POWER AND ENERGY MANAGEMENT OF MODERN ARCHITECTURES IN ADAPTIVE HPC RUNTIME SYSTEMS

BY

EHSAN TOTONI

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2014

Urbana, Illinois

Doctoral Committee:

Professor Laxmikant V. Kalé, Chair
Professor Josep Torrellas
Professor María Jesús Garzarán
Doctor Fabrizio Petrini, IBM TJ Watson Research Center

Abstract

Power and energy efficiency are important challenges for the High Performance Computing (HPC) community. Excessive power consumption is a main limitation for further scaling of HPC systems, and researchers believe that current technology trends will not provide Exascale performance within a reasonable power budget in near future. Hardware innovations such as the proposed Exascale architectures and Near Threshold Computing are expected to improve power efficiency significantly, but more innovations are required in this domain to make Exascale possible.

To help shrink the power efficiency gap, we argue that adaptive runtime systems can be exploited. The runtime system (RTS) can save significant power, since it is aware of both the hardware properties and the application behavior. We use application-centric analysis of different architectures to design automatic adaptive RTS techniques that save significant power in different system components, only with minor hardware support.

In a nutshell, we analyze different modern architectures and common applications and illustrate that some system components such as caches and network links consume extensive power disproportionately for common HPC applications. We demonstrate how a large fraction of power consumed in caches and networks can be saved using our approach automatically. In these cases, the hardware support the RTS needs is the ability to turn off ways of set-associative caches and network links. We also present some required RTS techniques, such as recognizing the running application's pattern using pattern recognition to predict its future and adapt the hardware appropriately. Furthermore, we address two types of prevalent heterogeneity: utilization of accelerator devices and process variation. To study accelerators, we analyze and optimize an example application on a heterogeneous architecture and demonstrate techniques for efficient mapping on different devices (CPU and GPU). To address process variation challenges, we develop accurate models that let the RTS schedule efficiently in the presence speed and power consumption variation. Using the models,

we develop a novel scheduling framework that uses integer linear programming to enforce different performance and power consumption constraints.

To my loving family and friends, who helped me on this journey.

Acknowledgments

Ph.D. is a long and difficult journey, which one cannot possibly finish successfully without extensive support of others. I would like to thank my advisor, Prof. Kale, for letting me explore different research ideas. He was patient with me and gave me invaluable insights throughout my Ph.D. program. He also provided me the support I needed to achieve my goals.

University of Illinois is a great institution with world class faculty members. I profited from the advise and support of many other faculty members as well. For each of the research topics I pursued, I was able to work with a world-renowned expert of that field. I would like to thank Prof. Torrellas, Prof. Garzaran, and Prof. Heath for helping me in various projects. Their advice and support made finishing my research projects possible. Through my interactions with these experienced researchers, I learned how to approach a research problem, how to analyze the tradeoffs and find solutions, and how to present the results. I also need to thank Dr. Fabrizio Petrini (IBM Research) for providing me feedback, motivation, and insight about the big picture.

Parallel Programming Laboratory (PPL) is filled with many amazing students and staff. Their accomplishments and brilliant research ideas were constant sources of inspiration for me. Many ideas came out of our discussions, and they also assisted me in many other ways. I would like to thank Phil and Michael for proofreading my manuscripts and helping me prepare my presentations. Thanks to Nikhil for helping me solve technical problems and Esteban for motivating me. Also, thanks to Osman and Akhil for collaborating with me in power-related projects.

My life out of the lab was very important during these years. Champaign-Urbana's community, especially Iranians, welcomed me from the very first moment I arrived. I had an amazing time these years, learned a lot of things, and made some of my best friends. All in all, my life in Champaign was an unforgettable experience!

Last but not least, the emotional support from my family and friends was very important these years. In the most difficult times, I always felt loved and supported, which helped me carry on and continue to make progress. I will always be grateful to them.

Grants

This work was partially supported by the following sources:

- Illinois-Intel Parallelism Center (I2PC), which is supported by Intel Corporation, partially supported this research.
- This work was also partially supported by U.S. Department of Energy grant DE-SC0006706.
- This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

Table of Contents

List of Figures

List of Tables

List of Algorithms

CHAPTER 1

Introduction

Power and energy issues are increasingly important for computers at different scales. The number of transistors on a chip continues to increase as predicted by Moores law, but the chip's power and energy consumption do not scale as before. Previous generations of CMOS could scale down the voltage (known as "Denard scaling" [1]) resulting in much lower power and energy consumption. However, the voltage of the transistors cannot be scaled much further easily because of the physical limitations. Therefore, the limited power budget of the processors should be used most efficiently.

Power and energy limitations have a direct impact on science and engineering applications in High-Performance Computing (HPC) environments. The HPC community is aiming to keep the total power intake of future Exascale machines (the next generation of faster computers) at tens of mega watts (MW), whereas current systems with only 10 PetaFLOPS of performance are already consuming over 10 MW of power. Multiple innovations must be developed to dramatically reduce the total power usage of supercomputers.

Realizing Exascale systems is mostly limited by power and energy consumption issues, and new design approaches are needed. Exascale systems are required to deliver 100-1000x higher performance compared to today's Petascale machines, but stay within similar power envelope. Various analyses in the literature, such as the comprehensive Exascale Computing Study [2], conclude that Exascale cannot be reached by following the current trends of energy efficiency. Hence, much more energy-efficient approaches are needed for the design of computer systems, potentially changing multiple layers, from hardware to applications.

There are various efforts to improve hardware's power efficiency towards Exascale, such as new architectures that are proposed in order to fulfill this objective [3, 4]. To reduce various overheads, they include a large number of relatively simple processor cores rather than fewer complex cores. Simple cores do not have the overheads of power consuming features of

heavy cores, such as out-of-order execution and speculation. In essence, simple cores use the available power to execute useful operations, rather than spending power for scheduling instructions (e.g. out-of-order execution). However, they require more parallelism in the application to attain the same performance levels. In addition, Exascale architectures use heterogeneity to map different tasks to best matching processor type. For example, large latency-optimized cores are used for operating system and runtime, while simple cores are used for the applications calculations. Furthermore, these architectures strive to improve the energy efficiency of memory systems. For example, they do not provide full-chip cache coherence to avoid its overheads.

One major direction for power efficiency improvement is low voltage operation [5, 6]. The reason is that decreasing the supply voltage (V_{dd}) reduces dynamic power quadratically and also reduces static power. Researchers believe that the highest energy-efficiency can be obtained when the supply voltage is only slightly higher than the transistor's threshold voltage [7]. For current technologies, it roughly corresponds to V_{dd} of 0.5V, rather than the conventional 1V value. This regime is called Near-Threshold Computing (NTC). NTC can potentially decrease the chip's power consumption more than 40 times. However, low voltage operation slows down the processors significantly, brings more reliability challenges, and increases the effects of process variation. Therefore, circuits and devices need to be redesigned for low voltage operation.

Although these hardware innovation are expected to be significant, they are not enough to attain the power efficiency required for Exascale, based on the comprehensive Exascale report analysis [2]. Therefore, more innovations are required that are potentially software-centric. Among the software layers, the runtime system seems very promising comparatively. The runtime system is aware of both the running application and the hardware. Targeting the runtime most often avoids changing the application code, which is harder to change. It also avoids drastic changes and overheads in the hardware.

An HPC Runtime system (RTS) provides many services for the running application, and has great potential for power and energy management as well. The use of a runtime system for management of the application's communication and other parallel services is well known [8, 9]. However, research has demonstrated the capabilities of runtime systems for other features such as load balancing [10], fault tolerance [11], and power management as well [12, 13]. We take advantage of an introspective software component that is aware of both the current hardware status as well as the application evolution.

In this research, we demonstrate that an adaptive runtime system can improve the power efficiency of HPC systems without significant changes in the application, and only with minor hardware support. It can do so by monitoring both the application and the hardware,

and adapting to their properties. We only change the runtime layer of the system, since it is arguably the easiest to change compared to the hardware and the application layers. Our runtime studies follow a common theme. First, we exploit the capabilities of adaptive runtime systems as they are powerful tools that observe both the hardware status and the application behavior. Second, we analyze the relevant characteristics of common HPC applications and exploit them in our methods. Third, we propose small necessary hardware support, considering the properties and opportunities in modern hardware.

