are between 110 and 135 Watts. No other optimizations have as significant effect on the power or energy usage.



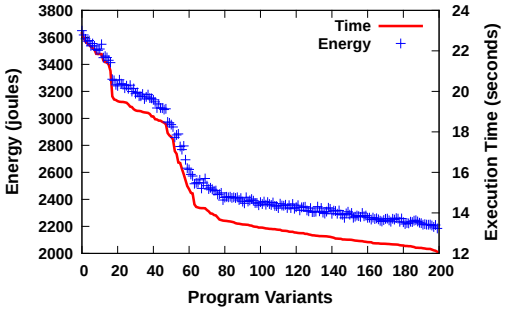
(a) The relationship between the execution time and the energy consumption of all *covariance* Polybench program variants on Sandy Bridge Processor (sorted by execution time). The spikes that happen at around variants 750, 1950, and 4236 are caused by "maxfuse" loop transformation.



(b) The correlation between the execution time and the energy consumption of *2mm* Polybench on Sandy Bridge Processor (sorted by execution time). The spikes that happen at around variants 500 and 1000 are caused by bad tiling configuration.

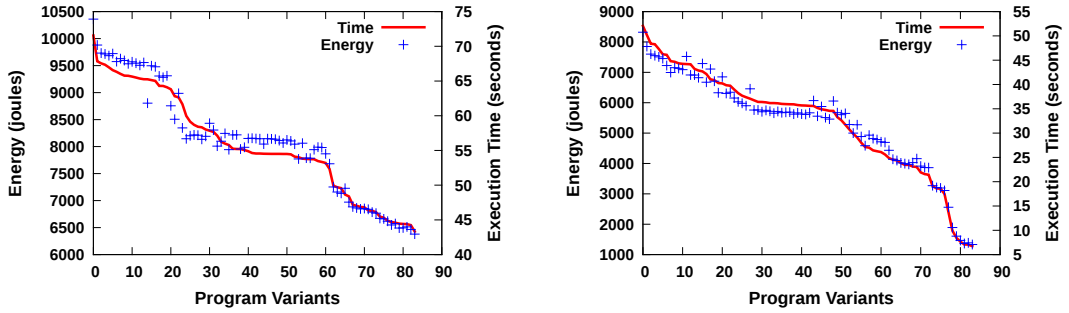


(c) The correlation between the execution time and the energy consumption of stencil *seidel-2d* Polybench on Sandy Bridge Processor (sorted by execution time). Jumps in energy usage (and decreased execution time) are results of turning parallelization on.



(d) The correlation between the execution time and the energy consumption of LULESH program on Sandy Bridge Processor (sorted by execution time).

Figure 4.3: Execution time and energy consumption correlation of Polybench programs (covariance, 2mm, and seidel-2d) and LULESH.



(a) Time and energy correlation on Sandy Bridge

(b) Time and energy correlation on Xeon Phi

Figure 4.4: Graphs showing the correlation between the execution time and the energy consumption of *brdr2d* on a Sandy Bridge processor and on a Xeon Phi architecture (both sorted by execution time).

4.3.2 Modified LULESH

For 200 variants of LULESH, Figure 4.3(d) shows the energy used and execution time. The energy curve mirrors the execution time. A slight ($< 2\%$) run-to-run variation in the energy, presents a minor opportunity for energy tuning beyond execution time. LULESH optimizations overall provide almost a $2\times$ reduction in execution time (22.9 vs 12.1 seconds - 47% reduction) and a significant decrease in energy (3650 vs 2185 Joules - 40% reduction). No single optimization resulted in a significant increase in power, although the power required did rise slightly (from 160 Watts to 180 Watts - 12% increase).

4.3.3 The Cardiac Wave Propagation Application

brdr2d contains two symmetric SCoPs (because of loop unswitching). Each SCoP contained 42 statements. The number of dependencies between these statements was 638 (there were no loop carried dependencies). PolyOpt detected and applied loop fusion (maxfuse or smartfuse) and loop tiling transformations (various tile sizes) as well as vectorization and auto-parallelization to the SCoPs. The fastest 84 program variants were chosen for study on both the Sandy Bridge processor and Xeon Phi coprocessor.

The other program variants did not finish execution before preset timeouts. 49 of the 84 considered programs had "maxfuse" flag turned on.

Figure 4.4 compares execution time and energy consumption (for input size of 2048) for the cardiac simulation application on the Sandy Bridge processor and on a Xeon Phi card. Both Figure 4.4(a) and Figure 4.4(b) show that the energy tracks the time. Saving energy consumptions is consistent with improving performance on both processors. The *brdr2d* application has a small number of loops and one dominant loop. In Figure 4.4(a), the effect of fusion (smartfuse and maxfuse) on power was small (less than 10 Watts difference). Some variants with "bad" tile sizes required less power/energy (indicated by the energy drops). Overall the effect of fusion was insignificant. Figure 4.4(b) has the time line above the energy line for the left half but below for the right half. We noticed for the Phi, that the "smartfuse" option clearly used lower power than "maxfuse" (at least 20 Watts). However, the performance of "maxfuse" was much better than "smartfuse" and the overall execution time and energy use for "maxfuse" was lower (up to 5×). On our Xeon Phi, the "maxfuse" optimization setting combined with a good tiling size exposed more parallelism that the processor could take advantage of. The fastest execution times occurred with "maxfuse" and good tiling sizes.

4.4 Polyhedral Optimization Results on a Cardiac Wave Propagation Application

Optimizing *brdr2d* on the Sandy Bridge Processor and on a Phi coprocessor show the advantage of using a polyhedral framework to optimize for both execution time and energy.

4.4.1 Results on Sandy Bridge Processor

To better understand the optimization variants of *brdr2d* they were executed with four different input sizes. Figure 4.5 compares the best variant for each input size relative to the base-line OpenMP version. As shown in Table 4.1, the best optimization

Table 4.1: This table shows the best program optimization for different input size.

Input Size	*Best Optimizations*
256	maxfuse, 1×128 tiling size, parallelization
512	maxfuse, 1×256 tiling size, parallelization
1024	maxfuse, 1×256 tiling size, parallelization
2048	maxfuse, 1×256 tiling size, parallelization

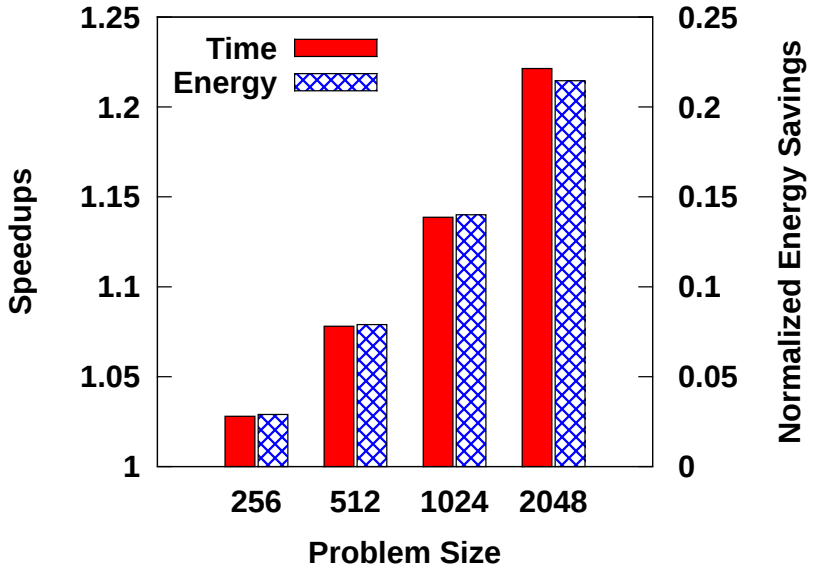


Figure 4.5: Graph showing the performance improvement and energy savings of the optimal program variant over the baseline OpenMP implementation for different problem size on Sandy Bridge Processor.

was different as the input size grew. For the problem size of 256, a tile size of 1×128 resulted in the fastest execution, For the larger input sizes,the variant with tile size 1×256 was fastest. As we increased the input size the optimized variants' relative performance and energy consumption improved (256 - 2.5% to 2048 - 21%). As the loop size increases, the loop nest becomes a more dominant portion of the execution and the relative performance from optimization improves. For the smaller input sizes, the data fit into various cache levels and the benefits of loop optimizations for data locality are ineffective.

4.4.2 Results on Intel Xeon Phi

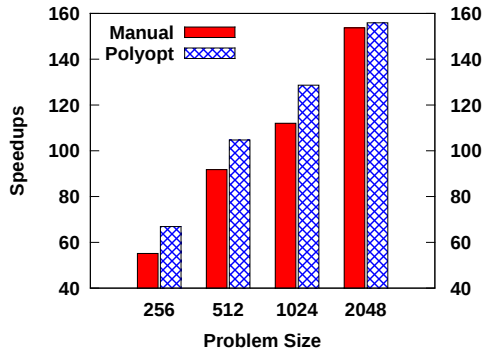
To show the benefits of using polyhedral optimization techniques on the Phi accelerator card, the performance of a manual OpenMP implementation can be compared with the best PolyOpt/PoCC generated OpenMP program variant (shown in Figure 4.6(a)). The speedups were calculated against a sequential Sandy Bridge execution.

The best PoCC variant of *brdr2d*, is over 20% faster than the manual OpenMP version written for the Xeon Phi card for small sizes. For the largest input size, 2048, the best PolyOpt/PoCC variant is still slightly faster than the manual OpenMP implementation and has an absolute speed up of over 150×. The optimal tiling size changes as the input grows. In each case, $1 \times size$ is preferred for maximum vectorization. As the problem size grows, non-tiled vectorization improves, reducing the effectiveness of tiling. This shows that manually choosing the right optimization combination is complicated and requires deep knowledge of the algorithm, the input, and the architecture. As expected the two main performance drivers for the Phi are parallelization, for threads, and vectorization, within threads.

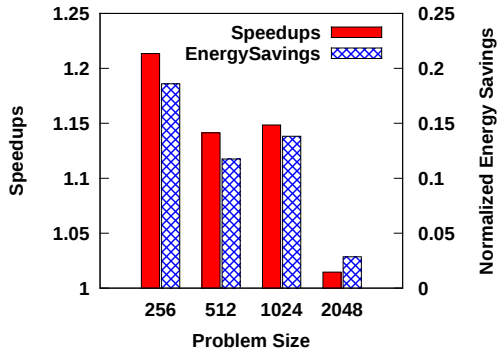
The polyhedral optimizations also improves energy. Figure 4.6(b) shows the *relative* speedups and the normalized energy savings offered by the polyhedral transformations and auto-parallelization. The energy savings approximately match the relative speedups, ranging from 20% down to 3% as size increases (and baseline vectorization improves).

4.5 Predicting the Optimization for Lowest Energy

We have shown that the optimization that leads to the minimum execution time also leads to the minimum energy in the polyhedral optimization space. Existing models that accurately predict the best compiler optimization sequence for execution time should also work for energy consumption. In this section, we provide the results of applying the model to autotune applications for energy.



(a) Graph showing the comparison between speedups of manual OpenMP implementation and the best PolyOpt/PoCC generated OpenMP program variant over the sequential implementation on MIC architecture.



(b) Graph showing the performance improvement and energy savings of the optimal PolyOpt/PoCC generated program variant over the baseline OpenMP implementation for different problem size on MIC architecture.

Figure 4.6: Polyhedral Optimization Results on MIC architecture

4.5.1 Energy Prediction Model Construction

Following the workflow shown in Figure 4.1, the execution time, power, and energy consumption were collected for all benchmarks from Polybench v4.0, including 11,192 polyhedral optimized versions. For each benchmark, the number of different combinations of the polyhedral optimizations applied are listed in Table 4.2. Different polyhedral combinations can result in the same code being generated. Table 4.2 also lists the number of unique program variants for each benchmark.

The control flow graphs were then extracted using the LLVM compiler from the original 29 benchmarks, resulting in 29 graphs. These graphs are represented such that the node labels contains the histograms of instruction counts in the basic blocks of the code. Similar to Park et al., [69], each node in the control flow graph contains the total number of instructions, the number of add/sub/mul/div instructions, the number of load/store instructions, the number of comparisons, and the number of conditional and unconditonal branches. The optimization sequences are encoded into fix-length

Table 4.2: This table shows the number of loop nests, the number of polyhedral combinations applied, and the unique number of variants for different benchmarks.

Benchmarks	*Loop Nests Level*	*Polyhedral Combinations*	*Unique Number of Variants*
2mm	3	520	374
3mm	3	520	375
adi	2	136	34
atax	2	136	48
bicg	2	136	63
cholesky	3	520	198
correlation	3	520	269
covariance	3	520	373
deriche	2	136	9
doitgen	4	2056	992
durbin	2	136	8
fdtd-2d	2	136	50
floyd-warshall	3	520	19
gemm	3	520	249
gemver	2	136	79
gesummv	2	136	62
gramschmidt	3	520	53
heat-3d	3	520	335
jacobi-1d	1	40	8
jacobi-2d	2	136	50
ludcmp	3	520	9
lu	3	520	65
mvt	2	136	93
seidel-2d	2	136	16
symm	3	520	9
syr2k	3	520	249
syrk	3	520	249
trisolv	3	520	40
trmm	3	520	157
Total	-	11,192	4535

feature vectors as well. Loop fusion, loop tiling, and loop vectorization are included in the feature vectors. For all polyhedral optimized versions, loop parallelization is applied and we omitted loop unrolling search space in order to avoid exceeding the memory of Matlab, which is used to construct the model. Four loop fusion related

optimizations and four loop tiling sizes are considered for each loop in a loop nest. We used three bits to distinguish between the three following fuse optimizations: "nofuse", "smartfuse", and "maxfuse". We used one bit to represent whether vectorization is applied. For loop tiling, we used one bit to indicate whether to apply loop tiling. When loop tiling is applied, we used four groups of three bits (total 12 bits) to represent at each loop nest level (up to four) what tiling size should be used (three bits represent four tiling sizes) at each loop nest.

The construction of the machine learning model relies on the similarities between each pair of training instances (program variants). In order to get this big similarity matrix (11,192 by 11,192), we derive it from the control flow graph similarity and optimization sequence similarity. The pair-wise control-flow graph similarities are calculated using the shortest path graph kernel which takes into account the node label similarities (i.e. instruction histograms). The optimization sequence similarities are calculated using XNOR operation. XNOR counts the number of common bits (corresponding to whether the particular optimization is applied or not). To derive each entry of the similarity matrix, the following formula is used:

$$k < (p_i, o_m), (p_j, o_n) >= k < p_i, p_j > \times k < o_m, o_n >$$

In the above equation, k refers to the similarity function that takes two paraemters of the same type as input. p_i and p_j refer to program variants while o_m and o_n refer to the optimization sequences. The number of unique tuples (p_i, o_m) is equal to the total number of program variants (11,192). This matrix is important in that it is fed into a SVM machine learning algorithm to construct a predictive model. How we derive the matrix directly affects the model generated.

In setting up the experiments, the model is always trained from 28 benchmarks, leaving the remanining one out for testing, i.e. we perform leave one out cross validation. We repeat the training and testing process for each benchmark, leaving it out and training on the rest. We report the results of top1 and top5 predictions. Top1 refers to the optimization sequence that is predicted to consume the least amount of energy. Correspondingly, in the top5 case, five optimization sequences that are predicted to

consume the least amount of energy are chosen. The selection of optimizations is based on the predicted energy consumption. We evaluate the prediction results of the best (top1) or five best (top5) optimizations by investigating how close the predicted best optimization combinations come to the optimal combination.

4.5.2 Prediction Results

Table 4.3 shows the energy prediction results for each benchmark. In this table, energy savings are calculated by comparing the energy consumption of the compiler transformed program variants to the energy consumption of the original sequential program. The percentage is calculated by dividing the predicted energy savings over the optimal energy savings. This value means the achievable energy improvement in our explored optimization space. For example, the best transformed variant from the top5 predicted compiler transformations reduces energy by 4.8× while the optimal compiler transformation reduces energy by 5.9×. So, the percentage of optimal energy improvement achieved from the top5 predicted compiler transformations in our explored optimization space is given by 4.8/5.9. The top five predicted optimization sequence achieved 82.98% of the optimal energy savings (× reduction in energy consumption) on average. With the 1-shot model, the predicited best optimization sequence achieved 74% of the optimal achievable energy savings.

The results of predicting the minimum power variant are presented in Table 4.4. Here the optimal and predicted power are relative to the sequential version's power consumption. Since the polyhedral optimizations we study here include parallelization, the power consumption of all transformed program variants is found to be larger than that of the sequential version. The model will predict the best *parallelized* version that consumes the least amount of power. In Table 4.4, if the power increase has an optimal/minimal value (or predicted value) of 1, it means the power is identical to the sequential version's power. If the value of power increase is 2.5, it means the best program variant's consumed power is 2.5× more than the consumed power of the sequential version. The further the predicted power increase is from the minimal power

Table 4.3: This table shows the results of the model when predicting the energy savings of compiler transformations. The "percentage" corresponds to the achievable energy savings given by the model compared to the best energy savings.

Benchmarks	*Optimal Energy Savings*	*Predicted Energy Savings (Top5)*	*Percentage (Top5)*	*Predicted Energy Savings (Top1)*	*Percentage (Top1)*
2mm	5.9×	4.8×	81.36%	4.1×	69.50%
3mm	5.9×	4.5×	76.27%	4.5×	76.27%
adi	6.2×	6.2×	100.00%	0.5×	8.06%
atax	1.5×	1.4×	93.33%	1.3×	86.67%
bicg	2.5×	2.5×	100.00%	2.5×	100.00%
cholesky	6.6×	5.8×	87.88%	5.8×	87.88%
correlation	31.0×	25.4×	81.94%	24.5×	79.03%
covariance	46.8×	24.1×	51.50%	24.1×	51.50%
deriche	0.9×	0.6×	66.67%	0.6×	66.67%
doitgen	7.3×	2.0×	27.40%	2.0×	27.40%
durbin	1.0×	1.0×	100.00%	1.0×	100.00%
fdtd-2d	1.1×	0.9×	81.82%	0.8×	72.73%
floyd-warhsall	3.1×	3.0×	96.77%	3.0×	96.77%
gemm	6.5×	5.5×	84.62%	3.3×	50.77%
gemver	4.5×	4.5×	100.00%	3.4×	75.56%
gesummv	2.9×	2.9×	100.00%	2.7×	93.10%
gramschmidt	9.5×	0.7×	7.36%	0.7×	7.36%
heat-3d	1.3×	1.3×	100.00%	1.2×	92.31%
jacobi-1d	1.5×	1.5×	100.00%	1.5×	100.00%
jacobi-2d	1.0×	1.0×	100.00%	1.0×	100.00%
ludcmp	0.9×	0.9×	100.00%	0.9×	100.00%
lu	21.4×	19.3×	90.19%	17.2×	80.37%
mvt	5.1×	4.2×	82.35%	3.2×	62.75%
seidel-2d	4.5×	4.2×	93.33%	3.9×	86.67%
symm	2.0×	1.2×	60.00%	1.2×	60.00%
syr2k	3.7×	3.6×	97.30%	3.2×	86.49%
syrk	22.2×	12.9×	58.11%	10.1×	45.50%
trisolv	1.7×	1.5×	88.24%	1.4×	82.35%
trmm	41.2×	41.1×	100.00%	41.1×	100.00%
Average	8.61×	6.50×	82.98%	5.89×	73.99%

increase, the lower the percentage of power achieved by the model, compared to the power consumed by the best transformed version of the program in the explored space.

For example, the best variant of the covariance benchmark has a power consumption increase of 2, which means the best program transformation (with parallelization) increased power by 2×. The predicted power increased by 2.5, which means that the power consumed by the compiler transformation that is given by our trained model increased power by 2.5×, compared to the sequential version. The percentage is calculated by dividing the minimal power increase (i.e., 2×) by the predicted power increase (i.e., 2.5×). Therefore, our model achieves 80% of the achievable minimal power consumption as compared to the best transformed variant. The table shows that the model achieved 77.03% of the achievable minimal power consumption in 5-shot and 75.31% in 1-shot prediction on average.