Avoiding drastic changes in the hardware and application is an important advantage of our approach. The approaches for power and energy efficiency can either make dramatic (revolutionary) hardware changes or incremental (evolutionary) ones. It is often argued that drastic revolutionary changes are required at this stage. However, small incremental changes had been more common so far because they are easier to sustain. Revolutionary changes for the benefit of only one domain (e.g. HPC) can potentially impair us from using commodity pieces, among other problems. For example, many large-scale supercomputers use commodity processors mainly designed for servers. Furthermore, even when the processors themselves are different, often times the same architecture design is used across various scales by just varying some parameters. For instance, Intel's Haswell microarchitecture [14] is used in processors from low power mobile devices to high-performance servers. Various instances of the architecture have different parameters such as the number of cores and the frequency range, but the underlying design is almost the same. Thus, incremental changes that allow the use of commodity hardware are highly preferred.

We argue that the same commodity processor can be used for more energy-efficient HPC machines as well, often with minor HPC-specific support. Thus, we strive to improve the power and energy efficiency of HPC machines while keeping them economically viable. Although we develop our methods mainly for HPC systems, they can be adopted for other domains as well (see Chapter 3 for an example).

Our proposals for power and energy efficiency are orthogonal with the mentioned hardware innovations and can be utilized at the same time. In some cases, our methods complement the hardware ones. For example, our variation-aware scheduling framework (see Chapter 6) addresses some of the challenges of process variation caused by low voltage operation.

Overall, we follow a cross-layer approach that focuses on the characteristics of HPC applications, capabilities of the adaptive runtime system, and the opportunities in different components (e.g. caches) of modern hardware. First, we analyze different architectural designs using multiple applications, focusing on the opportunities of heterogeneous and many-core architectures. Second, we study the opportunities and challenges (e.g. programmability) of heterogeneous architectures and propose energy efficient runtime methods to exploit on-chip

heterogeneity. Third, we strive to improve the energy efficiency of processor caches using adaptive runtime systems, since they consume a large fraction of processors power. Forth, we propose a similar adaptive runtime system based approach to improve the energy efficiency of large-scale networks. Fifth, we tackle the challenges of semiconductor process variation by intelligent runtime scheduling. These studies complement each other, and provide insight for better power and energy efficiency in different systems.

1.1 Dissertation Overview

This dissertation is organized as follows. Chapter 2 evaluates and analyzes different parallel architectures, such as heterogenous and many-core architectures. We use Intel Single-Chip Cloud Computer (SCC) to analyze many-core architectures and compare them to others. The analysis criteria in this study are speed, power, energy, programmability, and application portability. The results show that many-cores represent a balanced point in terms of these metrics, since they can run existing code fast with low power. On the other hand, we show that heterogeneous architectures can be exceptionally superior for some applications, but they are difficult to program. We study them in more detail in the subsequent chapter. In addition, we study one important challenge for many-cores, namely process variation, in Chapter 6. In general, we use the insights of this chapter to tackle various power and efficiency challenges in later chapters.

In Chapter 3, we study the energy efficiency opportunities and challenges of heterogeneous architectures that were identified in the previous chapter. The opportunities include mapping different kernels of the running application to the more efficient device (CPU or GPU in this case) for that kernel. On the other hand, programmability is a main challenge, for example. To study the issues, we optimize and analyze a vision application on a heterogeneous on-chip architecture and improve its energy efficiency. This mobile application is very similar to large-scale HPC applications in various aspects, and hence, our results are extensible to other domains. We demonstrate that a unified programming paradigm such as OpenCL can increase programmer productivity since the same or similar codes can run on both devices. In addition, the results suggest that trying to get the maximum utilization of the heterogeneous platform naively is not efficient, and one has to map the kernels carefully. Since a fixed power budget is shared among devices, techniques such as software pipelining that map each kernel to the best device, while increasing utilization can be the most effective.

We continue by focusing on the cache hierarchy in Chapter 4, since caches consume a large fraction a processors power budget. We analyze the memory access patterns of common HPC

applications and demonstrate why many of these applications cannot take advantage of all the cache capacity effectively in many cases. On the hardware side, since the caches are usually set-associative, they are already partitioned. Therefore, one can turn some of the *ways* of set-associative caches off to save significant power. We introduce a runtime system based method to reconfigure caches in this manner adaptively. In the context of this study, we illustrate how an HPC application's pattern can be identified and expressed using formal language theory. This helps the runtime system predict the future of the application and adapt effectively. In addition, we introduce a software-controlled streaming method that is controlled by the runtime system to adapt to the applications pattern. Overall, our approach makes the cache hierarchy more power-efficient for HPC applications.

We apply the same methodology we used for caches to improve power-efficiency of the large-scale networks in Chapter 5. Similar to caches, networks consume substantial power disproportionately, even when the running applications do not need it. We show that common HPC applications do not use a large fraction of the network links in modern high-radix topologies. In this case, the hardware property that we exploit for power efficiency is that some network links can be turned off. Therefore, we develop a runtime system based method to improve the power efficiency of large-scale HPC networks by turning links on and off adaptively. Note that we use the previous chapter's ideas to design our runtime techniques for a different component of the system, which shows the generality of our adaptive runtime system approach.

We return to many-core architectures in Chapter 6 and study their inherent heterogeneity, which is caused by process variation. This problem is different than previous ones, and therefore, it needs different runtime techniques. Because of process variation, different cores have different power and performance characteristics. The runtime needs to choose a configuration (a set of cores) that delivers the best performance while staying within the power budget. The number of configurations is exponential in the number of cores and hence, sampling all of the configurations is not possible. To solve this problem, we develop models that let the runtime system predict the performance and power consumption of different configurations of cores. Using these models, we develop a novel scheduling framework based on integer linear programming that lets the runtime enforce different performance and power constraints effectively. For example, the runtime can choose the set of cores that provide the best performance under a certain power budget. This runtime scheduling framework is also capable of enforcing other constraints, such as task mapping constraints for better communication performance.

Finally, we summarize our contributions in Chapter 7. This chapter also includes the possible future research directions based on this dissertation's insights.

CHAPTER 2

Performance, Power, and Energy Evaluation of Modern Architectures .

Following Moore's law, the number of transistors that can be placed on a chip keeps increasing rapidly with each technology generation. Not surprisingly, users expect the performance to also improve over time with the increase in the number of transistors. However, performance depends on multiple factors beyond the transistor count.

The architecture community has historically tried to turn the higher transistor count into higher performance. For example, during the single-thread era, excess transistors were used for architectural features such as pipelining, branch prediction, or out-of-order execution. These features were able to improve performance while largely keeping the application unchanged.

However, in the last several years, some key technology parameters such as the supply voltage have stopped scaling. This has led to chips with unsustainably-high power, power density and energy consumption. This fact combined with the diminishing returns from single-thread architectural improvements, has pushed forward thread parallelism as the only solution to make effective use of the large number of transistors available. The result has been a paradigm shift toward parallelism.

A key architectural challenge now is how to support increasing parallelism and scale performance, while being power and energy efficient. There are multiple options on the table, namely "heavy-weight" multi-cores (such as general purpose processors), "light-weight" many-cores (such as Intel's Single-Chip Cloud Computer (SCC) [15]), low-power processors (such as embedded processors), and SIMD-like highly-parallel architectures (such as General-Purpose Graphics Processing Units (GPGPUs)).

The Intel SCC [15] is a research chip made by Intel Labs to explore future many-core

architectures. It has 48 Pentium (P54C) cores in 24 tiles of two cores each. The tiles are connected by a four by six mesh in the chip. The SCC naturally supports the message passing programming model, as it is not cache-coherent in hardware. We have ported Charm++ [16] and Adaptive MPI (AMPI) [16] to this platform to be able to run existing sophisticated applications without any change.

The goal of this chapter is to explore various trade-offs between the SCC and the other types of processors. We use five applications to study their power and performance: NAMD, Jacobi, NQueens, Sort and CG (conjugate gradient). These applications exhibit different characteristics in terms of both computation and communication. The processors used are the Intel SCC as a light-weight many-core, the Intel Core i7 as a heavy-weight multi-core, the Intel Atom as a low-power processor, and the Nvidia ION2 as a GPGPU. These processors represent different cutting-edge architectures. To compare these architectures, the applications are executed with the same input parameters and we measure speed, power, and energy consumption.

Our results show that each of the designs is effective in some metric or condition and there is no single best solution. For example, the GPGPU provides a significant advantage in terms of power, speed and energy in many cases, but its architecture is not general enough to fit all the applications efficiently. In addition, the GPGPU requires significant programming effort to achieve this efficiency (as we had to use different codes to run on the GPGPU) and cannot run legacy codes.

The Intel SCC results suggest that light-weight many-cores are an opportunity for the future. The SCC has lower power than the heavy-weight multi-core and runs faster than the low-power design. Also, the light-weight many-core is general enough to run legacy code and is easy to program (in contrast to the GPGPU). However, some weaknesses of the platform should be addressed in future designs to make it competitive with sophisticated multi-cores. One such weakness that we identified is slow floating-point performance.

This chapter also proves that the low-power processor does not necessarily result in less energy consumption. As shown by our data on the Intel Atom platform, the extra delay has a greater effect than the power savings achieved.

The rest of this chapter is organized as follows. Section 2.1 describes the architecture of the platforms that we study. Section 2.2 briefly introduces the applications, as their characteristics are very important to understand their scaling and power consumption on different platforms. Section 2.3 evaluates the platforms using the applications. We compare the architectures in Section 2.4 using the results of the previous section and analyze the tradeoffs of each one. We discuss the related work in Section 2.5 and conclude in Section 2.6.

2.1 Platforms

Here we describe the platforms that we evaluate, with a focus on their design concept and level of parallelism. Among these platforms, the SCC is a research chip while the others are examples of commodity platforms, which are being widely used in different machines.

2.1.1 Intel Single-chip Cloud Computer

The "Single-Chip Cloud Computer" (SCC) is Intel's new research many-core architecture. It has 48 Pentium cores connected through a mesh interconnect. It has been created by Intel Labs to facilitate software research on future many-core platforms. This chip is not cache coherent and it naturally supports the message passing parallel programming paradigm.

Figure 2.1 shows the architecture overview of the SCC [15]. The cores are arranged in groups of two in 24 tiles. The tiles are connected in a four by six mesh configuration. The cores are simple second-generation off-the-shelf Pentium cores (P54C). Each core has 16KB L1 data and 16KB L1 instruction caches as well as a 256KB unified L2 cache. Each tile has a 16KB SRAM called Message Passing Buffer (MPB), which is used for communication inside the chip. These MPBs form a shared address space used for data exchange. The cores and MPB of a tile are connected to a router by Mesh Interface (I/F) unit. The SCC also has four DDR3 memory controllers in the four corners of the mesh network to connect cores to memory.

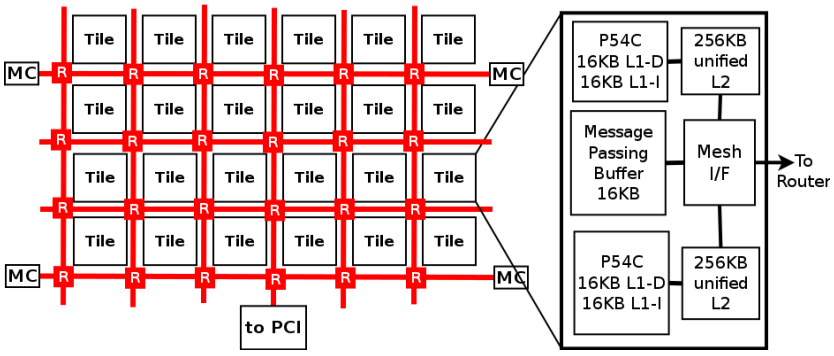


Figure 2.1: Architecture overview of the SCC.

The SCC is implemented in 45nm CMOS technology and has 1.3 billion transistors. The area of each tile is 18 mm^2 with a total die area of 567 mm^2. Power for the full chip ranges from 25W to 125W. It consumes 25W at 0.7V, with 125MHz cores, 250MHz mesh, and 50°C. It consumes 125W at 1.14V, with 1GHz cores, 2GHz mesh, and 50°C. Power for the on-die network is 6W for a 1.5 Tb/s bisection bandwidth and 12 W for a 2 Tb/s bisection

bandwidth. Figure 2.2 shows the power breakdown of the chip in two different modes: full power and low power [17].

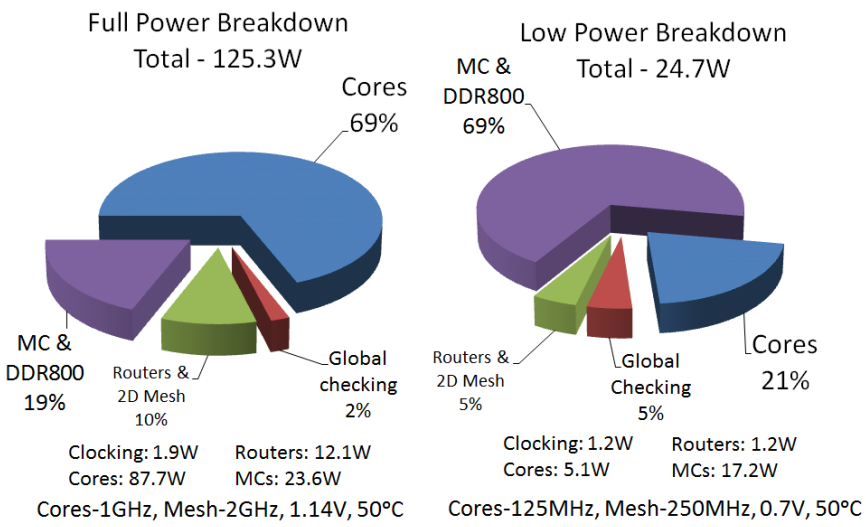


Figure 2.2: Power breakdown of the SCC in full-power and low-power mode (from J. Howard et al. [17]).

The SCC was designed with power management in mind. It includes instructions that let programmers control voltage and frequency. There are 8 voltage domains on a chip: one for the memory controllers, one for the mesh, and six to control voltage for the tiles (at the granularity of 4-tile blocks). Frequency is controllable at the granularity of an individual tile, with a separate setting for the mesh, thereby providing a total of 25 distinct frequency domains. The RCCE library [18] provides easy access to these features, but it is more limited than the available hardware features [17].

As mentioned, message passing is the natural way of programming this non-cache-coherent chip. For this purpose, there is a low level message passing interface called RCCE that provides low level access to the communication features of the SCC. It was designed so that the chip can still operate without any operating system ("bare metal mode") to reduce the overheads [18]. However, Linux can also be run on the SCC, which is the most common usage, and we ran our experiments in this mode. In addition, there is another interface called Rckmb, which provides the data link layer for running network services such as TCP/IP. We used the latter to port Charm++ and run existing applications. Using other layers to port Charm++ could result in some communication performance improvement; however, it would not change our conclusions.

Porting Charm++ and the applications did not involve any conceptual difficulty as the SCC can be easily viewed as a cluster on a chip. However, there are many technical issues involved in working with it. These include dealing with old compilers, an old operating

system, and unavailability of some standard software and libraries. Unfortunately, we could not perform many of the intended experiments because of these technical issues and others took much more time than expected.

2.1.2 Other Platforms

Intel Core i7 Processor The Intel Core i7 is a 64-bit x86-64 processor. We have used the Core i7 860 Nehalem processor chip, which has four CPU cores and on-chip cache memory on one 45nm die. Hyperthreading support allows it to appear to the OS as eight processing elements. The cores cycle at 2.8GHz (disregarding Turbo Mode), which is a relatively high frequency. Each of the four cores has 32KB instruction and 32KB data Level 1 caches, and 256KB of Level 2 cache. The four cores share an inclusive 8MB Level 3 cache. The specification of the Intel Core i7 is shown in Table 2.1.

Table 2.1: Intel Core i7 processor specifications.

Processor Number	i7-860
# of Cores	4
# of Threads	8
Clock Speed	2.8 GHz
Cache Size	8 MB
Lithography	45 nm
Max TDP	95W
VID Voltage Range	0.65V-1.40V
Processing Die Size	296 mm^2
# of Processing Transistors on Die	774 million

Intel Atom D525 The Intel Atom is the ultra-low-voltage x86-64 CPU series from Intel. It is designed in 45 nm CMOS and used mainly in low power and mobile devices. Hyper-threading is also supported in this processor. However, there is no instruction reordering, speculative execution or register renaming.

Due to its modest 1.8 GHz clock speed, even the fastest Atom D525 is still much slower than any desktop processor. The main reason behind our selection of the Atom for our experiments is that, while desktop chips have a higher frequency, the Atom is hard to beat when it comes to power consumption. Atom allows manufacturers to create low-power systems. However, low power does not always translate into high efficiency, meaning that

Atom may have low performance per watt consumed. We have explored this issue in our experiments.