However, our model is not as effective at predicting EDP. We discovered this by training our model using an application's EDP and only achieved 65% out of the optimal EDP on average. We believe this is due to the combined effect of mispredicting both energy and time. The combined effect of miss predicting energy and miss predicting execution time makes combining these predictions into an Energy-Delay Product prediction less reliable.

4.6 Discussion

When OpenMP appliations run on modern parallel architectures like the Intel SandyBridge processor, run-to-run variance is bound to have an impact on both performance tuning and energy tuning when using constructed predicting models that don't take this variance into account. Runtime variance can be seen as noise in the data used for training a model. Such runtime variability can come from hardware or software. Each processor chip is manufactured under slightly different conditions. Some processors are more efficient than others. An efficient chip may run over-clocked more of the time than a non-efficient chip. Location on a board or within a system, may affect the amount of cooling a chip receives which also impacts the clock frequency. Where the OS schedules an application can have significant impact on performance. Thread binding to cores impacts memory access latency. Remote memory access significantly

Table 4.4: This table shows the results of the model when predicting the power consumption *increase* (compared to the sequential version's power) of polyhedral transformed code. The closer the predicted power increase is to the minimal/optimal power increase, the better the transformed variant was.

Benchmarks	*Minimal Power Increase*	*Predicted Power Increase (Top5)*	*Percentage (Top5)*	*Predicted Power Increase (Top1)*	*Percentage (Top1)*
2mm	2.5×	2.5×	100%	2.5×	100%
3mm	2.5×	2.5×	100%	3.33×	75%
adi	2.5×	2.5×	100%	2.5×	100%
atax	1×	2.5×	40%	2.5×	40%
bicg	1.67×	2.5×	66.67%	2.5×	66.67%
cholesky	1×	2.5×	40%	2.5×	40%
correlation	1×	2.5×	40%	2.5×	40%
covariance	2×	2.5×	80%	2.5×	80%
deriche	1×	1×	100%	1×	100%
doitgen	1.67×	2×	83.33%	2×	83.33%
durbin	1×	1×	100%	1×	100%
fdtd-2d	1×	2.5×	40%	2.5×	40%
floyd-warshall	1×	1×	100%	1×	100%
gemm	2.5×	3.33×	75%	3.33×	75%
gemver	1×	2.5×	40%	2.5×	40%
gesummv	2.5×	2.5×	100%	2.5×	100%
gramschmidt	2.5×	3.33×	75%	3.33×	75%
heat-3d	2.5×	2.5×	100%	2.5×	100%
jacobi-1d	1.11×	1.11×	100%	1.11×	100%
jacobi-2d	1×	2.5×	40%	2.5×	40%
lu	1×	1.11×	90%	1.11×	90%
ludcmp	2.5×	2.5×	100%	2.5×	100%
mvt	2.5×	2.5×	100%	2.5×	100%
seidel-2d	1.11×	2.5×	44.44%	2.5×	44.44%
symm	1.11×	1.11×	100%	1.11×	100%
syr2k	2.5×	3.33×	75%	3.33×	75%
syrk	2×	3.33×	60%	3.33×	60%
trisolv	1.11×	2.5×	44.44%	2.5×	44.44%
trmm	2.5×	2.5×	100%	3.33×	75%
Average	1.7×	2.3×	77.03%	2.36×	75.31%

increases memory latency and possibly network congestion. The OS scheduling where

Table 4.5: Jacobi-2D Autotuning execution times

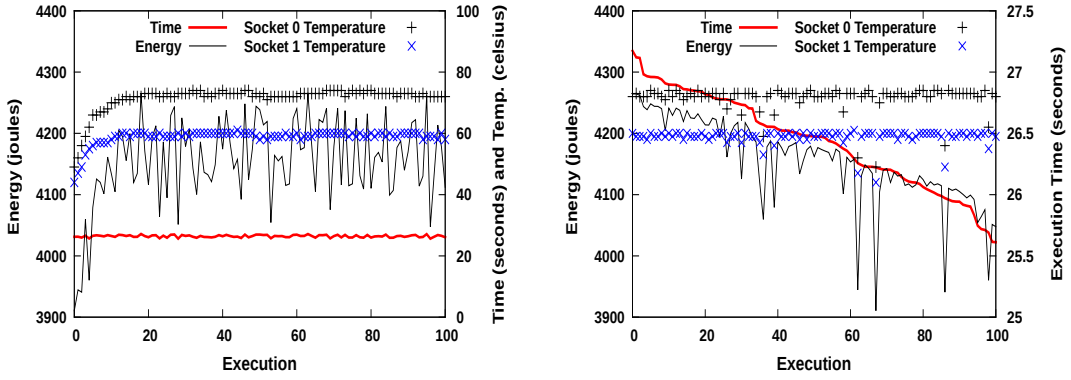
Run Number	nofuse	nofuse1X16	nofuse1X32	nofuse1X64
1	5.33	5.15	**5.09**	5.30
2	5.14	**5.08**	5.13	5.14
3	5.37	5.35	**5.20**	5.31
4	**5.04**	5.12	5.18	5.34
5	5.41	5.33	5.16	**5.08**
6	5.24	5.22	5.37	**5.02**
7	**5.06**	5.16	5.20	5.08
Average	5.22	5.20	5.19	**5.18**

an application runs and what applications are running simultaneously impact performance. The existence of run-to-run variance means that the "best" program version predicted by the model is at best one of the "best" versions [73]. Table 4.5 presents execution times for each of 7 runs of the fastest 4 program variants for Jacobi-2D polybench on the two socket 2.7 GHz E5-2680 SandyBridge system. For each run, the fastest program variants is in boldface. If only one run is used, any of the four transformed code versions could be picked as "best". When each experiment is run 7 times, on average the *nofuse1X64* set is almost 1% faster than the *nofuse* set. In preparing training data for our prediction framework, the experiments were repeated at least five times to mitigate the run-to-run variance.

Running a single parallel application repeatedly on the same Intel Sandy Bridge system shows socket temperature's effect on the execution time and the energy consumption. It also shows the execution and energy consumption differences of consecutive runs of an application. Figure 4.7(a) has 4 values plotted for each of the 100 runs of the original LULESH OpenMP application. The execution time is the thick red line on the bottom (scale on the right). It is around 26 seconds. The jagged black line (mostly in the middle) is the energy consumed by each run (scale on the left). Each run has a variation between consecutive runs of almost 100 Joules (or about 2%). More interesting are the temperature values (socket 0 - black plus signs at the top; Socket 1 -

blue crosses slightly lower), which start off at 46°C and 42°C respectively and over the first 6-12 runs, rise to respective steady-states of around 72°C and 59°C. The energy used by the early lower-temperature runs is significantly less than that of the later runs (between 200 and 300 Joules – 4-6%). By sorting the data by execution time, we get Figure 4.7(b). Note that the graphs have different scales to exaggerate the time differences – temperature is on the same scale in both graphs. A strong relationship between energy usage and execution time is evident, in Figure 4.7(b). Execution time varies between 27 and 25.5 seconds (6% variation) while energy generally falls from 4250 to 4050 joules(5% variation). Energy consumption generally tracks execution time. Outliers exist for runs when temperature had not reached the final steady state. There are also minor fluctuations between runs with the same time/temperature, which may be related to minor temperature differences, energy measurement inaccuracies, or other hard to measure differences. The time to reach steady-state temperature is about 5 minutes, with the bulk of the temperature rise in the first one or two minutes. Temperature has a very clear impact on the amount of energy used by an application. The lower the temperature, the less energy (and power) is used. When the system is idle, the temperature difference between the two sockets is about 4°C, but as the system becomes active, the difference is 13°C. The test system is a half-height blade, with the two processors in line. The air flows over one socket before reaching the second socket. The warmer air doesn't allow the second core to cool as much and it runs noticeably hotter. Also, LULESH has some known execution variabilities due to memory allocation, which can explain the 6% variation in execution time over the 100 runs. For a constant temperature, the power level and energy consumption are well correlated with time. Approximately 1% jitter exists between runs that are at the same temperature. To minimize the temperature's effect on the execution time, we pre-warmed up the machine everytime a set of benchmarks were run by running a compute-intensive program.

In addition to collecting the execution time, power, energy, and temperature for the entire application, we collected this information for various loops of the application.



(a) in execution order

(b) sorted by time

Figure 4.7: Time/Energy/Temperature for 100 runs of LULESH

Region	Mean Power	Region	Mean. Power
3	152.84	6	163.51
7	174.70	9	149.61
13	144.91	18	160.07

Table 4.6: Power for selected regions of LULESH

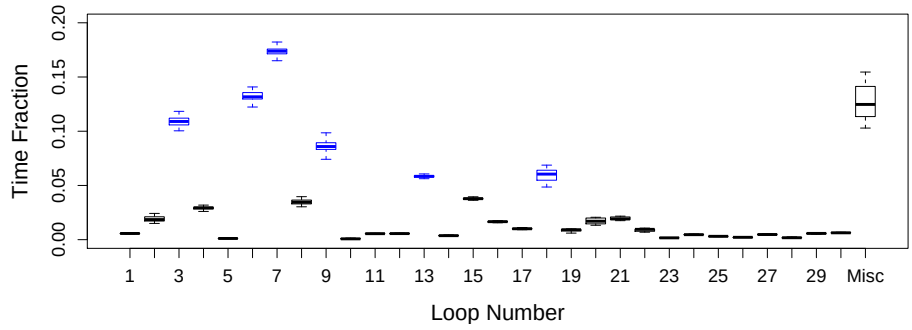
We observed vast differences between the measured power consumption of the loops. Figure 4.8(a) shows the amount of computation time spent in each of the 30 parallel regions and in serial regions of LULESH application. The percentages are calculated for all 100 runs. A box graph is used to show the repeatability of the execution time results and the variance in the power calculations for the same regions. Six loops each accounted for greater than 5% of the execution time and the run-to-run variance of these regions was small. Attempting to measure power at this level introduces sampling errors. For the Intel SandyBridge, loops less than one millisecond do not result in consistent values. Figure 4.8(b) highlights the errors with reported power usage ranges from almost 600W down to 0W. Note power usage varies tremendously between the various runs. All of the outliers (high and low) are the result of sampling

errors on loops that execute for short amounts of time, i.e., the execution time of these loops is around the same order of the time to do an MSR update. Intel states in their documentation that the energy counter should be sampled every millisecond or longer. For the six regions (in blue) that exceed 5% of the total computation, the results are consistent and the uncertainty boxes are small. In practice, the energy MSR is updated sporadically and some time between observations is needed. Looking at only the average power for the six most significant regions in Table 4.6, one finds significant variation between the individual parallel regions. Region 13 uses 17% less power than region 7. The amount of work being done can affect the power usage significantly. In order to get accurate power and energy readings for loops such as those in polybench kernels, we made sure that the dataset chosen was large enough.

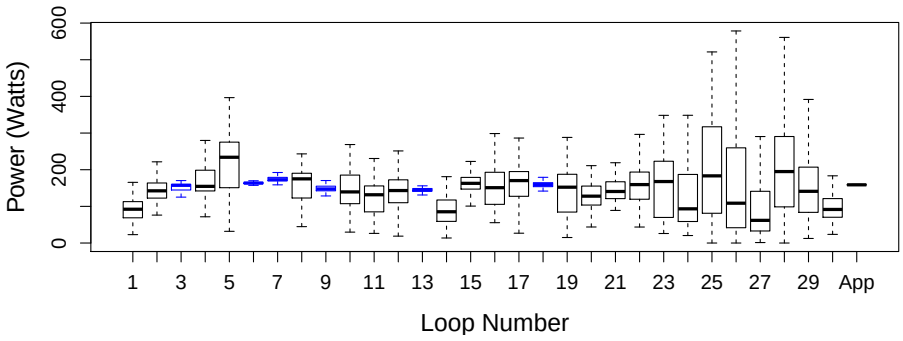
In summary, the existence of variance makes it difficult for machine learning models to accurately learn from data. The temperature has a clear impact on both the execution time and the power consumption of an application. Actions such as pre-warming the machine before timing and power runs need to be taken to mitigate the temperature's effect. Individual loops within an application have vastly different execution time and power cosumption. Power and energy measurement should be carried out on loops that run long enough to minimize measurement error.

4.7 Summary

In this chapter, we studied the correlation of energy consumption and execution time of transformed variants of a program using an autotuning framework that utilizes a polyhedral compiler to generate different versions of a set of programs. We observed that, without considering power management features like DVFS and CPU clock modulation, the execution time can be a good indicator for relative energy consumption. Although in most auto-tuning cases (without considering DVFS and/or CPU clock modulation) the minimum execution leads to minimum energy consumption, similar conclusions cannot be drawn for power consumption. In this chapter, we extended an existing execution time prediction model for the prediction of both power and energy.



(a) Execution time fraction for LULESH OpenMP parallel regions and misc. sequential regions.



(b) Power for LULESH OpenMP parallel regions and the entire application.

Figure 4.8: Execution time and power consumption of LULESH parallel regions

The energy prediction model is necessary for the prediction of energy consumption when power management techniques are considered. Both the power and energy prediction model achieved more than 78% of the achievable minimal power and energy consumption.

Chapter 5

OPTIMIZING OPENMP APPLICATIONS FOR ENERGY EFFICIENCY USING CPU CLOCK MODULATION

We now turn to the goal of exploring whether and how energy and energy-delay product (EDP) optimization of applications can be realized with power management techniques available on modern processors (e.g. Intel Sandy Bridge). In particular, we study CPU clock modulation, a less well-studied power saving technique, that has advantages over DVFS. The potential profit from CPU clock modulation needs to be determined for each loop within an application, as it could lead to significant energy benefit with little to no degradation in performance. We developed a framework that can support the exploration of energy measurement/control as well as performance monitoring. We utilize all the functionality of this framework to discover places CPU clock modulation could offer energy savings. This framework allowed us to find the appropriate clock modulation settings for individual loops to reduce overall application energy usage. In summary, this chapter explores hardware techniques to lower energy consumption of applications and improve energy efficiency at the same time.

5.1 Motivation

High-performance computing (HPC) applications consist of multiple parallel loops of different types, such as compute-intensive or memory-intensive loops. Each type of loop achieves the best energy efficiency (e.g. smallest EDP) at a different operating frequency. We use Energy-Delay Product (EDP) throughout this chapter as a metric for energy efficiency. The lower the EDP, the better the energy efficiency. EDP takes into consideration energy consumption and execution time, that is, power management techniques are expected to lower application energy consumption without

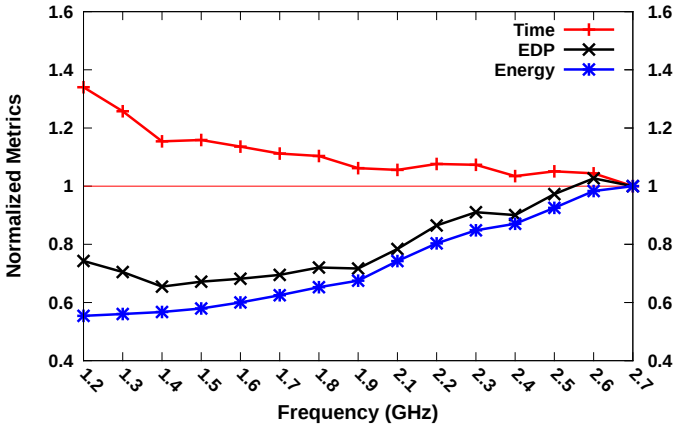


Figure 5.1: This figure shows the energy, Time, and EDP of a memory-intensive `LULESH` loop when varying frequency using DVFS. Values normalized to 2.7GHz.

much increase of the execution time. As a motivating example, two parallel loops from LULESH application are shown in Figures 5.1 and 5.2.

Figure 5.1 shows the normalized energy consumption, execution time, and EDP of a memory access intensive `LULESH` loop with changing frequencies achieved using DVFS. For a loop with high memory access density and limited computation, if frequency is reduced, the power required falls faster than the execution time increase. This results in a lower EDP. The decrease continues until very low frequency settings beyond which the execution time increases significantly. The minimum EDP occurs at 1.4GHz, with 43.2% energy savings and 15.4% time increase. This LULESH loop indirectly accesses arrays, resulting in a non-contiguous array access pattern and frequent cache misses. The CPUs are idle waiting for data, allowing the power to be reduced without large performance loss. For energy efficiency, this loop can run at reduced CPU clock frequency.

Figure 5.2 shows the normalized execution time, energy consumption, and EDP of a compute-intensive `LULESH` loop. Reducing frequency has a significant impact on the execution time of this loop. While energy consumption is marginally reduced (because of power/frequency decrease), the EDP rises because of the substantial performance

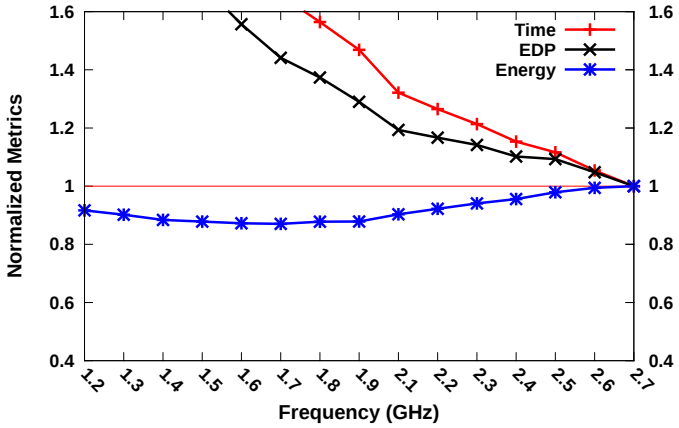


Figure 5.2: This figure shows the energy, Time, and EDP of a compute-intensive `LULESH` loop when varying frequency using DVFS. Values normalized to 2.7GHz.

degradation. The execution time increases proportionally to the decrease in the effective frequency. The lowest EDP occurs at the maximum clock rate suggesting that the loop should run at full speed.

Figures 5.1 and 5.2 show that for high memory access density loops, running at a significantly reduced frequency setting achieve both optimal energy and energy-delay product (EDP). But, for compute-intensive loops, running at lower frequencies leads to poor energy efficiency due to increases in execution time. LULESH contains two memory-intensive loops, taking up more than 25% of the total execution time, and two compute-intensive loops, taking up more than 40% of the total execution time. Hence, executing LULESH with a fixed frequency will lead to non-optimal energy efficiency. Benefits can be maximized by setting the frequency that is best for each individual loop.

5.2 Approaches

In order to change frequencies correctly for different parallel loops to improve energy efficiency, loops from various benchmarks were characterized. The characterization helped find energy saving opportunities where frequency control could be applied

with little to no negative impact. We chose CPU clock modualtion to drive the frequency change after comparing with DVFS for fine-granularity energy control. Before CPU clock modulation could be applied to appropriate loops, the impact of changing the frequency via CPU clock modulation was studied first for various loops.

5.2.1 Loop Characterization

For this research, we developed low-overhead clock modulation power saving techniques and easy-to-use energy measurement and control APIs to explore when clock modulation saves energy. Parallel loops are characterized to determine the clock modulation setting that has the best energy/delay trade-off. Loop characterization is facilitated by RCRtool's capability of monitoring the uncore subsystem. The uncore system on the Intel Sandy Bridge architecure is responsible for managing cache coherency, controlling QPI memory traffic between different sockets, and fulfilling the memory request of cores within a socket [38]. The uncore subsystem offers a set of counter registers that record all memory requests from cores to the Last Level Cache (LLC). Memory transaction monitoring is supported using 44-bit wide counters (Cn_MSR_PMON_CTR{3:0}). Last-Level-Cache (LLC) activity is used to separate loops into memory-bound/compute-bound and balanced. Memory Access Density (MAD) is computed by examining TOR_OCCUPANCY memory counter updates during loop execution. This counter collects statistics on how many memory requests by all cores have to be serviced via the L3 cache. These memory requests are put into the Table of Requests (TOR). The MAD metric essentially records how fast the TOR_OCCUPANCY counter grows. Higher values usually result from increased cache misses/memory references. For example, if the MAD metric is 50, it means 50 memory requests per-cycle from all cores are put into the Table of Requests. This metric provides insight into application's memory characteristics. Loops are then categorized using this memory access density metric. We also designed a runtime energy control program that monitors the memory access density while running an application. Depending on the measured memory access density metric, the machine frequency level

is changed accordingly via clock modulation.