The specification of the Intel Atom processor that we used is shown in Table 2.2.

Table 2.2: Intel Atom D525 processor specifications.

Processor Number	D525
# of Cores	2
# of Threads	4
Clock Speed	1.80 GHz
Cache Size	512 KB
Lithography	45 nm
Max TDP	13W
VID Voltage Range	0.800V-1.175V
Processing Die Size	87 mm^2
# of Processing Transistors on Die	176 million

Nvidia ION2 Platform Nvidia ION is a system/motherboard platform that includes Nvidia's GPU, DDR3 or DDR2 SDRAM, and the Intel Atom processor. The Nvidia ION2 has a dedicated graphics card for the new Atom CPUs. The ION2 is based on the GT218 chip (GeForce 305M, 310M) with dedicated memory (compared to the old ION that was a chipset graphics card). ION2 systems can use CUDA (Nvidia's General-Purpose Computing on Graphics Processing Units technology) as well as OpenCL (Open Computing Language), to exploit the parallelism offered by the CPU and the GPU together. This platform is used in low-power devices, and yet is equipped with a GPU. Hence it has the potential to offer great benefits in performance and power at the same time. We have used a 12" ION2 Pinetrail netbook platform for our experiments.

The specification of the CPU was mentioned in Table 2.2. The specification of the graphics processor is shown in Table 2.3.

2.2 Applications

The characteristics of the applications are important to understand their different behavior and derive conclusions about the architectures. In this section, we describe the applications we used to examine the different parallel architectures. We choose scalable parallel applications that use Charm++ [16] or MPI message passing paradigms. For the GPU platform,

Table 2.3: NVIDIA ION2 graphics card specifications.

ION Series	ION2
GPU Number	GT218
# of CUDA Cores	16
Clock Speed	475 MHz
Memory	256 MB
Memory bus width	64-bit
Power consumption	12W

we use appropriate versions of the applications based on the OpenCL or CUDA models. The benchmarks are reasonably optimized but not highly optimized for any particular architecture. These applications represent different classes of programs with different characteristics to stress the platforms.

Iterative Jacobi The Jacobi calculation is a useful benchmark that is widely used to evaluate many platforms and programming strategies. A data set (2D array of values in our case) is divided among processors and is updated in an iterative process until a condition is met. The communication is mostly a nearest neighbor exchange of values. An OpenCL implementation was used for ION2, while a Charm++ version was used for the other platforms.

We selected Jacobi in our experiments since it is representative of stencil computations, which are widely used in scientific programs.

NAMD NAMD is a highly scalable application for Molecular Dynamics simulations [19]. It uses hybrid spatial and force decomposition to simulate large molecular systems with high performance. It is written in Charm++ to exploit its benefits such as portability, adaptivity and dynamic load balancing. It typically has a local neighbors communication pattern without bulk synchronization and benefits from communication and computation overlap. It also tries to keep its memory footprint small in order to utilize caches and achieve better memory performance. We chose this application to represent dynamic and complicated scientific applications. We used the ApoA1 system as input, which has 92,224 atoms.

NQueens The NQueens puzzle is the problem of placing n chess queens on an $n \times n$ chessboard so that no two queens attack each other. The program will find the number

of unique solutions in which such valid placement of queens is possible. This problem is a classic example of the state space search problems. Since NQueens is an all-solutions problem, the entire solution tree needs to be explored. Thus, this problem also presents a great opportunity for parallelization with minimum communication.

We selected this problem since it is an integer program, as opposed to the other floating-point problems, and it represents state space search applications.

CG Conjugate Gradient is one of the NAS Parallel Benchmarks (NPB) [20], which are widely used to evaluate the performance of different parallel systems. CG is an iterative method and involves lots of communication and data movement. We chose it to stress the communication capabilities of our platforms. Unfortunately, we did not have a version of it available on our GPU system.

Integer Sort Parallel sorting is a widely used kernel to benchmark parallel systems because it represents commercial workloads with few computation and heavy communication. It does not have the massive floating point computations and regular communication patterns (such as nearest neighbors) of many scientific workloads. We use a straightforward implementation of Radix Sort in OpenCL and a similar Charm++ version.

Table 2.4 shows some performance counter values for the applications. They are obtained using the PAPI library running on the Core i7 system. The numbers are normalized with respect to the number of dynamic instructions executed. Since the applications are already well-known, these few numbers give enough insight to help explain the results.

Table 2.4: Performance counters for the applications.

	Jacobi	NAMD	NQueens	CG	Sort
L1 Data Misses	0.0053	0.0053	0.0003	0.0698	0.0066
Cond. Branches	0.0205	0.0899	0.1073	0.1208	0.0556
Float-pt operations	0.0430	0.3375	0.0004	0.2078	0.0001

2.3 Evaluation Results

We now run the applications on the proposed platforms and measure the scaling behavior and the power and energy consumption. The data provides insight into the effectiveness of different architectural approaches. We focus on the ability of the applications to exploit the

parallelism of the platform and how using more parallelism will affect power and energy consumption. We connected power meters to the whole systems (rather than just the processor chips) to measure the power consumption.

2.3.1 Intel SCC

We have ported Charm++ to the SCC using the network layer (TCP/IP) provided by Rckmb. Thus, Charm++ programs can run on the SCC hardware without any change to the source code. This simplifies the porting of software significantly. We used the highest performance options available in the SCC software toolkit (1.3) to run the experiments and characterize the system. Table 2.5 shows the SCC configuration used in the experiments.

Table 2.5: SCC configuration used in the experiments.

Operating mode	Linux
Communication mechanism	TCP/IP
Tile frequency	800 MHz
Mesh frequency	1600 MHz
Memory controller frequency	1066 MHz
Supply Voltage	1.1 V
Idle power	39.25 W

Figure 2.3 shows the speedup of the five applications on different numbers of cores. It can be observed that using more cores improves performance, and that all the applications except CG are scalable on this platform. In addition, these Charm++ applications are scalable without any change to the source code. In Jacobi, the execution time per step on one core is $7.87s$, and on the full machine is $0.32s$. This corresponds to a 24.6 speedup on 48 cores. NQueens takes $568.36s$ to execute on one core and $19.66s$ on 48 cores. The resulting speedup is 28, which is even higher than Jacobi. In NAMD, the time per step is $34.96s$ on one core and $1.23s$ on all the cores. This also corresponds to a speedup of 28 on 48 cores. Thus, NAMD is an example of a full-fledged application that can use many-core architectures effectively.

Sort is the most scalable application, with a 32.7 speedup. This shows that the network can handle the communication effectively.

On the other hand, CG is the worst-scaling application, with a speedup of just 4.91 on 32 cores. We could not run it on more cores because this application requires the number of cores to be a power of two. The reason why CG is not scalable is the fine-grain global

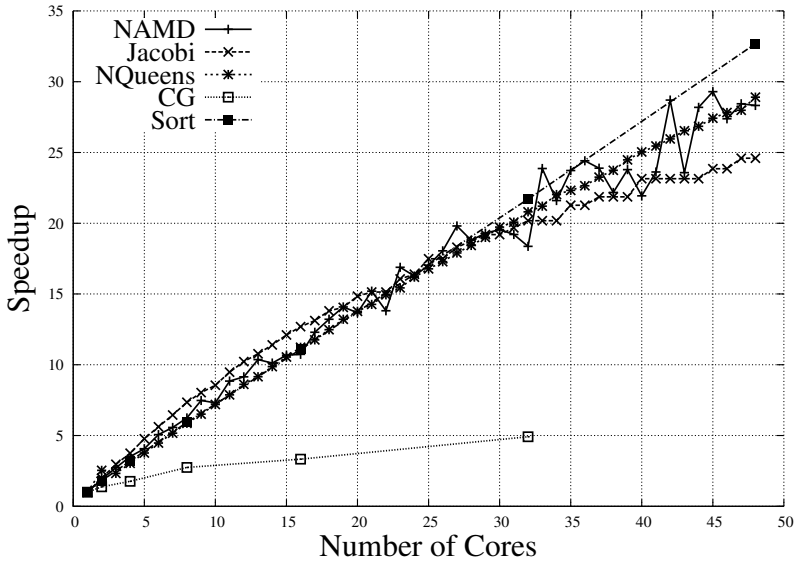


Figure 2.3: Speedup of the applications on different numbers of SCC cores.

communication present in the algorithm. For example, there is a global reduction after each phase. Specifically, the performance counter data of Table 2.4 shows that CG has a high number of L1 cache misses on Core i7. Since the problem size is small and fits in the cache, the misses are caused by high communication. Thus, the SCC communication system (network and software) may not be suitable for this application. Optimizing the network for global communication such as by adding a collectives network can help significantly in this case. Also, tuning the runtime system such as by tuning the collective algorithms may help.

As indicated above, CG requires a power-of-two number of cores. This is an important consideration when designing many-cores, since some programmers may assume the number of cores to be a power of two for simplicity. A promising solution is to use virtualization, and use any number of virtual processors that the application needs on the available physical processors. This feature is available in Charm++ but we do not evaluate it here.

Finally, we ran four of the applications on every possible number of cores – from 1 to 48. The speedups are consistent in all the applications. In NAMD, there is some noise due to the application's dynamic and adaptive behavior.

Figure 2.4 shows the power consumption of the platform using different numbers of cores. These values are the maximum values seen during the run time of each individual application. For Jacobi, the power goes from 39.58W using just one core to 73.18W using all the 48 cores. NAMD's power consumption is similar to Jacobi. NQueens consumes more power, and goes up to 85.5W.

CG and Parallel Sort consume less power compared to the other applications. This is

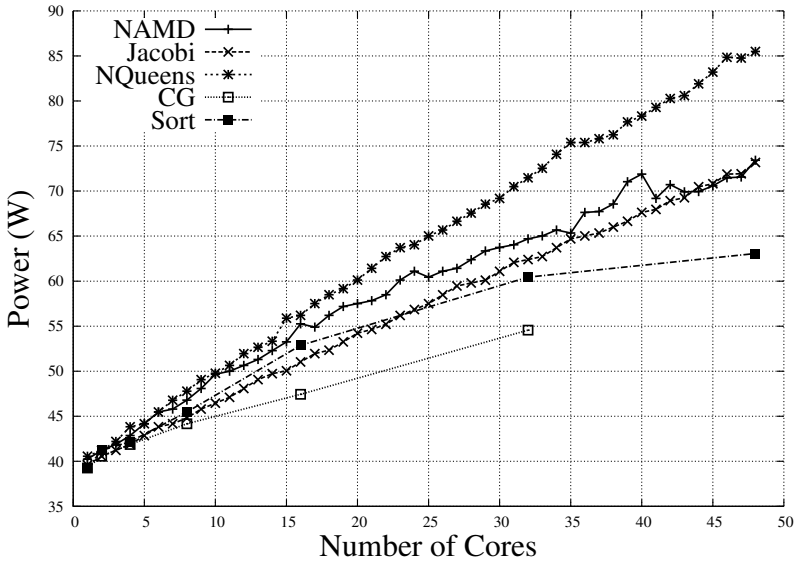


Figure 2.4: Power consumption of the applications on different numbers of SCC cores.

mainly because these applications are communication-bound and processors often stall, waiting for the communications to be completed. CG consumes the least power because it has the most stall time.

Figure 2.5 shows the energy consumed by each application, which is the power multiplied by time of execution. The energy of each application with a given number of cores is normalized to the application's energy using all the cores. This allows us to compare the applications. The general trend shows that using more cores to run the applications results in less energy consumption on the SCC. This is because the performance improvement attained by the added cores is higher than the power increases. Note that on the left side of Figure 2.5 there is a large reduction in energy as we use more cores. This is because the idle power of the system is high compared to the power added by adding one core, while the execution time decreases notably. For example, when going from one to two cores, the execution time drops to nearly half, while the power difference is small. Therefore, the energy consumption drops to nearly half.

The highest energy drop is for Sort and the lowest is for CG. This is because Sort is scalable, and the time savings is more significant than the power added by using more cores. Conversely, the inferior scalability of CG results in a small drop in energy consumption. Again, one should keep in mind the idle power when analyzing power and energy consumption because it offsets the power increase with more cores, and the time savings become more important.

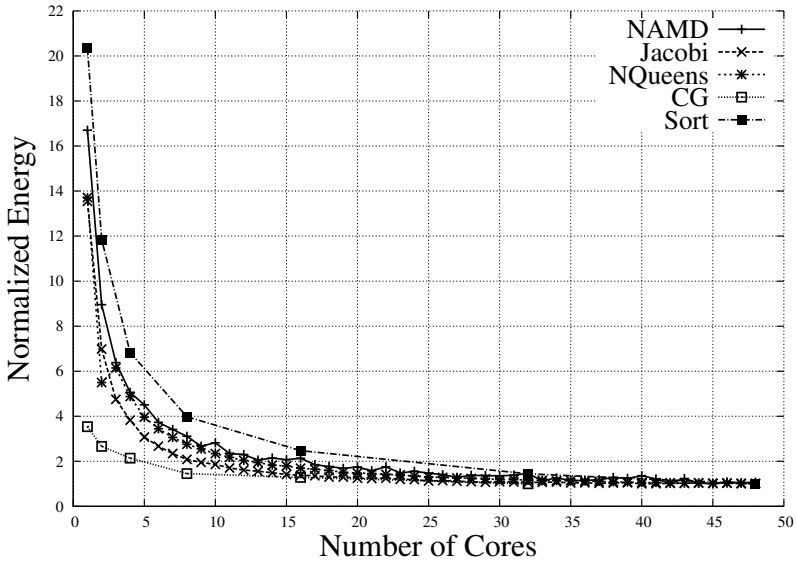


Figure 2.5: Energy consumption of the applications on different numbers of SCC cores. The data is normalized to the energy consumed with all the cores.

2.3.2 Intel Core i7 Processor

We have used the Charm++ infrastructure for the Intel x86_64 platform. In addition, we have used the same Jacobi, NAMD, NQueens, and Sort programs written in Charm++ and the same CG written in MPI as the ones we ran on the SCC. Since the Intel Core i7 is a quad core processor with hyperthreading, we can run up to 8 threads.

Figure 2.6 shows the speedup of these applications on different numbers of threads. By increasing the number of threads, we initially observe good speedups. Note the reduction in speedup when 5 threads are used as opposed to 4 threads. This is probably because at least two threads have to share the resources of one core. Therefore, they become slower and slow down the whole application. However, with increased parallelism, the application becomes faster and the slowdown is compensated.

Figure 2.7 shows the power consumption of the Core i7 platform using different numbers of threads. As expected, the power used by this processor increases as the number of threads increases. However, the increase is much higher than in other platforms. The power range of the Core i7 system is from around 51W for the idle power up to 150W, which is much higher than the SCC power.

Figure 2.8 presents the energy consumed by each application, which is the power multiplied by the execution time. As in the case of the SCC, the energy is normalized to the energy consumed when running all the threads. The general trend is that with more cores, the energy consumption goes down. Again, as we saw in the case of speedup, when using 5

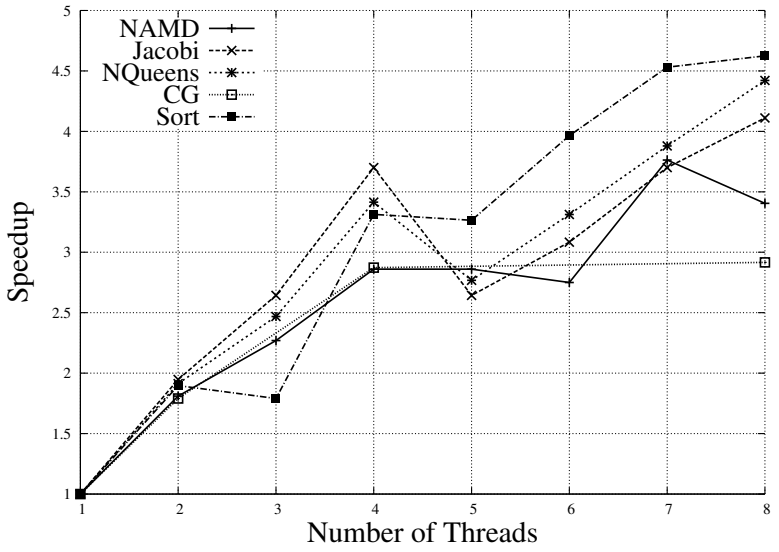


Figure 2.6: Speedup of the applications on different numbers of threads in the Intel Core i7.

threads, we had some increase in the runtime and, therefore, in energy consumption. Note that the reduction in energy is not as large as in the SCC case.

2.3.3 Intel Atom D525

The Atom D525 is a dual core processor in which each core is 2-way hyper-threaded. All the 4 threads can execute independently. Hence we can specify up to 4 processors to Charm++. In Figure 2.9, we observe good speedups with the increase in the number of threads for several programs.