5.2.2 Multi-Frequency Execution For Energy Optimization

In this dissertation, multi-frequency execution refers to executing applications with different clock frequencies for each parallel loop. The frequency is lowered when entering memory intensive loops and reset exiting. Reducing the CPU frequency when executing loops with instruction stalls due to memory latency can reduce power consumption of those loops while not not significantly impacting total execution time.

Fine-granularity Energy Control

To support multi-frequency execution of applications (with different Duty Cycle Modulation setting), an API that invokes writes to IA32_CLOCK_MODULAION MSR is needed. This is achieved by setting the MSR for each core using root privilege. In HPC environment, applications rarely get root access. To get around that, we modified the Linux kernel to add a system call that supports the setting of values in Table 2.1. Each setting is accomplished by using inline assembly code that executes the wrmsr privileged instruction[1]. With this modification, any application can perform a frequency change via Duty Cycle Modulation. Although this poses security issues, this technique is only used for our proof of concept and a more secure technique could be developed. The two above approaches (system call vs. root writing msr device files) are equivalent in terms of transition latency, which is mostly related to user/kernel context switches for both system call and kernel driver approach. In order to compare CPU clock modulation based per-loop power management with DVFS, the DVFS set_frequency and reset_frequency API calls were placed around memory intensive loops in LULESH. Inside the API, the file /sys/devices/system/cpu/cpu*/cpufreq/scaling_cur_freq is modified to contain the target frequency. The Linux ACPI driver takes care of the

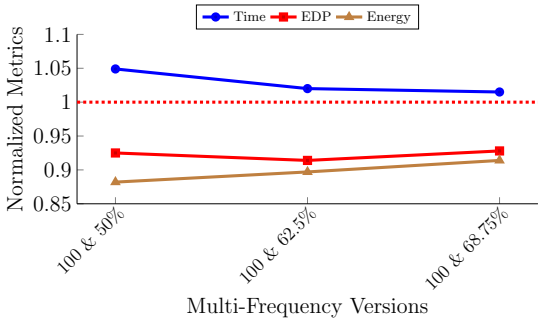
[1] Privileged instructions are enforced by hardware. Applications running in Ring 3 are prohibited from executing such instructions even with root privilege. Only the operating system (which runs in Ring 0) can execute such instructions.

actual frequency transition. For CPU clock modulation, we developed API calls that change machine frequency by setting appropriate values to msr files located under the /dev/cpu directory.

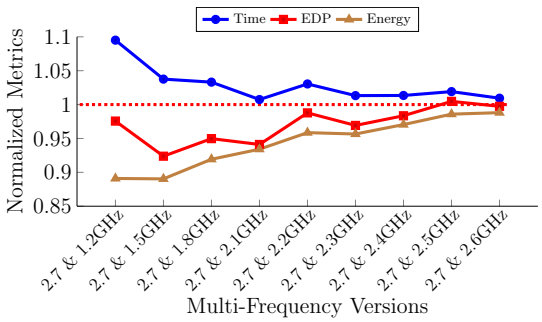
DVFS and CPU Clock Modulation Comparison

Oftentimes DVFS is considered to achieve better application energy efficiency. We show that fine-grained energy control using CPU clock modulation achieves equivalent or better energy efficiency compared to DVFS when applied to the LULESH application.

Figure 5.3 shows the LULESH application's normalized execution time, energy, and EDP obtained by executing the memory intensive loops with clock modulation settings varying from 50% to 68.75% (Figure 5.3(a)) and frequencies varying from 1.2GHz to 2.6GHz (Figure 5.3(b)). The compute-intensive region is kept at 100% for CPU clock modulation and 2.7GHz for DVFS. The baseline was executing at fixed 2.7GHz throughout without setting CPU clock modulation.



(a) This figure shows the multi-frequency execution using clock modulation. Clock modulations are changed upon memory intensive loop entry and exit.



(b) This figure shows the multi-frequency execution using DVFS. DVFS frequencies are changed upon memory intensive loop entry and exit.

Figure 5.3: This figure shows normalized metrics of multi-frequency execution of LULESH application over the default single frequency execution. 100% and 2.7GHz are the default single-frequency execution for clock modulation and DVFS, respectively.

From Figure 5.3 we can see that CPU clock modulation could achieve the same or even better EDP than DVFS. Although DVFS is known for its quadratic potential of saving energy, CPU clock modulation is seen to outperform DVFS by just changing the frequency linearly with finer-granularity of energy control. For example, as shown in Figure 5.3(a), running LULESH with 100% and 62.5% clock modulation (2.7GHz and 1.68GHz) results in 8.6% EDP savings with only 2% performance slowdown. Running LULESH with 2.7GHz and 1.8GHz DVFS frequency only improved EDP by 5%, but incurred 3.3% performance slowdown, as shown in Figure 5.3(b). In LULESH, the compute-intensive loops directly follow the memory intensive loops. For this situation, fine-grained energy control is needed because it transitions the system to a low power state quickly upon entering the memory intensive loops and to a default power state when exiting. CPU clock modulation is able to put the system into low power mode and to reset with little delay, allowing the loops to execute at appropriate clock frequencies.

5.3 Benchmarks and Experimental Setup

To study the effectiveness of clock modulation at reducing energy consumption, a variety of OpenMP benchmarks were used. Some of the programs consist of single loops, Polybench OpenMP. Two benchmarks simulate realistic applications, `LULESH` and `miniFE`. Another benchmark suite, NAS Parallel Benchmark, and a realistic application, `brdr2d`, the cardiac wave propagation simulation application, were also tested.

LULESH

Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (`LULESH`) [42] is chosen and its OpenMP implementation (v1.0), best for the Intel Sandy Bridge architecture, was used for this evaluation [52]. This OpenMP implementation has 12 parallel loops, resulting from application of loop fusion to the original OpenMP implementation. Half of the 12 parallel regions take up more than 90% of the total application time, each of them taking up approximately 10% to 25% of the application's

execution time. The remaining loops take up less than 10% in total, and none of them take up more than 5% of the total application time.

miniFE

The Mantevo Suite Release 2.0 [53] contains eight miniapps representing a variety of real applications. Only three of the miniapps have OpenMP versions. Of these, we choose `miniFE`, which implements algorithms from an implicit finite-element application. It assembles a sparse linear system from the steady-state conduction equation and solves the linear system using un-preconditioned conjugate-gradient algorithm. `miniFE` contains a variety of parallel loops mimicking the structure of a large finite-element application.

Polybench

For the research in this chapter, we used the Polybench programs for energy behavior characterization. However, the polyhedral optimization space was not studied. Parallelized OpenMP versions [14] of these programs were used for evaluation.

`brdr2d`

We also studied `brdr2d`, an open-source 2D monodomain cardiac wave propagation simulation used in Chapter 3 and 4. This application was parallelized and optimized using various programming languages and techniques including OpenMP, OpenACC, and a polyhedral compiler. In this chapter, the OpenMP version was evaluated. The application has an outer while loop that typically executes an inner parallel loop nest thousands of times. The inner loop nest accounts for more than 90% of the total execution time.

NAS Parallel Benchmark

The NAS Parallel Benchmarks (NPB) are a set of programs for benchmarking the performance of parallel supercomputers. The benchmarks consist of several kernels and pesudo applications. We used SNU NPB Suite, which is based on NPB3.3 version.

The SNU NPB benchmark suite contains OpenMP implementations of 10 benchmarks [88]. We chose input sizes from the two largest SNU NPB input classes (C & D) for evaluation so that the benchmarks could execute long enough to produce stable execution time and power consumption readings.

Experimental Setup

All tests were performed on an M620 Dell Blade with 2 Intel Xeon E5-2680 (Sandybridge) CPUs containing eight cores each and 64 GB of main memory with hyper-threading disabled. The default clock speed is 2.7GHz. The cache size is 20MB per socket (40MB total). The blade runs 2.6.32-431 version of the Linux kernel that supports MSR accesses. Energy measurements tend to be sensitive to temperature [74]. To mitigate temperature effects, pre-measurement runs warmed the system up and all tests were run consecutively. Each test was repeated at least five times. The Intel ICC compiler (v14.0.2) was used unless otherwise stated, and the O3 optimization flag was turned on for each test.

5.4 Results

In this section, we report our loop characterization results on whether loops are more memory intensive or compute-intensive. These results also show how the energy efficiency of the loops are impacted by changing the effective CPU frequency using the clock modulation technique.

5.4.1 Application Power Characterization Results

The understanding of energy and performance characteristics of loops across different benchmarks and within applications is critical to optimization. Using the energy measurement tool, RCRtool, the energy and performance of loops with different CPU clock modulation settings were measured. The impact to the energy and the energy-delay product (EDP) of different loops exhibits three different types of loops. We name these types of loops Mem-Int, Comp-Int, and Balanced according to whether the loops are memory access intensive, compute intensive, or a balance of compute and

memory. Table 5.1 describes the characteristics of each loop type when changing the frequencies.

Table 5.1: This table describes the characteristics of three types of loops when varying the frequencies.

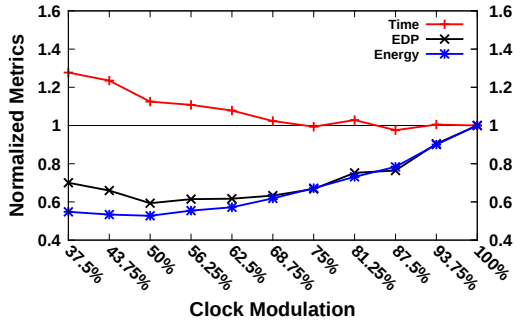
Loop Type	*Energy Characteristics*
Mem-Int	High memory access density. Running at lower frequencies does not increase execution time. Power is reduced and EDP is significantly lowered.
Comp-Int	Low memory access density. No slack exists and slowing frequency increases execution time. EDP increases for any reduced computation rate. Should be run at full frequency.
Balanced	Balanced memory access and computation. Such loops provide a little slack, i.e., lowering frequency slightly saves energy with relatively low performance degradation. Energy can be reclaimed from some computation bubbles during memory intense phases.

`LULESH` includes three of these types of loops. Various Polybench loops also fall into all three categories. The Mantevo application, `miniFE`, only contains Comp-Int and Balanced loops. The `brdr2d` application contains one dominant loop with balanced memory access and computation. Similar to the Polybench loops, kernels and pseudo applications in NPB fall into three categories.

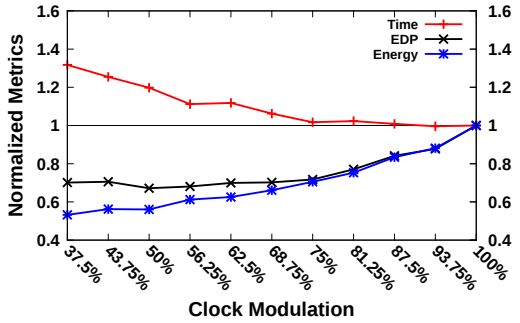
High Memory Access Density Loops

As we have seen in Figure 5.1, when reducing the frequency of a loop with high memory access density and limited computation, the power required falls faster than the execution time increases. This results in a reduced energy-delay product (EDP). The decrease continues until very low frequency settings where execution time increases significantly. In this section, we show how the EDP changes when the frequency is changed by setting the CPU clock modulation instead of setting DVFS frequencies.

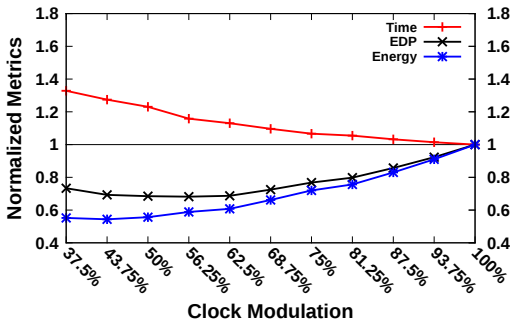
Figure 5.4(a) shows the normalized energy consumption, execution time, and EDP of one memory access intensive `LULESH` loop by varying the clock modulation. While lowering effective frequency generally leads to longer execution time, the loop



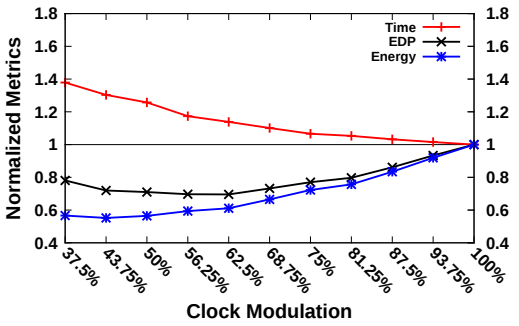
(a) This figure shows the energy, Time, and EDP of a high memory access density `LULESH` loop when varying frequency using clock modulation



(b) This figure shows the energy, Time, and EDP of a high memory access density NPB MG loop when varying frequency using clock modulation



(c) This figure shows the metrics of `fdtd-2d` Polybench when varying clock modulation



(d) This figure shows the metrics of `jacobi-2d` Polybench when varying clock modulation

Figure 5.4: Characteristics of high memory access density loops in `LULESH`, `MG` benchmark, `fdtd-2d` Polybench, and `jacobi-2d` Polybench are shown. The normalized metrics are energy, time, and energy-delay product (EDP). Lower is better. The baseline is 100% clock modulation setting.

with high memory access was affected only a little (2.3%) by a low frequency setting, e.g. clock modulation level of 68.75%, as shown in Figure 5.4(a). With the time being roughly the same, the power is reduced linearly changing the effective frequency from 100% to 62.5%. It went down from about 150 Watts to 85 Watts for two sockets. Therefore, the energy curves display a sharp decrease on the right half of Figure 5.4(a). The minimum EDP occurs at 50% clock modulation setting, with 47.3% energy savings

and 12.5% time increase. The EDP curve follows the energy curve closely until the time curve surges. This LULESH loop incurs high memory access traffic because the arrays are accessed indirectly. With such a non-contiguous array access pattern, cache misses frequently happen and the CPUs become idle, waiting for data. Power is greatly reduced without large performance loss.

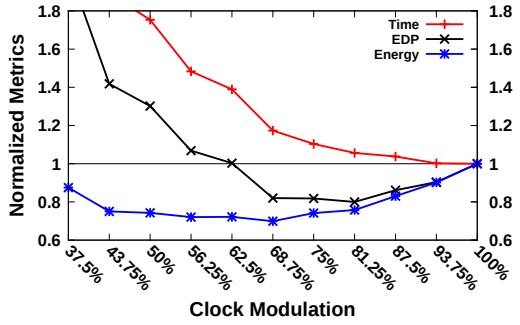
For the NPB MG benchmark, the curves in Figure 5.4(b) look very similar to Figure 5.4(a). The minimum EDP occurs at 50%, with 44% energy savings and 19.8% time increase.

Figures 5.4(c) and 5.4(d) depict similar graphs (Time, Energy, and EDP) for `fdtd-2d` Polybench and `jacobi-2d` Polybench, respectively, when varying the clock modulation setting from 37.5% to 100%. `fdtd-2d` saves 24.4% energy at a 81.25% frequency setting while only slowing down by 5.5%, resulting in a 20.2% reduction in EDP. The EDP minimum for `fdtd-2d` occurs at a 56.25% clock rate. It achieves 41.2% energy savings with a 15.8% execution time increase and a 31.8% EDP improvement. We save saves 24.3% energy for `jacobi-2d` at an 81.25% frequency setting while only slowing it down by 5.3%, resulting in a 20.3% reduction in EDP. `jacobi-2d` has a minimum EDP occuring with a 62.5% clock rate. It achieves 39% energy savings with a 13.8% execution time increase, for a 30.5% EDP improvement.

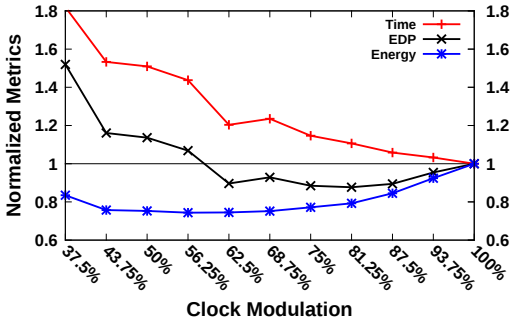
Figure 5.4 shows that for high memory access density loops, running at a significantly reduced power setting achieves both optimal energy and energy-delay product (EDP). LULESH contains two such loops, taking up more than 25% of the total execution time. By reducing frequency for indirect access (LULESH) and stencil operations (Polybench programs and MG benchmark) overall energy and EDP can be improved.

Loops with Balanced Memory Access and Computation

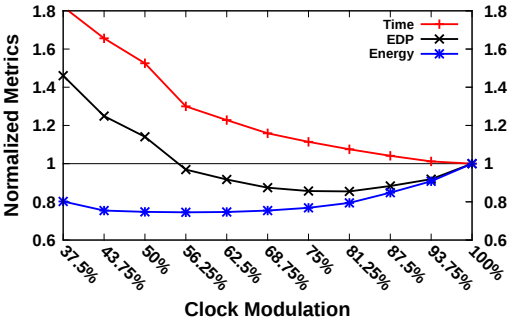
Compared to loops with high memory access density, the execution time of loops with balanced memory access and computation is affected slightly more when reducing the frequency. Since there is still a considerable amount of memory accesses in these loops and so the performance downgrade is not large enough to offset the reduction in



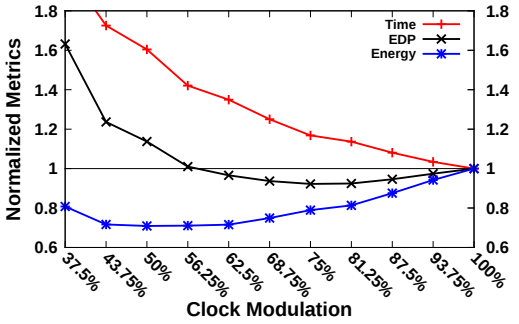
(a) This figure shows the metrics of a `LULESH` loop when varying clock modulation



(b) This figure shows the metrics of a `miniFE` loop when varying clock modulation



(c) This figure shows the metrics of `cholesky` Polybench when varying clock modulation



(d) This figure shows the metrics of `brdr2d` application when varying clock modulation

Figure 5.5: Loops with balanced memory access and computation in `LULESH`, `miniFE`, `cholesky` Polybench, and `brdr2d` realistic application are shown. The normalized metrics are energy, time, and EDP (lower is better). The baseline is 100% clock modulation setting.

power consumption. The best EDP for all programs is achieved between 68.75% and 81.25% clock skipping settings.

Figure 5.5 shows the normalized execution time, energy, and EDP of loops with balanced memory access and computation in `LULESH` (Figure 5.5(a)), `miniFE` (Figure 5.5(b)), `cholesky` from Polybench (Figure 5.5(c)), and `brdr2d` (Figure 5.5(d)). In all three benchmarks, the energy consumption and EDP were reduced when the effective frequency was changed from large to smaller values. `LULESH` achieved minimal

EDP at 68.75%, `miniFE` at 81.25%, `cholesky` at 75%, and `brdr2d` at 75%. When the effective frequency is set to very small values, the EDP curve begins to rise due to large execution time increases.

The balanced loop in `LULESH` consumes greater than 10% of total execution time, providing an opportunity to save about 2% total application energy by reducing the clock frequency for this loop. The `miniFE` loop takes about 50% of total execution time, potentially saving nearly 10% total execution energy.