Figure 2.10 shows that the Atom system consumes much less power than the other platforms. The increase in power per thread added is less than 1 W. Since the idle power of the entire system is about 27.5 W, the power increase due to having all four threads active is about 15% of the idle power.

In Figure 2.11, we observe that using more threads leads to less energy consumption, like in the other platforms. The power used by the Atom increases with an increase in the number of threads. However, the increase is not as rapid as in the Core i7 because of Atom's simpler architecture. At the same time, execution time reduces considerably with more threads, and hence the energy consumption decreases.

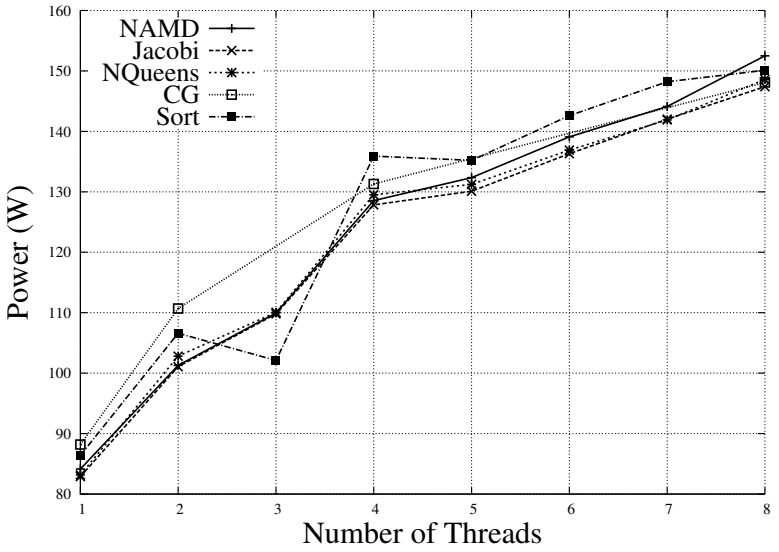


Figure 2.7: Power consumption of the applications on different numbers of threads in the Intel Core i7.

2.3.4 Nvidia ION2 Platform

We leverage the parallel computing power of the Nvidia ION2 platform using OpenCL for Jacobi, NQueens and Sort, and CUDA for NAMD. Unfortunately, we could not get a reasonably tuned version of CG. GPUs are very effective at data parallel applications due to the high number of simplistic SIMD compute units available in them. However, because the GPU is a coprocessor on a separate PCI-Express card, data must first be explicitly copied from the system memory to the memory on the GPU board. In general, applications which require a large amount of computation per data element and/or make full use of the wide memory interface are well suited to the GPU programming model.

Figure 2.12 shows data on the speed, power, and energy of the applications running on the ION2 platform. We use all 16 CUDA cores in the GPU, and do not change the parallelism of the applications because there would not be a significant difference in power. The speed bars show the speedup of the applications running on the ION2 platform relative to running on the Atom platform with the maximum number of threads. We see that the speedup is 22.65 for Jacobi and 16.68 for NQueens. While these are high speedups, given the number of compute cores available in the GPU, we would have expected higher speedups. The copying of device memory buffers at the end of each iteration forms the bottleneck in this computation. Note also that NAMD is no faster on ION2 than on Atom. NAMD has had scalable results on GPGPUs and we believe that, with careful tuning, it is possible to achieve better results. However, the effort was estimated to be high. Finally, Sort is 1.76 times slower than all the

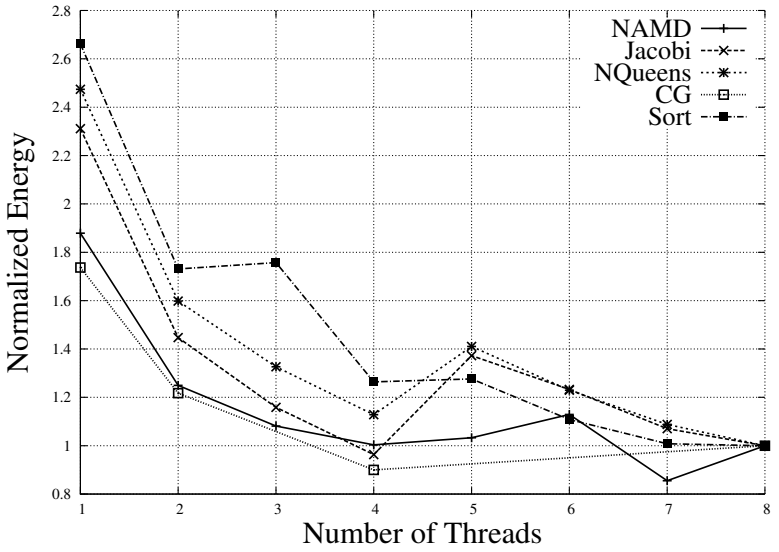


Figure 2.8: Energy consumption of the applications on different numbers of threads in the Intel Core i7. The data is normalized to the energy consumed with all the threads.

threads on the Atom. Thus, Sort is not suitable to run on GPGPUs.

The power bars show the absolute power consumption of the ION2 platform in W using all the 16 CUDA cores. We see that the power consumption ranges from 25 W to 35 W. These numbers are much lower than those of the Core i7. Note that we have not performed our experiments on more powerful GPUs, which are expected to provide better performance, albeit at the cost of some more power consumption. In any case, we do not expect the power consumption to rise as high as a heavy-weight processor like the Core i7.

The final bars show, for each application, the energy consumed by the ION2 normalized to the energy consumed by the Atom. We normalize the energies to the Atom numbers to be able to compare the energy of the different applications — otherwise, long-running applications would dwarf short-running ones. For these bars, we use the Y axis on the right side. Overall, from the figure, we see the ION2 is more energy-efficient than the Atom for Jacobi and NQueens; the opposite is true for NAMD and Sort.

2.3.5 Load Balancing

As the number of cores per chip increases, load balancing becomes more important (and challenging) for efficient use of the available processing power. Here, we investigate the effectiveness of dynamic load balancing on the SCC (with 48 threads) compared to the Core i7 (with 8 threads). We use *LBTest*, which is a benchmark in the Charm++ distribution,

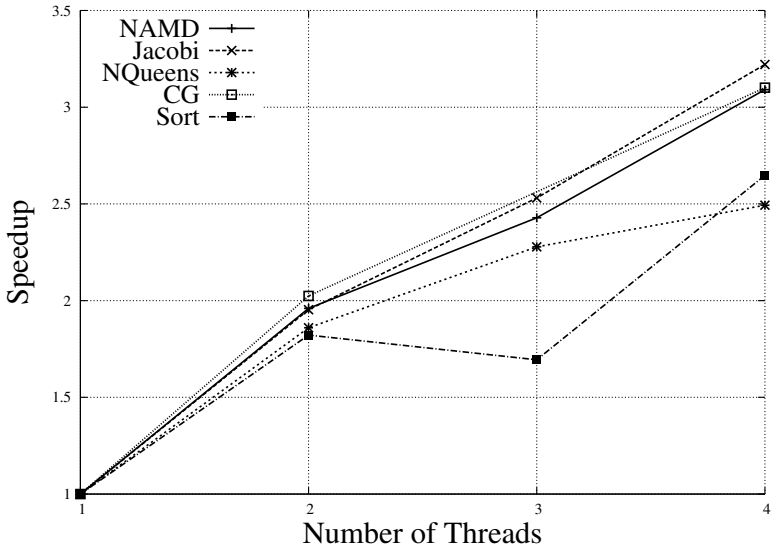


Figure 2.9: Speedup of the applications on different numbers of threads in the Atom processor.

with *RefineLB* as the balancer. LBTest creates a 3D mesh graph where the nodes have objects that perform random computation. Each object also sends a message (of a size that can be specified) to one of its neighbors randomly. In our case, the size of the messages is very small compared to the computational load, so communication does not matter much. RefineLB is a simple load-balancing strategy that tries to balance the load by gradually removing objects from overloaded threads.

Figure 2.13 shows the utilization of each of the threads in the SCC and Core i7 platforms before and after applying the load balancer. The figure is obtained using Projections [21], which is a performance visualization tool in the Charm++ infrastructure. From the figure, we can see that the load balancer perfectly balances the load on the Core i7 platform and somewhat improves the balance on the SCC platform. On average, it increases the average thread utilization from 35% to 50% in the SCC, and from 59% to 99% in the Core i7. Load balancing is more difficult in a platform like SCC, which has many cores. The load balancer has increased the average utilization of the SCC cores significantly, but at 50% average utilization, it is clear that more effective methods are needed in the future. Overall, it can be shown that the load balancer improves the performance of the benchmark (with the same input parameters) by 30% in the SCC and by 45% on the Core i7.