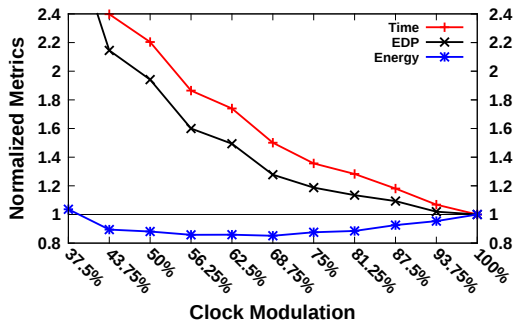
Low Memory Access Density Loops

Compute intensive loops correspond to loops with low memory density. Similar to Figure 5.2, reducing frequency using CPU clock modulation has a significant impact on the execution time of such loops. While the energy consumption may still be reduced (because of the power decrease), the energy-delay product (EDP) will get higher due to a substantial performance downgrade.

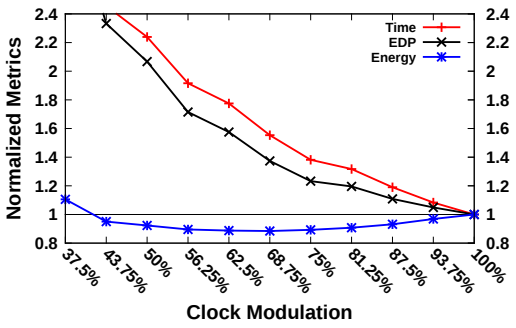
Figure 5.6 shows the normalized execution time, energy consumption, and EDP of compute-intensive loops in `LULESH`, `miniFE`, and `doitgen` and `covariance` of Polybench. In all figures the execution time increases are proportional to the decreases of the effective frequencies. The EDP curve immediately rises. The EDP curve is flat at 93.75% in Figure 5.6(d) and rises for any lower values. In constrast to Figures 5.4 and 5.5, the energy curves are much less sharp and tend to be flat in Figure 5.6. In Figure 5.6(a), the minimum energy occurs at 56.25% with a total savings of 14.2%. The execution time was increased by 80% and the EDP more than 50%. These results show that these kinds of loops should be run at full speed.

Polybench Loops: A Collection of All Three Types of Loops

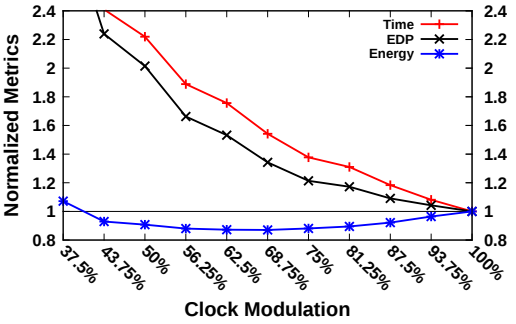
Figure 5.7 shows the normalized time, energy, and EDP of the Polybench kernels running at the best non-full speed setting. The figure sorts the kernels by normalized EDP in increasing order from left to right. In addition to showing the detailed values in Figure 5.7, Table 5.2 shows each benchmark's Memory-Access Density value in the



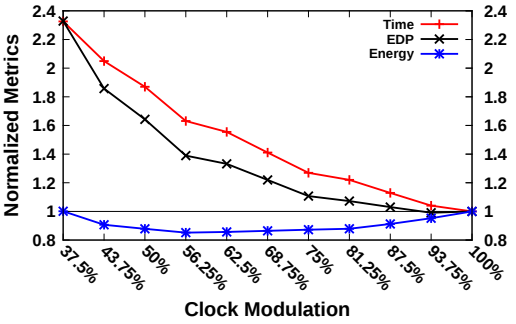
(a) This figure shows the metrics of a `LULESH` loop when varying clock modulation



(b) This figure shows the metrics of a `miniFE` loop when varying clock modulation



(c) This figure shows the metrics of `doitgen` Polybench when varying clock modulation



(d) This figure shows the metrics of `covariance` Polybench when varying clock modulation

Figure 5.6: This figure shows low memory access loops in four benchmarks: `LULESH`, `miniFE`, `doitgen` Polybench, and `covariance` Polybench. The normalized metrics are energy, time, and EDP (lower is better). The baseline is 100% clock modulation setting.

last column. Five programs have approximately 20% EDP improvements running at a setting of either a 75% or 81.25%. These programs are shown on the left of the left vertical line in the figure. The average memory concurrency is high (44.4, as shown in Table 5.2) for these loops, while the computation is low (execution increases between 3% and 5%). The second group of six programs in between the two vertical lines are run at a higher optimal clock rate (87.5%). These programs have approximately the same average memory concurrency (43 to 53), but more parallel computation. Their

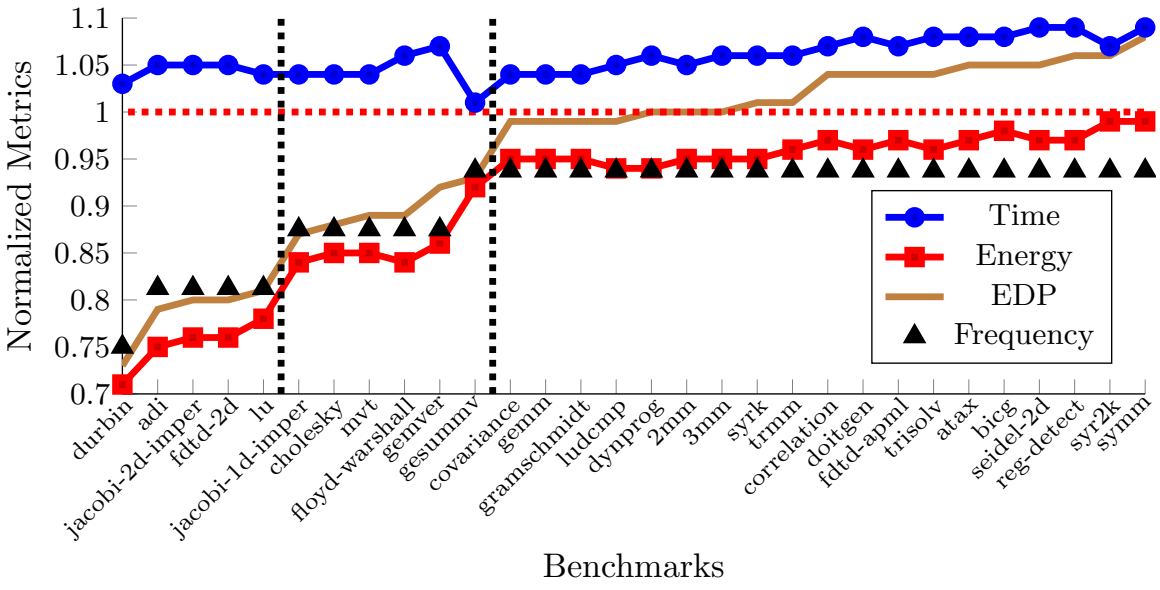


Figure 5.7: Graph showing the normalized time, energy, and EDP of polybench programs running at the best non-full speed setting.

performance degradation is between 4% and 7%. The clock squashing is increasing execution time, but EDP is still reduced by 8% to 13%. One program, `gesummv`, has very high concurrency (53) but enough computation so that the best clock frequency is 93.75%. It runs at almost full speed and EDP is reduced by 7%. For the rest of the programs (to the right of the right vertical line), the memory concurrency is low enough that the execution time is almost entirely a function of the CPU speed and any reduction in frequency result in EDP being either flat or worse than executing the program at full speed. The last row shows that for these programs the EDP is increased to 1.028 on average.

5.4.2 Multi-Frequency Execution Results

By changing frequencies during application execution specific to different loops, overall energy consumption can be improved with little execution time impact, achieving a better energy-delay product (EDP). We use the phrase "MultiFreq execution"

Table 5.2: This table shows the normalized time, energy, and EDP of polybench programs running at the best non-full speed setting. The Mem-Value column shows each benchmark's memory access density. Mean-High, Mean-Balanced, and Mean-Low give the average for three sets of kernels divided by the vertical lines in Figure 5.7.

Benchmarks	*Freq. Setting*	*Time*	*Energy*	*EDP*	*Mem-Value*
durbin	75.00%	1.03	0.71	0.73	33
adi	81.25%	1.05	0.75	0.79	45
jacobi-2d-imper	81.25%	1.05	0.76	0.80	47
fdtd-2d	81.25%	1.05	0.76	0.80	47
lu	81.25%	1.04	0.78	0.81	50
jacobi-1d-imper	87.50%	1.04	0.84	0.87	48
cholesky	87.50%	1.04	0.85	0.88	48
mvt	87.50%	1.04	0.85	0.89	43
floyd-warshall	87.50%	1.06	0.84	0.89	47
gemver	87.50%	1.07	0.86	0.92	47
gesummv	93.75%	1.01	0.92	0.93	53
covariance	93.75%	1.04	0.95	0.99	19
gemm	93.75%	1.04	0.95	0.99	16
gramschmidt	93.75%	1.04	0.95	0.99	25
ludcmp	93.75%	1.05	0.94	0.99	28
dynprog	93.75%	1.06	0.94	1.00	18
2mm	93.75%	1.05	0.95	1.00	17
3mm	93.75%	1.06	0.95	1.00	17
syrk	93.75%	1.06	0.95	1.01	17
trmm	93.75%	1.06	0.96	1.01	31
correlation	93.75%	1.07	0.97	1.04	14
doitgen	93.75%	1.08	0.96	1.04	4
fdtd-apml	93.75%	1.07	0.97	1.04	17
trisolv	93.75%	1.08	0.96	1.04	17
atax	93.75%	1.08	0.97	1.05	42
bicg	93.75%	1.08	0.98	1.05	33
seidel-2d	93.75%	1.09	0.97	1.05	11
reg_detect	93.75%	1.09	0.97	1.06	3
syr2k	93.75%	1.07	0.99	1.06	24
symm	93.75%	1.09	0.99	1.08	36
Mean-High	80%	1.044	0.752	0.786	44.4
Mean-Balanced	88.54%	1.043	0.86	0.897	47.7
Mean-Low	93.75%	1.068	0.962	1.028	20.56

to describe an execution of the program where multiple effective clock frequencies are used during the execution of a single application.

The `LULESH` and `miniFE` benchmarks both contain memory-bound and compute-bound loops. They are used to show the EDP benefits of multi-frequency execution. `miniFE` is part of the Mantevo Suite Release 2.0 [53]. It implements algorithms from an implicit finite-element application.

`LULESH` contains two loops with a high memory access density. Lowering the frequency while executing these loops changes their execution time only minimally. Since these two loops take up around 25% of total application time, 40% energy savings with a 16% of execution time increase for the two loops translates into 10% energy savings with only 4% of execution time increase for the entire application.

Table 5.3 compares the average execution time, energy consumption, and EDP of three multi-frequency `LULESH` executions with those of the single-frequency `LULESH` executions in the context of clock modulation. The top rows show the single frequency runs that minimized time (minT), energy (minE) and EDP (minEDP). The bottom rows are multi-frequency results, where the two high memory-access-density loops were run with reduced frequency (50%, 62.5%, and 68.75%) and rest of the code at maximum frequency. All values are normalized to the execution that led to minimum time (minT row). The MultiFreq-2 version of LULESH using multiple frequencies across different loops saved 10.3% energy and only increased execution time 2%. In the most aggressive scenario, the MultiFreq 1 version saved 11.8% with less than 5% execution time increase. MultiFreq 3, the least aggressive still saved 8.6% energy and resulted in a slowdown of only 1.5%. Comparing to the best single-frequency EDP version, multi-frequency execution achieved a better EDP (8.6% vs. 5.7%) with a greatly reduced performance penalty (2% vs. 15.7%). Multi-frequency execution's EDP was generally better and the performance penalty impact was much smaller.

Table 5.4 shows results from generating multi-frequency versions for the memory intensive loop in `miniFE`. The most aggressive setting, MultiFreq 1, reduces energy by 10.7%, but only runs for 2.9% longer. At this setting, EDP is improved by 8.1%. The

Table 5.3: This table compares the execution time, energy consumption, and EDP for `LULESH`

Version	*Duty Cycle Level*	*Time*	*Energy*	*EDP*
minT	100%	1.000	1.000	1.000
minE	56.25%	1.542	0.743	1.145
minEDP	81.25%	1.157	0.816	0.943
MultiFreq 1	100% & 50%	1.049	0.882	0.925
MultiFreq 2	100% & 62.5%	1.020	0.897	0.914
MultiFreq 3	100% & 68.75%	1.015	0.914	0.928

Table 5.4: This table compares the execution time, energy consumption, and EDP for `miniFE`

Version	*Duty Cycle Level*	*Time*	*Energy*	*EDP*
minT	100%	1.000	1.000	1.000
minE	62.5%	1.351	0.763	1.031
minEDP	81.25%	1.153	0.819	0.945
MultiFreq 1	100% & 81.25%	1.029	0.893	0.919
MultiFreq 2	100% & 87.5%	1.023	0.923	0.944
MultiFreq 3	100% & 93.75%	1.000	0.954	0.954

least aggressive setting, MultiFreq 3, increased execution time by only 0.04%, yet still saved 4.6% energy. When the results are compared to the best cases running at a single frequency, energy is minimized at a lower frequency (62.5% clock - 23.7%), but at a noticeable performance cost (35.1%) and an increased EDP (3.1%). The best EDP running at a single frequency (81.25%) is only 5.5% better than running full speed, but by going to a multi-frequency execution EDP savings over the default full speed execution can be increased by 8.1%.

We also investigated the multi-frequency execution potential of the NAS Parallel Benchmarks (NPB). These benchmarks are thought of as generally compute heavy codes. We summarize whether multi-frequency execution benefits each benchmark in Table 5.5. If a multi-frequency execution of a benchmark is not beneficial (in terms of EDP), we report its normalized time, energy and EDP to be 1 and Duty Cycle Level 100%. Otherwise, we report the normalized time, energy and EDP that could be achieved with the combination of frequency (lower frequency applied to memory-bound

code regions). Half of the ten NPB benchmarks benefited from multi-frequency execution (lower EDP achieved). For MG and SP, less than 10% execution time increase yielded more than a 16% EDP improvement. LU benchmark achieved 10% EDP reduction with less than 5% performance slowdown. The BT and CG (Class D) benchmarks slightly improved EDP with 1% and 5% performance slowdown, respectively. Since the clock modulation latency is at least the OS context-switch latency, problem size D is required for achieving EDP improvement.

Table 5.5: This table shows the potential benefits of multi-frequency for NPB benchmarks.

Version	*Duty Cycle Level*	*Time*	*Energy*	*EDP*
BT (Class C)	100% & 75%	1.014	0.945	0.958
CG (Class C)	100%	1	1	1
CG (Class D)	100% & 93.75%	1.051	0.940	0.989
EP (Class C)	100%	1	1	1
FT (Class C)	100%	1	1	1
IS (Class C)	100%	1	1	1
LU (Class D)	100% & 87.5%	1.058	0.852	0.902
MG (Class D)	100% & 75%	1.098	0.759	0.834
SP (Class D)	100% & 75%	1.091	0.748	0.816
UA (Class C)	100%	1	1	1

Energy and the energy-delay product (EDP) can be reduced by changing frequency for each parallel region with little performance impact. By setting the appropriate frequency for each parallel loop, an application can save energy and reduce EDP. Previous work statically or dynamically adjusted frequencies for MPI program phases using DVFS [26, 51], we distinguish our work by applying clock modulation to OpenMP application phases, which are usually more fine-grained than MPI phases.

5.4.3 Memory Access Density Based Runtime Energy Control

We have shown that our memory access density metric accurately indicates when frequency can be safely reduced. We developed a simple runtime control program that monitors the metric and reduces the frequency if the metric is above a threshold.

The sample interval and the threshold are all empirically determined. We show that applications can automatically get improved EDP using our memory access density based runtime energy control.

In this experiment, we tested MG and SP of the NPB benchmark suite. Table 5.6 compares the execution with runtime energy control versus static energy control for both benchmarks. Static-single means the application is run with a single and maximum frequency. Static-multiple means the application is statically determined to run with multiple frequencies. Dynamic means the machine frequency is dynamically changed while running the application, determined by the memory access density value.

Table 5.6: Comparison of dynamic energy control with static energy control of MG and SP NPB benchmark is shown. All metrics are relative to those of static-single.

Benchmarks	*Time (s)*	*Energy (J)*	*EDP (J·s)*
MG/static-single	1	1	1
MG/static-multiple	1.097	0.759	0.833
MG/dynamic	1.071	0.791	0.848
SP/static-single	1	1	1
SP/static-multiple	1.091	0.748	0.816
SP/dynamic	1.056	0.876	0.925

We can see that for both SP and MG, the dynamic runtime change of frequency resulted in improved EDP, compared to the single static execution of benchmarks. Although the dynamic approach does not save as much energy as the static-multiple case, the execution time with dynamic execution is lower for both benchmarks. For MG benchmark the dynamic approach achieves 20% energy savings and 15% EDP improvement with only 7% performance slowdown. For SP benchmark the dynamic approach reduces energy consumption by 12.4% and EDP by 7.5% with 5.6% performance slowdown.

Runtime energy control of applications using memory access density metric achieves automatic energy savings without the need of source code change. We are

working on improving the runtime energy control strategy to have more precise frequency change given a memory access density value.

5.5 Discussion

The pros and cons of energy control techniques need to be studied when they are used to execute programs with multiple frequencies. Frequency transition latency is an important aspect of the energy control techniques. CPU clock modulation and DVFS regulate the CPU frequency in two different mechanisms. The former requires only a change of a control register's content while the latter also involves changing the physical voltage and current. In our proposed multi-frequency execution of an application, frequency change requests occur often. Therefore it is helpful to know how long it takes for the system to fulfill the frequency transition request. Measuring how long the energy control APIs take is not enough and may be inaccurate. There is inherent delay between when the software finishes executing the frequency change requests and when the hardware realizes the change. Because such delay is not directly measurable, we use an open-source statistical tool called FTaLaT [55] to show that the CPU clock modulation frequency transition is actually much faster than DVFS.

5.5.1 Frequency Transition Latency Comparison

Intuitively, because changing the frequency by DVFS involves physical regulation of voltage too, the latency between requesting a frequency and fulfilling it is relatively larger than software controlled clock modulation. We show this by comparing the latency of changing the frequency using DVFS and clock modulation. The estimation of latency is obtained from the FTaLaT (Frequency Transition Latency) tool. The FTaLaT tool by default only supports the latency estimation of DVFS frequency change. We extended this tool to support the latency estimation of clock modulation frequency change[2].

[2] The extension can be found at https://github.com/weiwangudel/ftalat. A pull request is under review.

FTaLaT estimates the transition latency for each pairs of frequencies supported on a system. It uses a sophisticated statistical approach to separate the real transition delay from measurement noise. There are two main steps involved. The first step measures the execution time of a kernel with the start frequency and the target frequency. The second step sets the CPU frequency to target, and iteratively measure kernel execution time until a change of kernel execution time is detected with a certain confidence interval.

We replaced every DVFS frequency change function in the original FTaLaT code with our CPU clock modulation energy control function. Using the exact same approach, we accurately estimate the transition latency of CPU frequencies supported by clock modulation. Table 5.7 and Table 5.8 show the estimated frequency change latency in microseconds for each pair of the supported frequencies. We can see that the frequency transition latency for DVFS is about 30-40 microseconds. In sharp contrast, the latency for switching between 16 levels of clock modulation is less than 2 microseconds. Thus, the CPU clock modulation transition is roughly 15× to 20× faster than frequency transition using DVFS.

Table 5.7: This table shows DVFS frequency change latency in microseconds.

From (GHz) / **To (GHz)**	1.2	1.4	1.6	1.8	2.1	2.3	2.5	2.7
1.2	0	29	28	29	28	29	29	29
1.4	35	0	29	30	29	30	28	30
1.6	35	35	0	29	28	28	27	29
1.8	36	35	33	0	27	27	28	27
2.1	39	37	36	35	0	28	27	28
2.3	40	39	37	37	35	0	28	27
2.5	41	40	38	38	36	34	0	28
2.7	41	41	41	39	36	36	34	0

In summary, CPU clock modulation has lower frequency transition overhead than DVFS and can achieve the same or better energy efficiency than DVFS when applied in a fine-grained manner. Furthermore, it supports per-core energy control

Table 5.8: This figure shows the latency of changing clock modulation frequency in microseconds.

To \ From	5	6	7	8	9	10	11	12	13	14	15	16
5	0.0	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	0.6	0.6
6	1.3	0.0	1.3	1.3	1.3	1.3	1.3	1.3	1.2	1.3	0.6	0.6
7	1.3	1.3	0.0	1.3	1.3	1.3	1.3	1.3	1.3	1.3	0.6	0.6
8	1.3	1.3	1.3	0.0	1.3	1.3	1.3	1.3	1.3	1.3	0.6	0.6
9	1.3	1.3	1.3	1.2	0.0	1.3	1.3	1.3	1.3	1.3	0.6	0.6
10	1.3	1.3	1.3	1.3	1.3	0.0	1.3	1.3	1.3	1.3	0.6	0.6
11	1.3	1.3	1.3	1.3	1.3	1.3	0.0	1.3	1.3	1.3	0.6	0.6
12	1.3	1.3	1.3	1.3	1.3	1.3	1.3	0.0	1.3	1.3	0.6	0.6
13	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3	0.0	1.3	0.6	0.6
14	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3	0.0	0.6	0.6
15	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3	0.0	0.5
16	1.3	1.2	1.3	1.2	1.2	1.3	1.2	1.2	1.2	1.2	0.6	0.0

and could be applied on top of DVFS. In this work, we favor CPU clock modulation due to these two advantages over DVFS.

5.5.2 The Need for Unprivileged Energy Control

Previously we have shown that energy efficiency is improved with software controlled clock modulation. Greater energy efficiency can be achieved with *unprivileged* control.

Motivation

Unprivileged energy control refers to changing CPU frequencies without going through the operating system's kernel space. The frequency transition latency is greatly reduced because applications can request the desired frequency without incurring a context-switch overhead. Depending on the required granularity of performing energy control, existing techniques like clock modulation may be severely limited by the overhead of executing a privileged *wrmsr* instruction. Taking software clock modulation as an example, even though energy control by an application is enabled by a

new system call,[3] the overhead is dominated by the OS context switch. At the lower level, the value of the IA32_CLOCK_MODULATION msr controls the clock modulation setting. The instruction to change this msr is still privileged, meaning that only the kernel code could execute this instruction. Table 5.9 shows the time it takes to execute 1 million pairs of setting/resetting frequency control code.

Table 5.9: This table shows the overhead of executing 1 million pairs of energy changes via system call. The overhead is in seconds per million pairs.

Changing From Clock Modulation level (Freq.) X to Clock Modulation level 16 (2.7GHz)	Time (Seconds)
X=5 (0.84GHz)	33.7456
X=6 (1.01GHz)	27.239
X=7 (1.18GHz)	22.8995
X=8 (1.35GHz)	22.011
X=9 (1.52GHz)	24.2508
X=10 (1.69GHz)	21.3569
X=11 (1.86GHz)	20.2146
X=12 (2.03GHz)	17.551
X=13 (2.19GHz)	17.6344
X=14 (2.36GHz)	6.78666
X=15 (2.53GHz)	6.31159
X=16 (2.7GHz)	4.81499

Each pair of the API call involves 8 msr writes. We see that even if there is zero latency in changing the clock modulation setting, the overhead of executing the APIs is in the order of microseconds. This overhead needs to be reduced. The proposed solution is to make only the writing to the IA32_CLOCK_MODULATION msr register *unprivileged*. Applications perform msr-write operation to this register in a way similar to performing arithmetic operations. By doing this, the software introduced overhead (including the kernel/user context switch) is greatly reduced. However, unprivileged energy control should be performed only in trusted environment. Such unprivileged

[3] We implemented a prototype system call that supports CPU clock modulation on an Intel SandyBridge architecture running Linux 3.2.63.

energy control mechanism violates security policies of an operating system. Nevertheless, more energy saving opportunities can be created by unprivileged clock modulation energy control. In addition, security risk can be minimized by only allowing the change of current core clock rate to be unprivileged (changing other cores that the code does not run on would still be privileged).

Preliminary Study of Clock Modulation and Decoupled Access-Execution Model

The benefits of unprivileged clock modulation control are many folds. For example, it can be used on top of DVFS energy control. Doing so extends the range of attainable machine frequencies without adding extra transition overhead on top of DVFS frequency change. Unprivileged clock modulation, if implemented, also makes *Decoupled Access-Execution (DAE) model* more advantageous. However, since there is no support for unprivileged energy control using clock modulation, we apply existing clock modulation techniques to DAE and use the results to explain why the privileged energy control will not benefit DAE.

The idea of the decoupled access-execution model [89, 44, 40, 45] is to perform data prefetch in fine-granularity access phases and then performing computation with the prefetched data in the execution phase. The CPU frequency can be lowered during the access phase because the CPU is idle waiting for data. In task-based decoupled access-execution model [44, 40, 45], the granularity of the access phase of a task is 2 to 30 microseconds and the execution phase is 5 to 320 microseconds on a 4-core SandyBridge architecture. As we have shown before, the frequency transition overhead is on the order of microseconds. Therefore, existing energy control mechanisms prevent decoupled access-execution model from being helpful in practice. Table 5.10 shows the execution time, energy, and power of running libQ benchmark with coupled access-execution model and decoupled access-execution model. In DAE, energy control API calls are inserted before and after the access task. The table shows that decoupled access-execution model with energy control resulted in significant overhead (20×

slowdown) compared to the traditional coupled access-execution (CAE) model. This is because the overhead of executing the energy control API is on the same order of manginitude as the access phase and execution phase, which are repeatedly called in a loop.

Table 5.10: This figure shows the execution time, energy, and power of coupled access-execution (CAE) model and decoupled access-execution (DAE) model for libQ benchmark. In DAE model, energy control APIs are called before and after the access phase.

Benchmarks	*Time (s)*	*Energy (J)*	*Power (Watts)*
libQ (CAE)	154.94	8280.21	53.44
libQ (DAE)	3162.55	252608.55	79.88

With the presence of an instant frequency transition mechanism, e.g. enabled by unprivileged msr-write to the IA32_CLOCK_MODULATION register, frequency transition can be achieved on the order of nanoseconds, i.e., a few instructions. Such reduction in frequency transition latency directly benefits decoupled access-execution model in that the model practically achieves greatly improved application energy efficiency.

In summary, using the existing energy control API incurs significant overhead and renders the DAE model useless. Unprivileged energy control would benefit various applications with respect to improving energy efficiency. Although it raises some new issues, the benefits it brings outweigh the disadvantages.

5.6 Summary

In this chapter, we optimized OpenMP applications for energy efficiency by per-loop energy control. Loops are categorized into three types according to how their performance and power respond to reduced frequencies. The improved energy-delay product is obtained by lowering the CPU frequency via clock modulation during the execution of memory intensive loops. For compute intensive code regions, the maximum frequency is chosen. CPU clock modulation is observed to have less overhead than

DVFS in supporting multi-frequency execution of applications, which requires fast transition from the existing frequency to the target frequency.

Chapter 6

COMBINING SOFTWARE AND HARDWARE TECHNIQUES FOR ENERGY OPTIMIZATION

6.1 Introduction

In Chapter 4, we investigated energy autotuning from the software perspective where a compiler transformation framework is used to generate application energy tuning space. Regular applications, including scientific kernels were transformed, vectorized, and parallelized to maximize the performance achieved by polyhedral compilers. The energy autotuning process predicted with up to an 83% accuracy the best optimization sequence that would result in the best energy consumption. In Chapter 5, we studied energy optimization from the hardware perspective where we leveraged the CPU clock modulation power management feature on modern Intel CPUs. We observed that applications with fine-granularity loops fall into three categories when considering whether their energy efficiency would be improved by lowering the CPU frequency (either by DVFS or clock modulation). Our focus in this chapter is to combine software and hardware techniques to achieve better energy efficiency of applications. In particular, we experiment with concurrency throttling (reducing the number of OpenMP software threads) along with modulating CPU clock frequency for application performance and energy. In order to study how combined techniques work on an IBM Power8 architecture, we apply concurrency throttling and DVFS[1] to several benchmarks. Aside from combining concurrency throttling with hardware power management techniques, we also study how compiler transformations are impacted by

[1] CPU clock modulation hardware feature is not supported on IBM Power8 architecture.

the CPU clock modulation technique. Lastly, such impact is studied in a power-capped environment where the CPU power consumption is limited by a preset threshold.

6.2 Approach

This section describes experiments combining software techniques (like polyhedral optimizations and concurrency throttling) with hardware techniques (like DVFS, CPU clock modulation, and power capping) on modern Intel and IBM multicore architectures.

6.2.1 Combining Clock Modulation with Concurrency Throttling

Clock modulation reclaims energy consumed by cores waiting for data to arrive during computation. It can be combined with other power saving techniques to save additional energy. Concurrency throttling, a technique that reduces the number of threads to mitigate resource contention, has been shown to be an effective approach to reduce energy consumption of several benchmarks including `LULESH` and `BOTS` [76]. Combining clock modulation and concurrency throttling reclaims energy by two methods: 1) active cores waiting for data and 2) inactive cores that will otherwise cause memory bandwidth contention. Benchmarks that benefit from concurrency throttling can be improved by combining it with clock modulation to achieve more energy savings and EDP improvement. Concurrency throttling for parallel regions is done manually and the best configuration (i.e. how many threads to use) is often determined statically via exhaustive search.

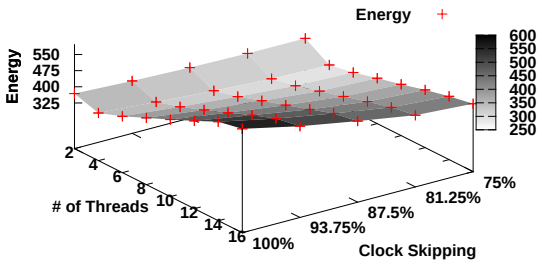
If a parallel loop has a lot of memory accesses and can benefit from clock modulation, we hypothesize that the number of threads executing the parallel loop could be reduced to reduce memory bandwidth contention. If the memory is being fully utilized, the performance impact will be minimal (eliminating thrashing could actually improve performance), and the power consumption will be greatly reduced (by idling cores). Clock modulation and concurrency throttling save energy in different ways and can be used together for significant combined improvement.

6.2.2 Combining DVFS with Concurrency Throttling on IBM Power8 Architecture

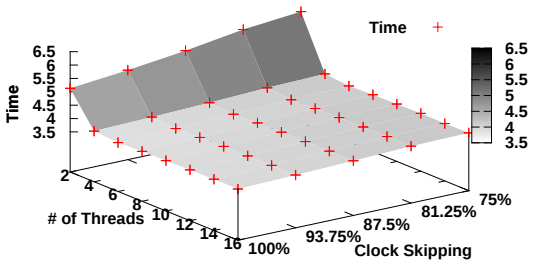
The IBM Power8 architecture features up to 8-way simultaneous multi-threading (SMT) per-core [24]. Users can configure the SMT level without restarting the system via a shell command. In addition, the Power8 architecture supports per-core DVFS control. Physical cores on the same chip could be configured to have different frequency and voltage combinations. The interface is supported by the ACPI-like powernv-cpufreq kernel module. There are 69 DVFS frequencies available on the Power8 architecture from 2.061GHz to 4.332GHz. Different from previous evaluations [3], we characterized application performance under both different DVFS and concurrency settings. Similar to Chapter 5 where we performed per-loop energy efficiency optimization, our analysis on the Power8 architecture is performed at the granularity of loops rather than the entire application. The fine granularity of measurements made it possible to combine DVFS and concurrency throttling to improve application performance and energy. We also compared application performance with SMT-8 and SMT-4 using the same amount of threads. We tested three OpenMP benchmarks: Graph500, LULESH 2.0, and miniFE. The results from running applications with different frequency and thread settings helped us identify memory-bound regions that could benefit from dynamic runtime frequency change. We invoke energy control API calls before and after these code regions so that the best performance and energy efficiency is achieved.

6.2.3 Polyhedral Transformation and Clock Modulation

In Chapter 5, we showed that applications were affected differently while reducing the CPU frequency via clock modulation. We applied polyhedral transformations to the original polybench application and studied how these transformed variants of the programs would react to CPU clock modulation with respect to their execution time, power, and energy consumption.



(a) Energy with concurrency throttling and clock modulation. Minimum occurs at (75%, 4)



(b) Time with concurrency throttling and clock modulation. Minimum occurs at (100%, 6)

Figure 6.1: `fdtd-2d` Polybench results when applying both concurrency throttling and clock modulation.

6.2.4 CPU clock modulation under power capping constraints

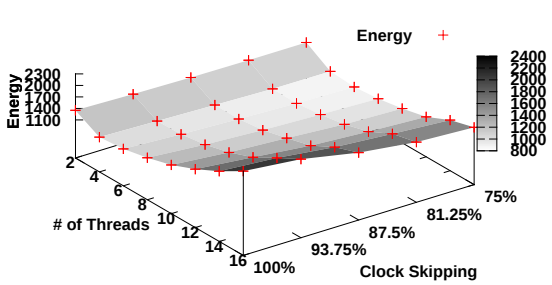
Results in Chapter 4 show that polyhedral transformation like loop tiling and/or loop fusion could significantly increase the application power consumption. Under a power-capped environment, the power consumption of the entire CPU is set to a limit. We study how compiler transformations that increase application power are affected by such a power bound. In addition to just evaluating polyhedral transformations under a power cap, we also changed the CPU clock modulation setting and observed how CPU clock modulation affected the performance and power of different compiler transformed code variants.

6.3 Results

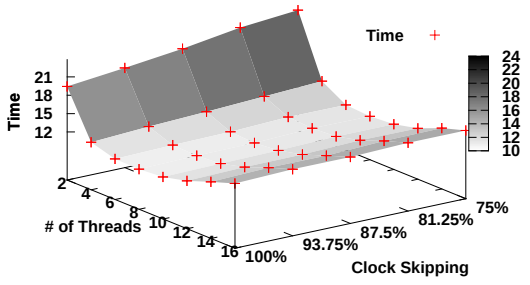
In this section, we report on experimental results of combining software and hardware techniques for application energy optimization.

6.3.1 Clock Modulation with Concurrency Throttling

We begin with the results of combining CPU clock modulation with concurrency throttling for improved energy efficiency, and then report our findings on the



(a) Energy results. Minimum occurs at (75%, 6)



(b) Time results. Minimum occurs at (100%, 8)

Figure 6.2: `LULESH` results when applying both concurrency throttling and clock modulation.

relationship between the memory access density and concurrency throttling.

Case 1: Concurrency Throttling is Beneficial

Figure 6.1 shows the energy and execution time of fdtd-2d when varying the number of threads from 2 to 16 and the clock modulation setting from 75% to 100%. Static concurrency throttling and clock modulation are applied to the whole program and the energy/time of its loop nest are reported. Note, fdtd-2d only contains one loop nest. Execution time on this loop is basically constant for all thread counts 4 and above. Looking at the performance for a fixed number of threads, as the clock frequency is reduced, it produces flat performance (maximum 7.4% increase for setting the clock to 75%). Comparing the normal execution (16 threads - 100% clock frequency) with the energy optimal choices (4 threads - 75% clock frequency), 54% of the energy is saved with only a 6.7% execution time increase. Adding clock modulation accounted for 10% energy savings even after concurrency throttling reduced the energy consumption by 44%. The tests where repeated using the GCC v4.4.6 compiler and the results were roughly equivalent when varying the number of threads and clock modulation.

For `LULESH`, concurrency throttling and clock modulation are applied to the whole program and the energy/time of one loop with high memory access density are

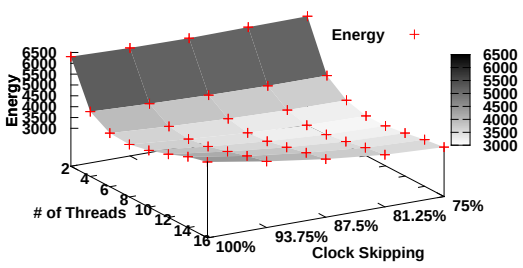
reported. Figure 6.2 shows a similar energy and execution time plot when varying the number of threads and the clock modulation. The minimum energy is achieved using 6 threads and a 75% setting of clock modulation. Execution time is *reduced* when reducing the number of threads. This loop suffers from memory bandwidth contention. Concurrency throttling mitigates such contention and improves performance. Looking at the performance for a fixed number of threads as the frequency is reduced, again it shows flat performance (maximum 6.6% increase for setting the clock to 75%). Energy consumption using 6 threads is driven down by 22% when setting the frequency to 75%.

Having identified the energy-saving potential of combining clock modulation and concurrency throttling, the two techniques are applied to the loops with high memory access density in `LULESH`. Up to 13% energy reduction and 17% EDP improvement are obtained using both techniques. Table 6.1 compares the metrics obtained from the default execution, the concurrency throttling execution (thread number changed from 16 to 6), and the combination of concurrency throttling and clock modulation execution (thread number changed to 6 and frequency reduced to 75%). Energy savings from clock modulation was an additional 7% to the 6% saved by concurrency throttling. Clock skipping increased execution time 3%, but when combined with concurrency throttling was still faster than the original execution. Overall, combining concurrency throttling and clock modulation lowered EDP by 17%, while concurrency throttling alone lowered EDP by 13%.

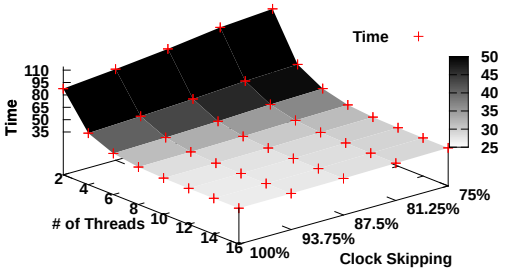
Table 6.1: `LULESH` with Concurrency Throttling (CT) and/or Clock Modulation (CM).

Version	*# of Threads*	*Duty Cycle Level*	*Time*	*Energy*	*EDP*
Default	16	100%	1.00	1.00	1.00
CT	6	100%	0.92	0.94	0.87
CT+CM	6	75%	0.95	0.87	0.83

`miniFE` has similar results when combining concurrency throttling and clock modulation. Figures 6.3(a) and 6.3(b) show the energy and execution time plot when



(a) Energy results. Minimum occurs at (75%, 8)



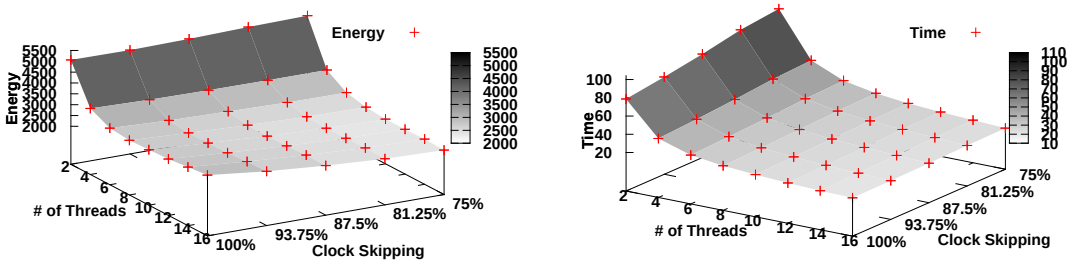
(b) Time results. Minimum occurs at (100%, 14)

Figure 6.3: `miniFE` results when applying both concurrency throttling and clock modulation.

varying the number of threads and the clock modulation for the memory intensive loop in `miniFE`. The minimum energy setting occurs with 8 threads and a 75% clock frequency, but the minimum time is achieved with 14 threads running at full clock frequency. The minimum EDP is achieved using 10 threads and a 81.25% setting of clock modulation. Compared to 16 threads and 100% setting, 33% energy is saved with 11% execution time increase. Since the loop takes up about *half* of the application execution time, a hybrid execution saves about 17% energy with less than 6% performance overhead. Table 6.2 compares the three different `miniFE` runs with different thread number and clock modulation settings for the memory intensive loop. When combining clock modulation and concurrency throttling, 21% energy is saved with only 6% performance increase, resulting in 16% EDP improvement.