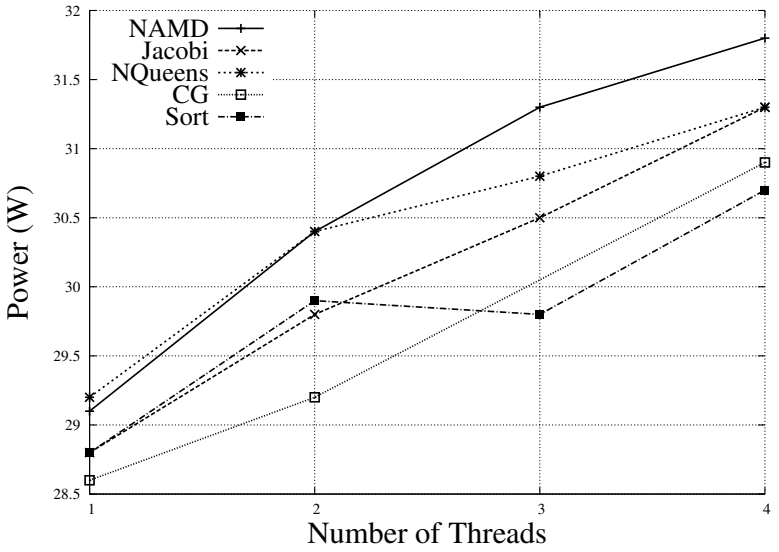


Figure 2.10: Power consumption of the applications on different numbers of threads in the Atom processor.

2.4 Comparison of Different Architectures

In this section, we use the data of previous sections to analyze and compare the architectures. For each of the three metrics (speedup, power and energy consumption), we run the applications on all the parallel threads of each platform, to use all the resources available. On all the platforms except the ION2, we use the same source codes, written either in Charm++ (Jacobi, NAMD, NQueens, and Sort), or in MPI (CG). For the ION2, we use OpenCL for Jacobi, NQueens and Sort, and CUDA for NAMD. Unfortunately, we could not get a reasonably tuned version of CG for the ION2.

When comparing the SCC to the other platforms, one should keep in mind that the SCC is a research chip, whereas the other platforms are highly-optimized production machines. With this in mind, one main purpose of the comparison is to find the weaknesses of the SCC and propose improvements for the future.

Figure 2.14 shows the speed of the five applications on the different platforms relative to Atom. As can be seen, the ION2 shows significantly better performance than the other platforms for Jacobi and NQueens, but not for NAMD or Sort. Jacobi and NQueens are simple highly-parallel applications with regular memory access and communication patterns. They match this "SIMD-like" ION2 hardware nicely. In contrast, Sort has irregular memory accesses and communication patterns, which make it unsuitable for the ION2. Finally, we could not obtain good speedups for NAMD on ION2 with our minimal porting and tuning effort, even though NAMD has been shown to scale well on GPGPUs elsewhere.

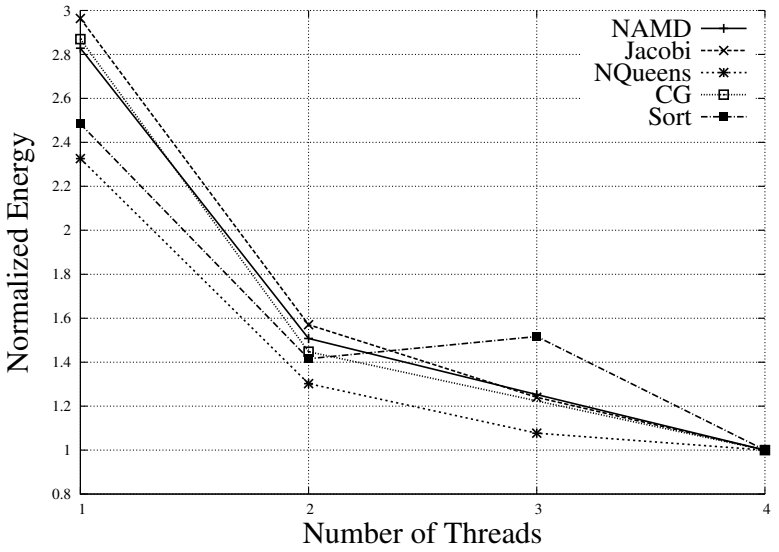


Figure 2.11: Energy consumption of the applications on different numbers of threads in the Atom. The data is normalized to the energy consumed with all the threads.

Porting applications to GPGPUs is one of the most important issues in these highly-parallel architectures. There are millions of lines of existing legacy parallel code, which cannot exploit GPGPUs easily (for example, scientific communities have a lot of parallel code mostly written in MPI). In addition, the effort for tuning and writing new code is high for GPGPUs. Generating highly-optimized codes on GPGPUs is not easy for an average programmer, and is not the subject of this work.

Overall, GPGPUs (and other architectures that are similar to SIMDs in general) are attractive for applications with simple control flow and high parallelism. However, they fail to provide good performance in other classes of applications.

In Figure 2.14, the Intel Core i7 is much faster than the other platforms for NAMD, CG and Sort. This shows that heavy-weight multi-cores are attractive solutions for dynamic applications with complex execution flow such as NAMD. The higher performance is due to high floating-point performance, support for short-length vector instructions such as SSEs, support for complex control flow (through aggressive branch prediction and speculation), and support for a high degree of instruction-level parallelism (attained by out-of-order execution).

In addition, these multi-cores are suitable for applications with irregular accesses and fine-grained communications, such as CG and Sort. These traditional platforms have highly-optimized cache hierarchies, share memory, and need less thread-level parallelism for high performance. Thus, irregular accesses are handled properly by the cache hierarchy, fine-grained communications are less costly because of shared memory, and there is less commu-

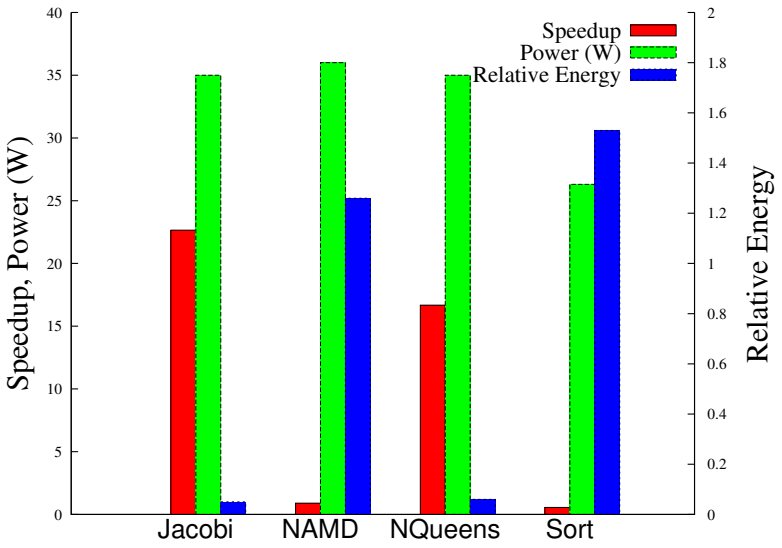


Figure 2.12: Speed, power, and energy of the applications running on the ION2 platform.