Table 6.2: `miniFE` with Concurrency Throttling (CT) and Clock Modulation (CM).

Version	*# of Threads*	*Duty Cycle Level*	*Time*	*Energy*	*EDP*
Default	16	100%	1.00	1.00	1.00
CT	10	100%	1.02	0.86	0.88
CT+CM	10	81.25%	1.06	0.79	0.84



(a) Energy results. Minimum occurs at (75%, 16)

(b) Time results. Minimum occurs at (100%, 16)

Figure 6.4: `brdr2d` results when applying both concurrency throttling and clock modulation.

Case 2: Concurrency Throttling is not Beneficial

Concurrency throttling does not reduce execution time and EDP for all high memory density loops. Figure 6.4 shows that for `brdr2d`, reducing the number of threads increases the execution time and the energy consumption. Reducing the number of active memory references, reduces memory concurrency and overall performance. Clock modulation and reducing the effective clock rate from 100% to 75% does, however, decrease energy consumption by reclaiming energy consumed during stalls in the ALU. The minimum energy is achieved with 16 threads and a 75% effective clock rate.

Memory Access Density and Concurrency Throttling

Similar to our study on how memory access density is related to program's preference for the Duty Cycle Modulation setting, we varied the number of threads for Polybench and recorded the preferred number of the threads to achieve the best Energy-Delay Product (EDP). Figure 6.5 shows the results of these experiments. We can see that benchmarks that prefer lower frequencies in Figure 5.7 also prefer less than 16 threads to achieve optimal EDP, except cholesky. For fdtd-2d and floyd-warshall, the EDP improvement is larger than 45%. All but three benchmarks that prefer higher frequencies in Figure 5.7 prefer using 16 threads. The three benchmarks are trisolv,

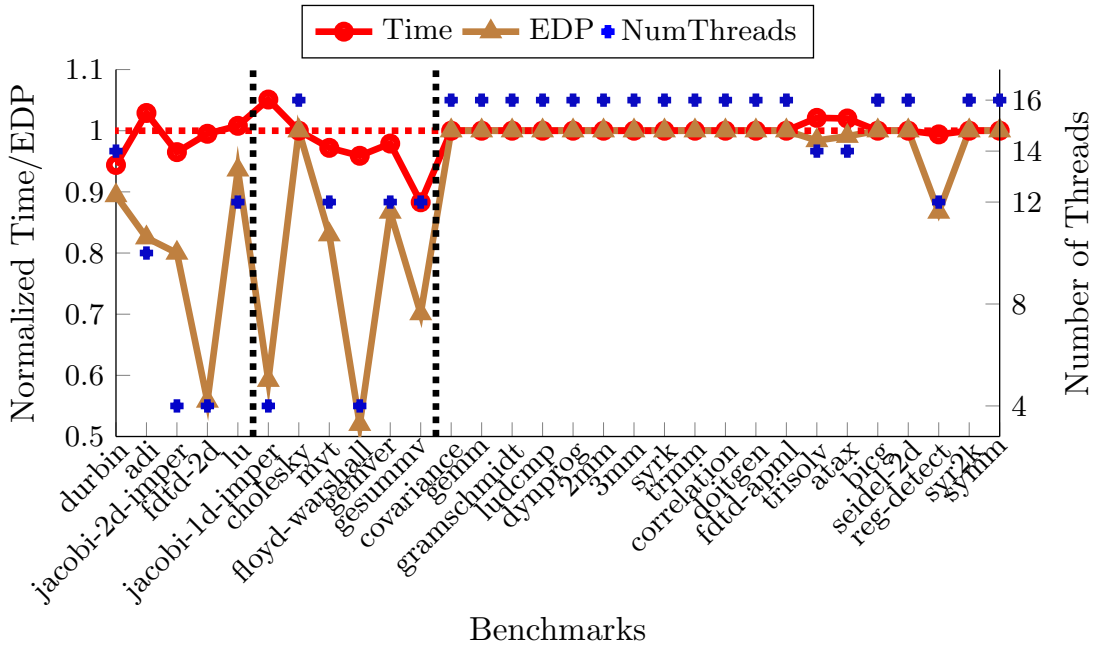
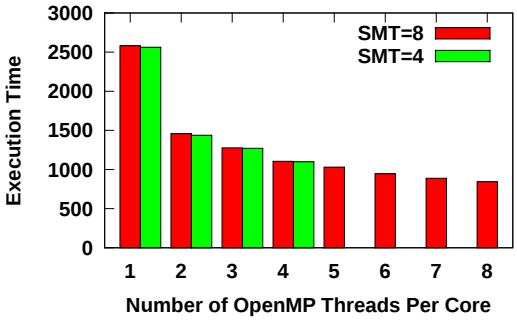
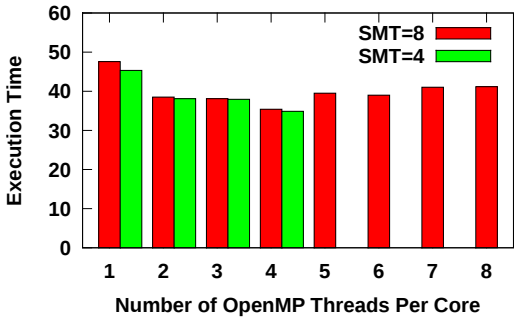


Figure 6.5: Graph showing the time and EDP of Polybench running with a thread configuration that achieves the best energy efficiency.

atax, and reg_detect and they perform best with 14, 14, and 12 threads, respectively. From Figures 5.7 and 6.5 we can see that benchmarks are likely to benefit from concurrency throttling and clock modulation if their memory access density is high. There is great potential to derive a model that predicts the clock modulation setting and thread number configuration based on the memory access density metric. We built a decision tree model that predicts the clock modulation setting using memory access density metric and source code features of the loops. Using Leave-One-Out-Cross-Validation, we achieved an accuracy of 90%. This means given a new application, we can determine the right clock modulation setting as well as the best thread count for that program with a high accuracy. We run the application once to collect our memory access density metric. Then we use the constructed decision tree model to predict the best frequency and number of threads to use for each application.



(a) This figure shows that Graph500 performs best with 8 threads per core.



(b) This figure shows that LULESH performs best with 4 threads per core.

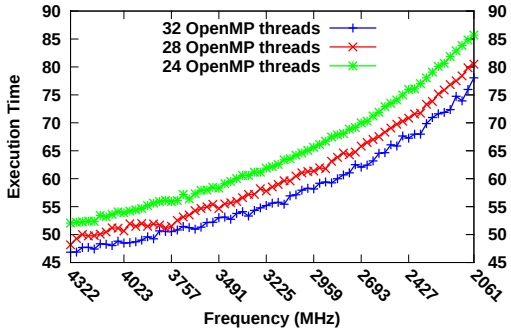
Figure 6.6: Comparing the application performance with varying number of threads per core and SMT settings

6.3.2 Combining software and hardware techniques on IBM Power8 system

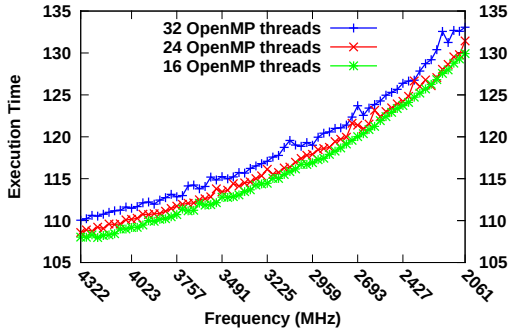
To evalulate whether combining DVFS and concurrency throttling is beneficial on IBM power8 systems, the experiments were performed on a TYAN GN70-BP010 Power8 system. The Tyan Power8 system has 4 processor cores and therefore supports up to 32 hardware threads on the SMT-8 configuration.

OpenMP Thread Concurrency Throttling

Figures 6.6(a) and 6.6(b) show the execution times when changing the number of OpenMP threads per core for Graph500 and LULESH. Applications do not always achieve the best performance with 8 threads per core. For example, LULESHperforms best with 4 threads per core. Graph500, on the other hand, is able to leverage all 8 threads per core. For the Graph500 benchmark, SMT-4 performs similarly to SMT-8. SMT-4 can potentially lead to better energy efficiency for applications that are memory-intense like LULESH and miniFE.



(a) Graph500: performance slowdown of more than 1.6X



(b) miniFE: performance slowdown of only 1.2X

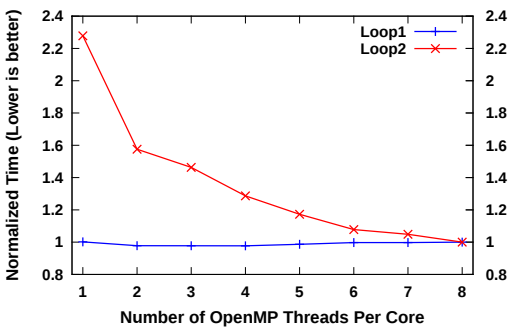
Figure 6.7: miniFE contains loops that respond differently when applying concurrency throttling and DVFS

DVFS

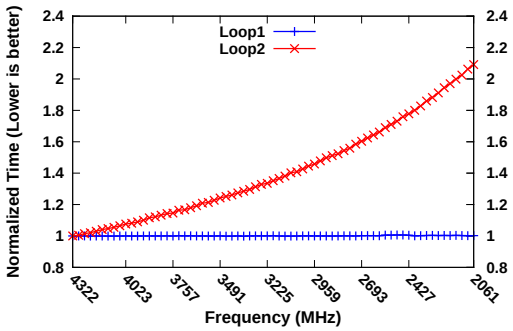
Figures 6.7(a) and 6.7(b) show the execution times when changing the DVFS frequency for the Graph500 benchmark and miniFE. A lower frequency leads to longer execution time for both benchmarks. However, applications slow down quite differently when reducing the frequency. Memory bound applications like miniFE provide energy saving opportunities for DVFS. Running at 2.061GHz, the Graph500 benchmark slows down by 1.6× while miniFE only slows down by 1.2×.

Combining DVFS and Concurrency Throttling

miniFE contains a dominant loop that is memory-bound. This loop (Loop1 in Figures 6.8(a) and 6.8(b)) is insensitive to reducing the frequency and the number of threads. The same two figures show that loops in miniFE behave differently when the frequency and the number of threads are reduced. Given this observation, we executed miniFE with 16 threads and mixed frequencies. Just before entering Loop1, the DVFS frequency is set to the minimum possible setting. The frequency is reset to the maximum immediately after Loop1. Table 6.3 shows that we achieve slight performance improvement (1.1%) by combining concurrency throttling and DVFS on



(a) miniFE: Concurrency throttling affects the two loops differently



(b) miniFE: DVFS affects the two loops differently

Figure 6.8: miniFE contains loops that respond differently when applying concurrency throttling and DVFS

the Power8 architecture.

Table 6.3: Applying DVFS and Concurrency Throttling improves performance and energy

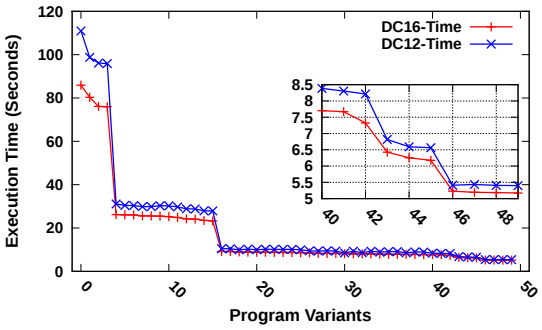
Executables	Total Number of Threads	DVFS Frequency	Execution Time
miniFE baseline	32	4.322 GHz	108.68 seconds
miniFE with DVFS & Concurrency Throttling	16	4.322 GHz & 2.061GHz	107.49 seconds

6.3.3 Polyhedral Transformation with CPU clock modulation

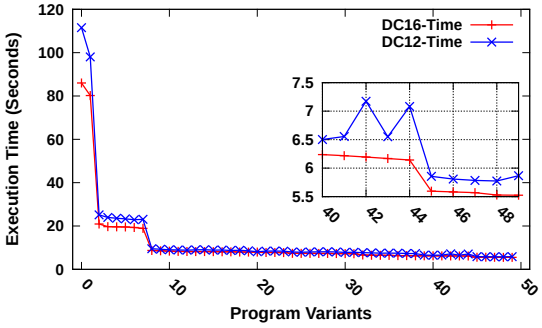
The impact of CPU Clock Modulation on the execution time, power consumption, and energy consumption of compiler transformed program variants is shown below.

Impact on Execution Time

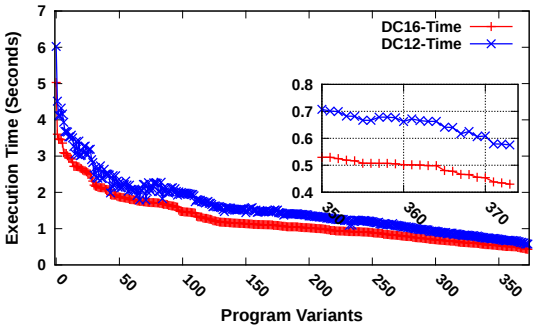
Figure 6.9 shows the execution time of the program variants of four polybench programs with two Duty Cycle Modulation settings: 16 (full frequency) and 12 (75%



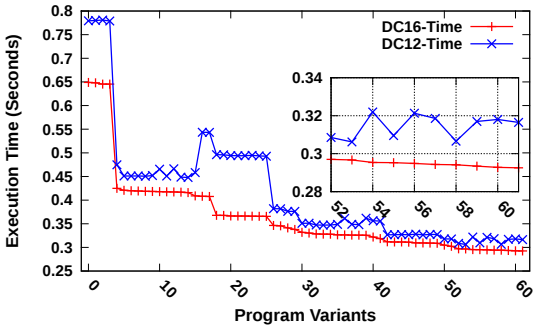
(a) Execution time of transformed program versions of jacobi-2D polybench under Duty Cycle Modulation setting of 16 (DC16) and 12 (DC12) respectively.



(b) Execution time of transformed program versions of fdtd-2D polybench under Duty Cycle Modulation setting of 16 (DC16) and 12 (DC12) respectively.



(c) Execution time of transformed program versions of 2mm polybench under Duty Cycle Modulation setting of 16 (DC16) and 12 (DC12) respectively.



(d) Execution time of transformed program versions of gesummv polybench under Duty Cycle Modulation setting of 16 (DC16) and 12 (DC12) respectively.

Figure 6.9: This figure shows the impact of changing Duty Cycle Modulation on the execution time of compiler transformed versions of the jacobi-2D, fdtd-2D, 2mm and gesummv polybench kernels.

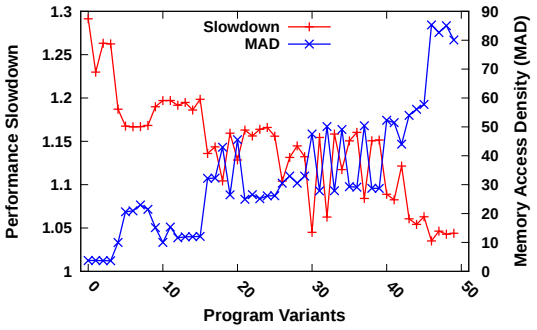
frequency). The X axis in all the graphs is sorted by the execution time under full frequency. The execution time of 50 compiler transformed program variants of jacobi-2D (Figure 6.9(a)) varies from 5 seconds to about 85 seconds with full frequency. For some fastest program variants of jacobi-2D, reducing the frequency by 25% percent (setting duty cycle to 12) only slightly increases the execution time. These program variants

had the *loop tiling* compiler transformation applied and a good tiling size played a key role in leading to the minimum execution time. Similar results were obtained for the other polybench stencil program fdtd-2D. Curves for fdtd-2D (Figure 6.9(b)) are in similar shape as those in jacobi-2D benchmark. Although the fastest program variant in full frequency case is not the fastest with duty cycle setting of 12, we suspect that this is due to the run-to-run execution variance. Figures 6.9(c) and 6.9(d) show the execution time of program variants for 2mm and gesummv polybench, respectively. Although the number of program variants varies between these four benchmarks, they are similar with regard to the impact of duty cycle modulation on the execution time. The fastest program variants are still the fastest when reducing the frequency by 25% using clock modulation.

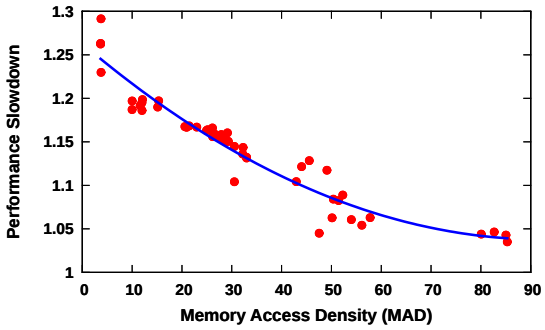
Reducing the effective frequency of the CPU by modulating the duty cycle level from 16 to 12 causes different performance impacts to compiler transformed program versions. A strong negative correlation between the performance slowdown and the Memory Access Density (MAD) metric explains why compiler transformed programs react differently to reduced frequency. Figure 6.10 shows that the jacobi-2D benchmark has a high correlation between the performance slowdown when reducing the frequency and the observed memory access density (MAD) metric. The X-axis of Figure 6.10(a) is the same as in Figure 6.9(a). The Y-axis gives the observed slowdown of each of the program variants when changing the duty cycle setting from 16 to 12 (i.e. reducing the speed by 25%). Figure 6.10(b) shows a fitted trend line. The function is $1.26284 - 0.00485301 * x + 0.0000261512 * x^2$ and yields a R-squared value of 0.9229. In addition to showing that program variants respond differently when reducing the frequency, Figure 6.10 shows that the MAD metric is highly correlated with the performance slowdown.

Impact on Power Consumption

We now look at CPU clock modulation's impact on the power consumption of the four polybench programs, as shown in Figure 6.11. Note that the figures are



(a) The performance slowdown and the Memory Access Density of all compiler transformed programs.



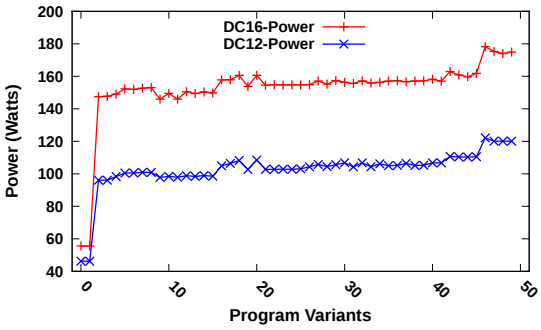
(b) The scattered plot of performance slowdown over the Memory Access Density and the trend line.

Figure 6.10: This figure shows the correlation between the Memory Access Density metric and the performance slowdown of jacobi-2D benchmark.

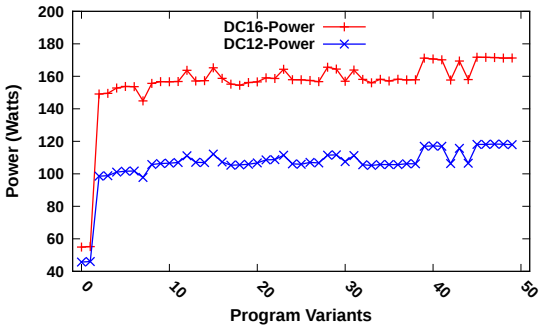
sorted in the same order as in Figure 6.9. For jacobi-2D and fdtd-2D polybench (Figures 6.11(a) and 6.11(b), the fastest executables happen to consume the largest amount of power. Although the two stencil programs have been very consistent in showing the fastest compiler transformed program consumed the highest amount of power and that the same executable consumed the least amount of energy, this is hardly the case for all benchmarks. Figures 6.11(c) and 6.11(d) show that the fastest program variants did not consume the maximum amount of power, in constrast to the jacobi-2D and fdtd-2D stencil benchmarks. For all benchmarks, almost every executable's power consumption is reduced by 50 Watts to 60 Watts due to reduced CPU clock frequency.

Impact on Energy Consumption

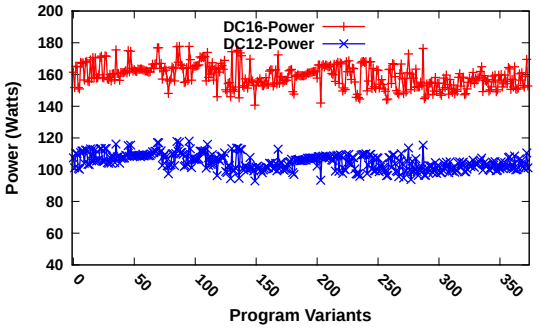
Figure 6.12 shows the energy consumption of program variants of four polybench programs under two duty cycle modulation settings. For all benchmarks the energy consumption in DC12 (75% frequency) case is much lower than that in DC16 (100%) for all program variants. Looking at the energy consumption together with Figure 6.9, it is not surprising that in most cases the fastest programs consume the least amount of energy.



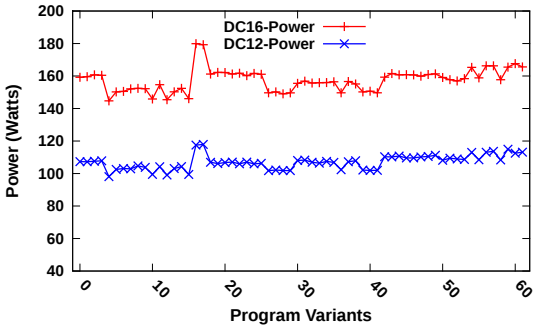
(a) Power consumption of transformed program versions of jacobi-2D polybench under Duty Cycle Modulation setting of 16 (DC16) and 12 (DC12) respectively.



(b) Power consumption of transformed program versions of fdtd-2D polybench under Duty Cycle Modulation setting of 16 (DC16) and 12 (DC12) respectively.



(c) Power consumption of transformed program versions of 2mm polybench under Duty Cycle Modulation setting of 16 (DC16) and 12 (DC12) respectively.

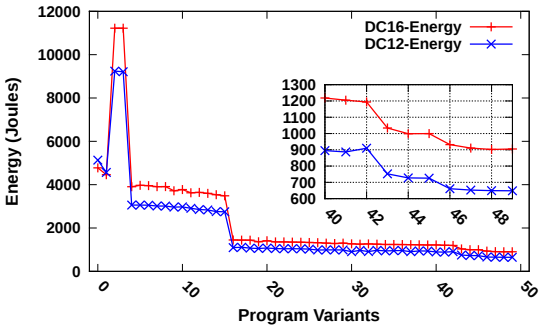


(d) Power consumption of transformed program versions of gesummv polybench under Duty Cycle Modulation setting of 16 (DC16) and 12 (DC12) respectively.

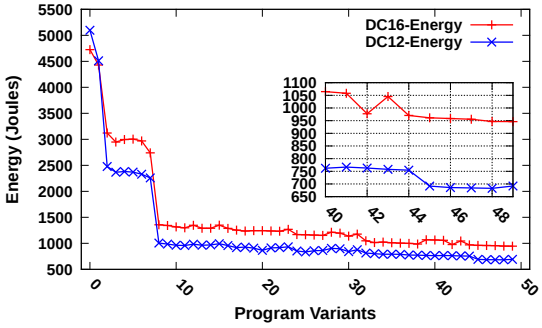
Figure 6.11: This figure shows the impact of changing Duty Cycle Modulation on the power consumption of compiler transformed versions of the jacobi-2D, fdtd-2D, 2mm and gesummv polybench kernels.

Summary

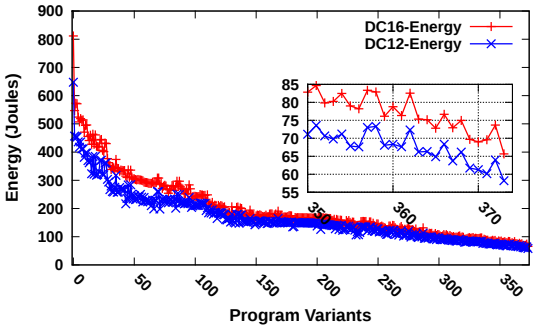
In summary, applying loop tiling with good tile sizes usually gives the fastest programs with relatively high power consumption, but still consume the least amount of energy. When reducing the CPU frequency via clock modulation, the power and energy consumption for all transformed programs decreases while the execution time



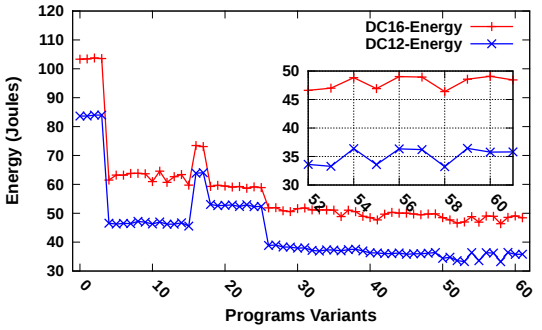
(a) Energy consumption of transformed program versions of jacobi-2D polybench under Duty Cycle Modulation setting of 16 (DC16) and 12 (DC12) respectively.



(b) Energy consumption of transformed program versions of fdtd-2D polybench under Duty Cycle Modulation setting of 16 (DC16) and 12 (DC12) respectively.



(c) Energy consumption of transformed program versions of 2mm polybench under Duty Cycle Modulation setting of 16 (DC16) and 12 (DC12) respectively.



(d) Energy consumption of transformed program versions of gesummv polybench under Duty Cycle Modulation setting of 16 (DC16) and 12 (DC12) respectively.

Figure 6.12: This figure shows the impact of changing Duty Cycle Modulation on the energy consumption of compiler transformed versions of the jacobi-2D, fdtd-2D, 2mm and gesummv polybench kernels.

increases.

6.3.4 CPU clock modulation under power capping constraints

We now experiment with a power cap of 30 Watts for each socket (60 Watts cap for two sockets). Surprisingly, we observed speedups for some program variants when reducing the CPU frequency by modulating the CPU clock.

Table 6.4: The execution time, energy and power consumption of jacobi-2D benchmark with 60 Watts power cap and different DC setting.

DC Level	Time (seconds)	Energy (Joules)	Power (Watts)
10	136.65	5712.81	41.81
12	111.36	5103.09	45.82
14	102.49	4971.64	48.51
16	118.91	5602.35	47.12

Table 6.5: The execution time, energy and power consumption of jacobi-2D benchmark with different DC setting but no power cap.

DC Level	Time (seconds)	Energy (Joules)	Power (Watts)
10	136.50	5735.15	42.01
12	110.94	5132.13	46.26
14	98.12	4890.33	49.84
16	85.91	4776.52	55.60

Table 6.4 shows the results obtained with one program variant of jacobi-2d benchmark under a power cap. The program took 118.91 seconds to finish execution with a duty cycle level of 16 but ran about 15% faster with a reduced frequency (duty cycle level of 14). The power was slightly increased and the energy consumed is about 12% less. Running with a duty cycle level of 12 also yielded faster execution than the default case. This phenomenon is non-existent in non power-cap case. Table 6.5 shows exactly the same executions without the power cap. The fastest execution among all duty cycle modulation settings is 16, and it took only 85.91 seconds to run. All other duty cycle settings all were 14% to 58% slower.

The two tables show that power capping offered performance improvement opportunities to applying CPU clock modulation for some applications.

The above program variant of jacobi-2d is a special case in that its execution time is relatively long. We hypothesize from our results on the power consumption under no power cap condition that the value of 55.6 Watts indicates that the program variant was most likely running in sequential fashion. In other words, parallelization and loop tiling transformation were not turned on. For the optimal case where the

program variants of jacobi-2D ran much faster with high power consumption, we see completely different situation.

Tables 6.6 and 6.7 show the execution time, energy and power consumption of the best jacobi-2D program variants with the power cap and without the power cap. When there is no power cap, reducing the frequency via Duty Cycle modulation affected performance slightly, but we see significant decreases in power, resulting in great energy savings, as expected. When there is a power cap of 60 Watts, all settings from 10 to 16 reached the power cap. They all finished around the same time of 21 seconds. Although no clear speedup is observed in this case, running with Duty Cycle level of 14 was about 1.5% faster than running with a higher Duty Cycle level (20.92 seconds vs. 21.22 seconds).

Table 6.6: The execution time, energy and power consumption of the fastest jacobi-2D program variants with different DC setting and 60 Watts power cap.

DC Level	Time (seconds)	Energy (Joules)	Power (Watts)
10	21.31	1174.35	55.10
12	21.22	1173.67	55.31
14	20.92	1157.27	55.32
16	21.22	1168.4	55.07

Table 6.7: The execution time, energy and power consumption of the fastest jacobi-2D program variants with different DC setting but no power cap.

DC Level	Time (seconds)	Energy (Joules)	Power (Watts)
10	5.44	647.45	118.96
12	5.41	649.21	120.06
14	5.35	750.03	140.31
16	5.18	902.20	174.02

6.4 Summary

In this Chapter, we combined software and hardware techniques to optimize applications for energy. We found that combining CPU clock modulation with concurrency throttling on Intel Sandy Bridge architecture achieved better energy efficiency

than applying either one alone. On IBM Power8 architecture, DVFS combined with concurrency throttling resulted in performance improvement. When combining CPU clock modulation with polyhedral transformation, we observed that program variants with high memory access density are less impacted by clock modulation and therefore are good candidates for improving energy efficiency. For these program variants, reducing the CPU frequency decreases power consumption significantly but only increases the execution time modestly. When applying CPU clock modulation under a power limit, we observed that certain program variant could be sped up by reducing the frequency and therefore improved the energy efficiency.

Chapter 7

CONCLUSION

Today, the impact of high performance computing is ubiquitous thanks to advances in parallel architectures and the programming languages that expose their computing power. In this dissertation, parallelization and optimization of a 2D wave propagation simulation application is performed on modern parallel architectures including NVIDIA GPUs and an Intel MIC accelerator. The programming languages used to harness the parallel architectures include CUDA, OpenCL, OpenACC and OpenMP. Low-level CUDA and OpenCL implementations of the simulation achieved as much as 220× speedups on a NVIDIA GPU over the sequential implementation that runs on a CPU. Without any code modification, the OpenCL implementation can run on an Intel Xeon Phi accelerator and a speedup of more than 70× is observed over the sequential implementation. High-level OpenACC implementation involved minimum source code changes to the sequential version but the CUDA and OpenCL code automatically generated from the OpenACC code achieved speedups that are comparable to those achieved by manually written CUDA and OpenCL code. Although both manually written OpenCL implementation and automatically generated OpenCL implementation could run on Intel's MIC architecture, their performance does not outperform our OpenMP implementation. A speedup of more than 120× over the sequential CPU implementation is achieved. Similar to OpenACC implementation, the amount of code changes using OpenMP is much less than that using CUDA and OpenCL. We conclude that high-level programming languages like OpenACC and OpenMP require much less work than low-level programming languages like CUDA and OpenCL while achieving similar speedups.

As HPC enters the exascale era, system power and energy are are becoming increasingly critical. Understanding the relationship between energy optimization and performance optimization as well as developing energy optimization techniques are two aspects of addressing the power wall problem. Maximizing energy savings requires keeping the power consumption to a minimum while performance continues to improve. On the other hand, saving energy requires minimizing performance impact when performance optimization is less likely.

Using a polyhedral compiler on a variety of small benchmarks and small applications has shown a high degree of correlation between execution time and energy consumption. Individual optimizations can however have significant impact on the power required by an application. For example, the Polybench program, covariance, when transformed with the "maxfuse" option resulted in a 20+% power increase. With the correct tile size "maxfuse" also resulted in the 50+% time decrease. The "maxfuse" optimization increases power consumption, but reduces total energy required to complete the computation due to the decrease in execution time. Understanding how power and energy are used at the small scale can contribute to the understanding of power/energy requirements of Exascale applications. Polyhedral optimization techniques can provide significant increases in performance, but currently require significant user modifications for realistic applications in order to construct polyhedral-friendly SCoPs that have reasonable compilation times. On small realistic applications, like LULESH and brdr2d, polyhedral transformations can be searched to find effective tiling sizes for SCoPs within the applications. Polyhedral optimizations combined with energy measurement capability allows for energy and power auto-tuning of benchmarks. Given a program and its optimizations, the constructed machine learning model can predict the energy and power consumption of each optimization with an accuracy of around 80%. Auto-tuning faces a challenge from run-to-run variance. Variance comes from different sources such as OS scheduling, temperature, and energy measurement granularity. Actions such as warming up the processors and repeating runs should be performed to minimize the impact of run-to-run variance on the auto-tuning process.

Energy management and optimization are necessary to maximize the effectiveness of every Joule or Watt. Frequency control via either CPU clock modulation or DVFS to minimize the energy consumption at the loop level is a promising energy management strategy. These methods require fine-grained control. The lower transition overhead of CPU clock modulation compared to DVFS on Intel SandyBridge archiecture allows CPU frequency adjustment for every loop. Applying these methods to several mini-apps, we were able to achieve energy savings ranging from 4.6% to 11.4% with execution time increase of less than 1%. The energy-delay product (EDP) improvement varied between 4.6% and 10.8%. These results show that CPU clock modulation can effectively reduce application energy usage while keeping the performance impact low. The ideal loop for applying CPU clock modulation usually incurs intensive memory access and/or high cache misses (consider indirect memory accesses), resulting in high values from memory-traffic hardware counters. Since the processor is less busy during the execution period of the memory intensive loop, frequency can safely be reduced to save energy. When the same loop finishes execution, CPU clock modulation quickly transitions the frequency of the CPU to the default or maximum operating level. As of now CPU clock modulation is implemented via privileged instructions on modern Intel CPUs. It is expected that CPU clock modulation can benefits more use cases if it were implemented via unprivileged instructions. Unprivileged frequency control using CPU clock modulation will feature nano-seconds frequency transition delay since only a few instructions are required for modulating the CPU clock.

When hardware techniques like CPU clock modulation and DVFS are combined with software techniques like concurrency throttling and polyhedral transformations, more energy saving opportunities are possible. Concurrency throttling saves execution time and energy for application phases that over-use a shared hardware component (e.g. memory bandwidth). Concurrency throttling and CPU clock modulation save energy in complimentary ways. Concurrency throttling saves energy by idling un-needed threads and CPU clock modulation saves energy by eliminating un-needed CPU cycles. For `LULESH`, the combined techniques result in a 13% energy reduction, a 5% performance

improvement, and a 17% overall energy-delay product (EDP) improvement on Intel SandyBridge architecture. On the IBM Power8 architecture, concurrency throttling is applied with DVFS. No performance slowdown is observed with fewer number of active cores (due to concurrency throttling) and lower frequencies of the active cores (due to DVFS). Polyhedral transformations applied to a set of scientifc kernels resulted in semantically equivalent versions with varing performance and power signatures. For versions that contain memory access intensive code regions, CPU clock modulation can be applied to increase energy efficiency. Under power-capped environments where the entire CPU is limited to consume a threshold power, applying CPU clock modulation can increase the performance of some polyhedral-transformed code. On a SandyBridge system, setting a power cap of 60 Watts and reducing the CPU frequency resulted in a 15% speedup for a transformed version of the jacobi-2D polybench program.

BIBLIOGRAPHY

[1] The Green500 List - November 2015. http://www.green500.org/, 2016.

[2] The Top500 List - November 2015. http://www.top500.org/, 2016.

[3] Andrew V. Adinetz, Paul F. Baumeister, Hans Böttiger, Thorsten Hater, Thilo Maurer, Dirk Pleiter, Wolfram Schenck, and Sebastiano Fabio Schifano. Performance Evaluation of Scientific Applications on POWER8. In Stephen A. Jarvis, Steven A. Wright, and Simon D. Hammond, editors, *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, volume 8966 of *Lecture Notes in Computer Science*, pages 24–45. Springer International Publishing, 2015.

[4] Konstantin Agladze, Matthew W. Kay, Valentin Krinsky, and Narine Sarvazyan. Interaction between spiral and paced waves in cardiac tissue. *American Journal of Physiology - Heart and Circulatory Physiology*, 293(1):H503–H513, 2007.

[5] ANL. AURORA. http://aurora.alcf.anl.gov/, 2016.

[6] Peter E. Bailey, Aniruddha Marathe, David K. Lowenthal, Barry Rountree, and Martin Schulz. Finding the limits of power-constrained application performance. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC, pages 1–12, New York, NY, USA, 2015. ACM.

[7] G. W. Beeler and H. Reuter. Reconstruction of the action potential of ventricular myocardial fibres. *The Journal of Physiology*, 268(1):177–210, 1977.

[8] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In *Proceedings of the 19th joint European conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC/ETAPS, pages 283–303, 2010.

[9] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Robert Lucas, Mark Richards, Al Scarpelli, Steven Scott, Allan Snavely, Thomas Sterling, R. Stanley Williams, Katherine Yelick, Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson,

William Dally, Monty Denneau, Paul Franzon, William Harrod, Jon Hiller, Stephen Keckler, Dean Klein, Peter Kogge, R. Stanley Williams, and Katherine Yelick. Exascale computing study: Technology challenges in achieving exascale systems, 2008.

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

[11] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 101–113, 2008.

[12] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. *SIGPLAN Notices*, 43(6):101–113, 2008.

[13] D. Campbell. VSIPL++ acceleration using commodity graphics processors. In *HPCMP Users Group Conference*, pages 315–320, June 2006.

[14] John Cavazos, Scott Grauer-Gray, Robert Searles, William Killian, Lifan Xu, and Sudhee Ayalasomayajula. Polybench/ACC. http://cavazos-lab.github.io/PolyBench-ACC/, 2016.

[15] Chun Chen, Jacqueline Chame, and Mary Hall. Chill: A framework for composing high-level loop transformations. Technical report, University of Southern California, 2008.

[16] Sangyeun Cho and R.G. Melhem. On the Interplay of Parallelization, Program Performance, and Energy Consumption. *IEEE Transactions on Parallel and Distributed Systems*, 21(3):342–353, 2010.

[17] P. Cicotti, A. Tiwari, and L. Carrington. Efficient speed (ES): Adaptive DVFS and clock modulation for energy efficiency. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 158–166, Sept 2014.

[18] CUDA. About CUDA. https://developer.nvidia.com/about-cuda, 2016.

[19] Matthew Curtis-Maury, Ankur Shah, Filip Blagojevic, Dimitrios S. Nikolopoulos, Bronis R. de Supinski, and Martin Schulz. Prediction Models for Multi-dimensional Power-performance Optimization on Many Cores. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT, pages 250–259, 2008.

[20] M. Daga, W. Feng, and T. Scogland. Towards accelerating molecular modeling via multi-scale approximation on a GPU. In *2011 IEEE 1st International Conference on Computational Advances in Bio and Medical Sciences (ICCABS)*, pages 75–80, Feb. 2011.

[21] J. P. Drouhard and F. A Roberge. Revised formulation of the Hodgkin-Huxley representation of the sodium current in cardiac cells. *Computers and Biomedical Research*, 20(4):333–350, 1987.

[22] Stijn Eyerman and Lieven Eeckhout. Fine-grained DVFS Using On-chip Regulators. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(1):1–24, February 2011.

[23] Paul Feautrier. Some efficient solutions to the affine scheduling problem. part ii. multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, 1992.

[24] E.J. Fluhr, J. Friedrich, D. Dreps, V. Zyuban, G. Still, C. Gonzalez, A. Hall, D. Hogenmiller, F. Malgioglio, R. Nett, J. Paredes, J. Pille, D. Plass, R. Puri, P. Restle, D. Shan, K. Stawiasz, Z.T. Deniz, D. Wendel, and M. Ziegler. 5.1 POWER8^{TM}: A 12-core server-class processor in 22nm SOI with 7.6Tb/s off-chip bandwidth. In *IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 96–97, Feb 2014.

[25] Vincent W. Freeh, Nandini Kappiah, David K. Lowenthal, and Tyler K. Bletsch. Just-in-time dynamic voltage scaling: Exploiting inter-node slack to save energy in MPI programs. *Journal of Parallel and Distributed Computing*, 68(9):1175–1185, 2008.

[26] Vincent W. Freeh and David K. Lowenthal. Using Multiple Energy Gears in MPI Programs on a Power-scalable Cluster. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 164–173, 2005.

[27] Elkin Garcia and Guang Gao. Strategies for improving performance and energy efficiency on a many-core. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF, pages 1–4, 2013.

[28] Rong Ge, Xizhou Feng, Wu chun Feng, and K.W. Cameron. CPU MISER: A Performance-Directed, Run-Time System for Power-Aware Clusters. In *International Conference on Parallel Processing*, ICPP, pages 1–8, Sept 2007.

[29] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming*, 34(3):261–317, June 2006.

[30] Scott Grauer-Gray, William Killian, Robert Searles, and John Cavazos. Accelerating financial applications on the GPU. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 127–136, New York, NY, USA, 2013. ACM.

[31] Martin Griebl and Christian Lengauer. The loop parallelizer LooPo. In *Proc. Sixth Workshop on Compilers for Parallel Computers*, pages 311–320, 1996.

[32] Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. Measuring Energy Consumption for Short Code Paths Using RAPL. *ACM SIGMETRICS Performance Evaluation Review*, 40(3):13–17, 2012.

[33] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, and T. Burton. Haswell: The Fourth-Generation Intel Core Processor. *IEEE Micro*, 34(2):6–20, Mar 2014.

[34] A. Hart, R. Ansaloni, and A. Gray. Porting and scaling OpenACC applications on massively-parallel, GPU-accelerated supercomputers. *European Physical Journal - Special Topics*, 210(1):5–16, 2012.

[35] J. A. Herdman, W. P. Gaudin, S. McIntosh-Smith, M. Boulton, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis. Accelerating hydrocodes with OpenACC, OpeCL and CUDA. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, SCC, pages 465–471, Washington, DC, USA, 2012. IEEE Computer Society.

[36] J. Hsu. When will we have an exascale supercomputer? [news]. *IEEE Spectrum*, 52(1):13–16, January 2015.

[37] Intel. Intel 64 and IA-32 architecures software developer's manual, volume 3. http://download.intel.com/products/processor/manual/253669.pdf, 2016.

[38] Intel. Intel Xeon Processor E5-2600 Product Family Uncore Performance Monitoring Guide. http://www.intel.com/content/dam/www/public/us/en/documents/design-guides/xeon-e5-2600-uncore-guide.pdf, 2016.

[39] Intel®. Intel®Power Gadget. https://software.intel.com/en-us/articles/intel-power-gadget-20, 2016.

[40] Alexandra Jimborean, Konstantinos Koukos, Vasileios Spiliopoulos, David Black-Schaffer, and Stefanos Kaxiras. Fix the code. Don't tweak the hardware: A new compiler approach to voltage-frequency scaling. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO, pages 262–272, New York, NY, USA, 2014. ACM.

[41] N. Kappiah, Vincent W. Freeh, and D.K. Lowenthal. Just In Time Dynamic Voltage Scaling: Exploiting inter-node slack to save energy in MPI programs. In *Proceedings of the ACM/IEEE SC Conference*, pages 1–9, Nov 2005.

[42] Ian Karlin, Abhinav. Bhatele, Bradford L. Chamberlain, Jonathan. Cohen, Zachary Devito, Maya Gokhale, Riyaz Haque, Rich Hornung, Jeff Keasler, Dan Laney, Edward Luke, Scott Lloyd, Jim McGraw, Rob Neely, David Richards, Martin Schulz, Charle H. Still, Felix Wang, and Daniel Wong. Lulesh programming model and performance ports overview. Technical Report LLNL-TR-608824, December 2012.

[43] M. W. Kay and R. A. Gray. Measuring curvature and velocity vector fields for waves of cardiac excitation in 2-D media. *IEEE Transactions on Biomedical Engineering*, 52(1):50–63, Jan. 2005.

[44] Konstantinos Koukos, David Black-Schaffer, Vasileios Spiliopoulos, and Stefanos Kaxiras. Towards more efficient execution: A decoupled access-execute approach. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS, pages 253–262, New York, NY, USA, 2013. ACM.

[45] Konstantinos Koukos, Per Ekemark, Georgios Zacharopoulos, Vasileios Spiliopoulos, Stefanos Kaxiras, and Alexandra Jimborean. Multiversioned decoupled access-execute: The key to energy-efficient compilation of general-purpose programs. In *Proceedings of the 25th International Conference on Compiler Construction*, CC, pages 121–131, 2016.

[46] N. Kurd, M. Chowdhury, E. Burton, T. P. Thomas, C. Mozak, B. Boswell, M. Lal, A. Deval, J. Douglas, M. Elassal, A. Nalamalpu, T. M. Wilson, M. Merten, S. Chennupaty, W. Gomes, and R. Kumar. 5.9 Haswell: A family of IA 22nm processors. In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 112–113, Feb 2014.

[47] N. Kurd, M. Chowdhury, E. Burton, T. P. Thomas, C. Mozak, B. Boswell, P. Mosalikanti, M. Neidengard, A. Deval, A. Khanna, N. Chowdhury, R. Rajwar, T. M. Wilson, and R. Kumar. Haswell: A Family of IA 22 nm Processors. *IEEE Journal of Solid-State Circuits*, 50(1):49–58, Jan 2015.

[48] Charles Lefurgy, Xiaorui Wang, and Malcolm Ware. Power capping: a prelude to power shifting. *Cluster Computing*, 11(2):183–195, 2008.

[49] Dong Li, B.R. de Supinski, M. Schulz, D.S. Nikolopoulos, and K.W. Cameron. Strategies for Energy-Efficient Resource Management of Hybrid Programming Models. *IEEE Transactions on Parallel and Distributed Systems*, 24(1):144–157, Jan 2013.

[50] Charles Lively, Xingfu Wu, Valerie Taylor, Shirley Moore, Hung-Ching Chang, Chun-Yi Su, and Kirk Cameron. Power-aware predictive models of hybrid (MPI/OpenMP) scientific applications on multicore systems. *Computer Science - Research and Development*, 27(4):245–253, 2012.

[51] Kelly Livingston, Nicolas Triquenaux, Thibault Fighiera, Jean Christophe Beyler, and William Jalby. Computer using too much power? Give it a REST (Runtime Energy Saving Technology). *Computer Science - Research and Development*, 29(2):123–130, 2014.

[52] LLNL. Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH). https://codesign.llnl.gov/lulesh.php, 2016.

[53] Mantevo.org. Home of the mantevo project. https://mantevo.org, 2016.

[54] Aniruddha Marathe, Peter E. Bailey, David K. Lowenthal, Barry Rountree, Martin Schulz, and Bronis R. Supinski. *ISC High Performance*, chapter A Run-Time System for Power-Constrained HPC Applications, pages 394–408. Springer International Publishing, 2015.

[55] Abdelhafid Mazouz, Alexandre Laurent, Benoît Pradelle, and William Jalby. Evaluation of CPU frequency transition latency. *Computer Science - Research and Development,*, 29(3-4):187–195, August 2014.

[56] MIC. Intel Many Integrated Core architecture. http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html, 2016.

[57] J. Michalakes and M. Vachharajani. GPU acceleration of numerical weather prediction. In *IEEE International Symposium on Parallel and Distributed Processing*, IPDPS, pages 1–7, April 2008.

[58] Arthur A. Mirin, David F. Richards, James N. Glosli, Erik W. Draeger, Bor Chan, Jean-luc Fattebert, William D. Krauss, Tomas Oppelstrup, John Jeremy Rice, John A. Gunnels, Viatcheslav Gurev, Changhoan Kim, John Magerlein, Matthias Reumann, and Hui-Fang Wen. Toward real-time modeling of human heart ventricles at cellular resolution: simulation of drug-induced arrhythmias. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC, pages 1–11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[59] A. Neic, M. Liebmann, E. Hoetzl, L. Mitchell, E.J. Vigmond, G. Haase, and G. Plank. Accelerating cardiac bidomain simulations using graphics processing units. *IEEE Transactions on Biomedical Engineering*, 59(8):2281–2290, 2012.

[60] Steven Niederer, Lawrence Mitchell, Nicolas Smith, and Gernot Plank. Simulating human cardiac electrophysiology on clinical time-scales. *Frontiers in Physiology*, 2(14):1–7, 2011.

[61] Boyana Norris, Albert Hartono, and William Gropp. Annotations for productivity and performance portability. In *Petascale Computing: Algorithms and Applications*, pages 443–462. Chapman & Hall / CRC Press, Taylor and Francis Group, 2007. Preprint ANL/MCS-P1392-0107.

[62] Rafael Sachetto Oliveira, Bernardo Martins Rocha, Ronan Mendonça Amorim, Fernando Otaviano Campos, Wagner Meira, Elson Magalhães Toledo, and Rodrigo Weber dos Santos. Comparing CUDA, OpenCL and OpenGL implementations of the cardiac monodomain equations. In *Proceedings of the 9th international conference on Parallel Processing and Applied Mathematics - Volume Part II*, PPAM, pages 111–120, Berlin, Heidelberg, 2012. Springer-Verlag.

[63] OpenACC. OpenACC–directives for accelerators. http://www.openacc-standard.org, 2016.

[64] OpenCL. OpenCL–the open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl, 2016.

[65] OpenMP. OpenMP–the OpenMP API specification for parallel programming. http://openmp.org/wp/, 2016.

[66] OpenMP.org. OpenMP Application Programming Interface. http://www.openmp.org/mp-documents/openmp-4.5.pdf, 2016.

[67] ORNL. SUMMIT - Scale new heights. Discover new solutions. https://www.olcf.ornl.gov/summit/, 2016.

[68] Eunjung Park, John Cavazos, and Marco A. Alvarez. Using Graph-based Program Characterization for Predictive Modeling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO, pages 196–206, New York, NY, USA, 2012. ACM.

[69] Eunjung Park, John Cavazos, Louis-Noël Pouchet, Cdric Bastoul, Albert Cohen, and P. Sadayappan. Predictive modeling in a polyhedral optimization space. *International Journal of Parallel Programming*, 41(5):704–750, 2013.

[70] Eunjung Park, Louis-Noel Pouche, John Cavazos, Albert Cohen, and P. Sadayappan. Predictive modeling in a polyhedral optimization space. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO, pages 119–129, Washington, DC, USA, 2011. IEEE Computer Society.

[71] Bernard J. Pope, Blake G. Fitch, Michael C. Pitman, John J. Rice, and Matthias Reumann. Petascale computation performance of lightweight multiscale cardiac models using hybrid programming models. In *Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 433–436, 2011.

[72] B.J. Pope, B.G. Fitch, M.C. Pitman, J.J. Rice, and M. Reumann. Performance of hybrid programming models for multiscale cardiac simulations: Preparing for petascale computation. *IEEE Transactions on Biomedical Engineering*, 58(10):2965–2969, 2011.

[73] Allan Porterfield, Sridutt Bhalachandra, Wei Wang, and Rob Fowler. Variability: A tuning headache. In *The First IEEE International Workshop on Variability in Parallel and Distributed Systems (VarSys)*, May 2016.

[74] Allan Porterfield, Rob Fowler, Sridutt Bhalachandra, and Wei Wang. OpenMP and MPI Application Energy Measurement Variation. In *Proceedings of the 1st International Workshop on Energy Efficient Supercomputing*, E2SC, pages 1–8, 2013.

[75] Allan Porterfield, Rob Fowler, and Min Yeol Lim. RCRTool design document; version 0.1. Technical Report RENCI Technical Report TR-10-01, 2010.

[76] Allan K. Porterfield, Stephen L. Olivier, Sridutt Bhalachandra, and Jan F. Prins. Power Measurement and Concurrency Throttling for Energy Reduction in OpenMP Programs. In *IEEE 27th International Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW)*, pages 884–891, May 2013.

[77] Louis-Noël Pouchet. Polybench/C. http://web.cse.ohio-state.edu/~pouchet/software/polybench/, 2016.

[78] Louis-Noël Pouchet. PolyOpt/C: a Polyhedral Optimizer for the ROSE compiler. http://www.cs.ucla.edu/~pouchet/software/polyopt/, 2016.

[79] Louis-Noël Pouchet. PoCC: the Polyhedral Compiler Collection. http://www.cs.ucla.edu/~pouchet/software/pocc/, 201666666.

[80] Daniel J. Quinlan and Chunhua (Leo) Liao. ROSE Compiler. http://rosecompiler.org/, 2016.

[81] Shah Mohammad Faizur Rahman, Jichi Guo, Akshatha Bhat, Carlos Garcia, Majedul Haque Sujon, Qing Yi, Chunhua Liao, and Daniel Quinlan. 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, pages 123–132, 2012.

[82] R. Reyes, I. Lopez, J.J. Fumero, and F. De Sande. Directive-based programming for GPUs: A comparative study. In *High Performance Computing and Communication IEEE 9th International Conference on Embedded Software and Systems*, HPCC-ICESS, pages 410–417, 2012.

[83] Ruymán Reyes, Iván López, Juan J. Fumero, and Francisco Sande. A preliminary evaluation of OpenACC implementations. *Journal of Supercomputing*, 65(3):1063–1075, September 2013.

[84] David F. Richards, James N. Glosli, Erik W. Draeger, Arthur A. Mirin, Bor Chan, Jean-luc Fattebert, William D. Krauss, Tomas Oppelstrup, Chris J. Butler, John A. Gunnels, Viatcheslav Gurev, Changhoan Kim, John Magerlein, Matthias Reumann, Hui-Fang Wen, and John Jeremy Rice. Towards real-time simulation of cardiac electrophysiology in a human heart at high resolution. *Computer Methods in Biomechanics and Biomedical Engineering*, 16(7):802–805, 2013.

[85] Barry Rountree, David K. Lownenthal, Bronis R. de Supinski, Martin Schulz, Vincent W. Freeh, and Tyler Bletsch. Adagio: Making DVS Practical for Complex HPC Applications. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS, pages 460–469, New York, NY, USA, 2009. ACM.

[86] Simon Scarle. Implications of the Turing completeness of reaction-diffusion models, informed by GPGPU simulations on an XBox 360: Cardiac arrhythmias, re-entry and the Halting problem. *Computational Biology and Chemistry*, 33(4):253–260, 2009.

[87] Eric Schweitz, Richard Lethin, Allen Leung, and Benoit Meister. R-stream: A parametric high level compiler. In *IEEE High Performance Extreme Computing Conference*, HPEC, pages 1–2, 2006.

[88] S. Seo, G. Jo, and J. Lee. Performance characterization of the nas parallel benchmarks in opencl. In *IEEE International Symposium on Workload Characterization*, IISWC, pages 137–148, Nov 2011.

[89] James E. Smith. Decoupled access/execute computer architectures. In *Proceedings of the 9th Annual Symposium on Computer Architecture*, ISCA, pages 112–119, Los Alamitos, CA, USA, 1982. IEEE Computer Society Press.

[90] R. Spurzem, P. Berczik, K. Nitadori, G. Marcus, A. Kugel, R. Ma andnner, I. Berentzen, R. Klessen, and R. Banerjee. Astrophysical particle simulations with custom GPU clusters. In *IEEE 10th International Conference on Computer and Information Technology*, CIT, pages 1189–1195, 2010.

[91] Lijuan Su, Wenqia Wang, and Hong Wang. A characteristic difference method for the transient fractional convection-diffusion equations. *Applied Numerical Mathematics*, 61(8):946–960, August 2011.

[92] Xiaoquan Su, Jian Xu, and Kang Ning. Parallel-META: A high-performance computational pipeline for metagenomic data analysis. In *IEEE International Conference on Systems Biology*, ISB, pages 173–178, sept. 2011.

[93] DOE ASCAC Subcommittee. Top Ten Exascale Research Challenges. http://science.energy.gov/~/media/ascr/ascac/pdf/meetings/20140210/Top10reportFEB14.pdf, 2016.

[94] Vaibhav Sundriyal, Masha Sosonkina, Alexander Gaenko, and Zhao Zhang. Energy saving strategies for parallel applications with point-to-point communication phases. *Journal of Parallel and Distributed Computing*, 73:1157–1169, 2013.

[95] Vaibhav Sundriyal, Masha Sosonkina, and Zhao Zhang. Achieving energy efficiency during collective communications. *Concurrency and Computation: Practice and Experience*, 25(15):2140–2156, 2013.

[96] Vaibhav Sundriyal, Masha Sosonkina, and Zhao Zhang. Automatic runtime frequency-scaling system for energy savings in parallel applications. *The Journal of Supercomputing*, 68(2):777–797, 2014.

[97] K. ten Tusscher and A. Panfilov. Wave propagation in excitable media with randomly distributed obstacles. *Multiscale Modeling & Simulation*, 3(2):265–282, 2005.

[98] E. Tessmer. Seismic finite-difference modeling with spatially varying time steps. *GEOPHYSICS*, 65(4):1290–1293, 2000.

[99] A Tiwari, M. Laurenzano, J. Peraza, L. Carrington, and A Snavely. Green Queue: Customized Large-Scale Clock Frequency Scaling. In *Second International Conference on Cloud and Green Computing (CGC)*, pages 260–267, Nov 2012.

[100] J. Treibig, G. Hager, and G. Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of the First International Workshop on Parallel Software Tools and Tool Infrastructures*, PSTI, San Diego CA, 2010.

[101] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization*, 9(4):54:1–54:23, January 2013.

[102] Wei Wang, John Cavazos, and Allan Porterfield. Energy auto-tuning using the polyhedral approach. In Sanjay Rajopadhye and Sven Verdoolaege, editors, *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques*, pages 1–8, 2014.

[103] Wei Wang, H. Howie Huang, Matthew Kay, and John Cavazos. GPGPU accelerated cardiac arrhythmia simulations. In *Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 724–727, 2011.

[104] Wei Wang, Allan Porterfield, John Cavazos, and Sridutt Bhalachandra. Using per-loop CPU clock modulation for energy efficiency in OpenMP applications. In *International Conference on Parallel Processing*, ICPP, pages 629–638, Sept 2015.

[105] Wei Wang, Allan Porterfield, John Cavazos, and Sridutt Bhalachandra. Compiler transformations meet cpu clock modulation and power capping. In *the 5th International Workshop on Power-aware Algorithms, Systems, and Architectures*, PASA, 2016.

[106] Wei Wang, Lifan Xu, John Cavazos, Howie H. Huang, and Matthew Kay. Fast acceleration of 2D wave propagation simulations using modern computational accelerators. *PLoS ONE*, 9:1–10, January 2014.

[107] Michael Welter and Heiko Rieger. Interstitial fluid flow and drug delivery in vascularized tumors: A computational model. *PLoS ONE*, 8(8):e70395, August 2013.

[108] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. OpenACC: first experiences with real-world applications. In *The 18th International European Conference on Parallel and Distributed Computing*, Euro-Par, pages 859–870, Berlin, Heidelberg, 2012. Springer-Verlag.

[109] Tomofumi Yuki and Sanjay Rajopadhye. Folklore Confirmed: Compiling for Speed = Compiling for Energy. In *Proceedings of the 26th International Conference on Languages and Compilers for Parallel Computing*, LCPC, 2013.

[110] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Hardware Execution Throttling for Multi-core Resource Management. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX, pages 23–23, 2009.

[111] Zhiyue Zhang, Yuping Wang, and Quanxiang Wang. A characteristic centred finite difference method for a 2D air pollution model. *International Journal of Computer Mathematics*, 88(10):2178–2198, July 2011.