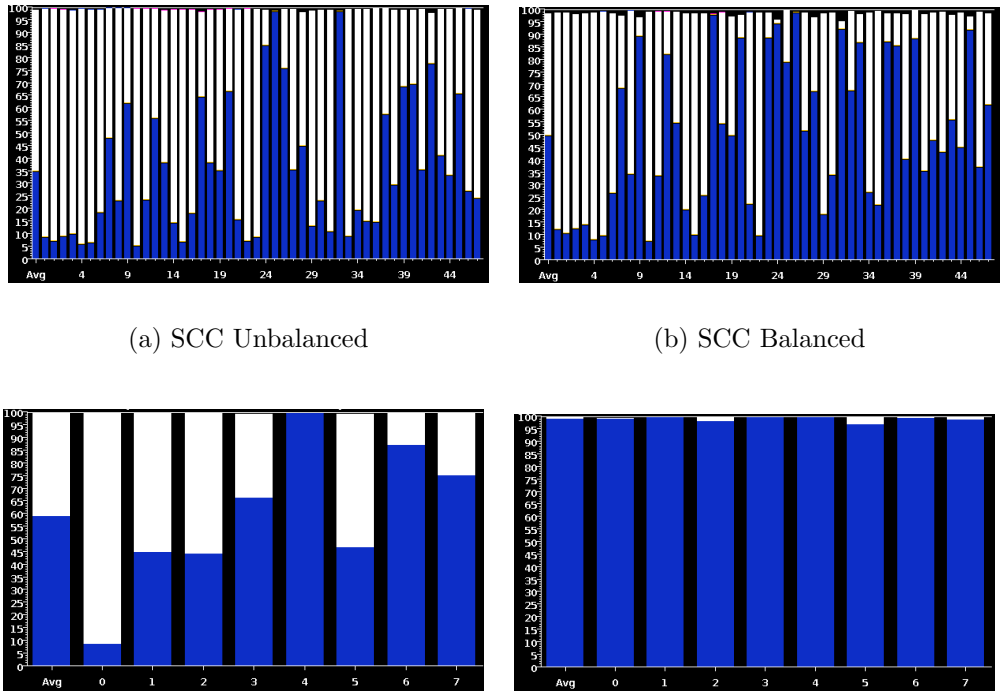
nication because of less parallelism.

Focusing on the SCC, the figure shows that, in general, the SCC speedups are not that good. The SCC is faster than the Core i7 for NQueens, but slower for the other applications. For such applications, it is comparable to the Atom. The fine-grain communication in CG is especially hard to support well in the SCC.

According to table 2.4, NQueens is an integer application, with few floating-point operations. On the other hand, Jacobi, which has a similar scaling behavior on the SCC, has many floating point-operations. Hence, low floating-point performance is a weakness of the SCC and enhancing it can improve performance substantially. Also, since §2.3 showed that all the applications except CG have scalable performance on the SCC, we believe that by improving sequential performance, the SCC can be much faster. The SCC needs more sophisticated cores, which is easily to attain because CMOS scaling will bring more transistors on chip in the near future. In addition, network performance also needs to be improved along with the cores, to keep the system balanced, which is also possible. Thus, an upgraded SCC many-core architecture can become a very attractive alternative for the future.

The Atom is slower than the other processors in most of the cases, which is expected by its low-power design.

Figure 2.15 shows the power consumption of the applications on the different platforms. In the figure, the Atom and the ION2 platforms consume low power. The Core i7 consumes the most power, because of its higher frequency and its many architectural features such as out-of-order execution. As can be seen in the figure, the power consumption of the SCC is



(a) SCC Unbalanced

(b) SCC Balanced

(c) Core i7 Unbalanced

(d) Core i7 Balanced

Figure 2.13: Utilization of the threads before and after balancing the load on the SCC and Core i7 platforms.

somewhere in between the low-power and the high-power platforms. The SCC decreases the power consumption compared to heavy-weight machines through more parallelism and lower frequency. It is an appealing platform because it moderates the high power consumption of conventional multi-cores, while still being general enough to avoid the usability issues of GPGPUs, and is generally faster than low-power designs like the Atom.

Figure 2.16 shows the energy consumption of the applications on the different platforms normalized to the energy consumed on the Atom platform. We use this nomalized-energy metric to be able to compare the different applications which, potentially, have a very different execution time. The figure shows that the ION2 platform is very energy-efficient for Jacobi and NQueens. For these regular applications, ION2 consumes low power and executes fast. However, ION2 is not so energy-efficient for the NAMD and Sort applications, because of their long execution time.

The Core i7 platform exhibits good energy efficiency across the board. The Atom platform is less energy-efficient. Although it uses low power, it has a relatively longer execution time and, therefore, the energy consumption is higher than in the Core i7 platform. Thus, low-power design does not necessarily lead to less energy consumption.

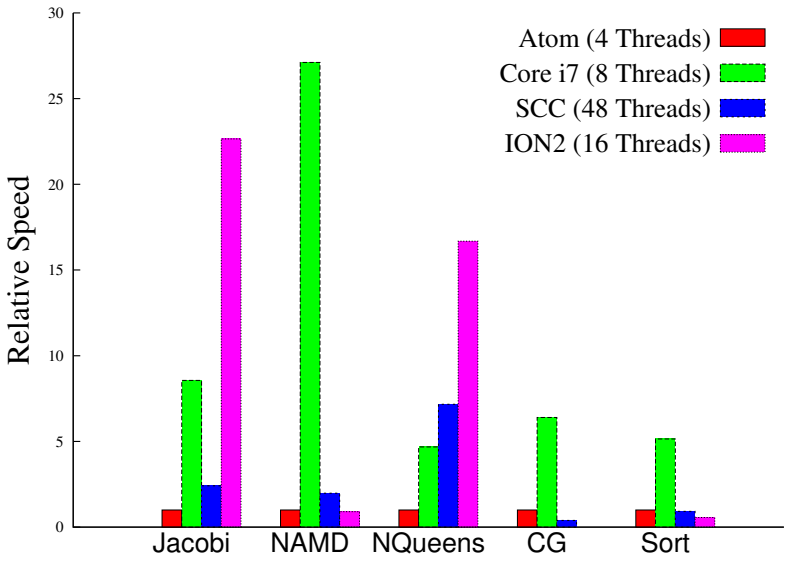


Figure 2.14: Speed of the applications on the different platforms relative to Atom.

The SCC platform is somewhat less energy-efficient than the Core i7 and Atom platforms for most applications, although it is still competitive. In the case of NQueens, the SCC is very energy-efficient. One exception is CG, where the SCC consumes substantially more energy. This is because CG has little computation and much communication. Overall, it is likely that an upgraded and refined SCC will have good energy efficiency for all of these applications.

2.5 Related Work

Marker et al. [22] port a dense matrix computations library to the SCC. They use the RCCE communications library and replace the collective communications. Using a lower-level communications library may have performance advantages, but it causes porting difficulties, especially when the collective is not implemented in the library. Our approach of porting the runtime system made running the applications possible without any change to the source code.

Power management is another topic of research for the SCC [23], because of its extensive DVFS support. Different parts of the SCC design, such as the network [24] or communication libraries [18] are described elsewhere [15, 17].

Different communication libraries, including MPI, have been studied and implemented for the SCC [25–27]. Porting Charm++ on top of them is a future study, which may result in

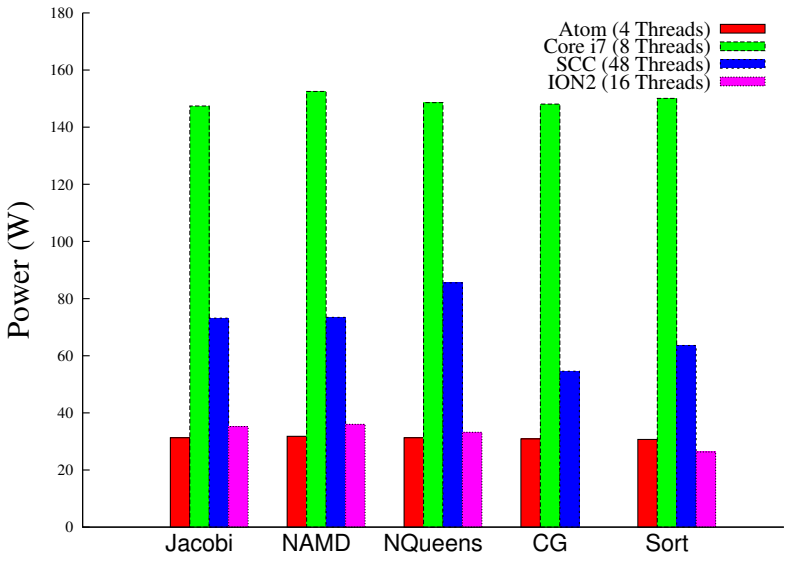


Figure 2.15: Power consumption of the applications on the different platforms.

performance improvements.

Esmaeilzadeh et al. [28] provide an extensive report and analysis of the chip power and performance of five different generations of Intel processors with a vast amount of diverse benchmarks [28]. Such work, however, does not consider many-cores or GPUs, which are promising architectures for the future of parallel computing.

2.6 Conclusion

Large increases in the number of transistors, accompanied by power and energy limitations, introduce new challenges for architectural design of new processors. There are several alternatives to consider, such as heavy-weight multi-cores, light-weight many-cores, low-power designs and SIMD-like (GPGPU) architectures. In choosing among them, several possibly conflicting goals must be kept in mind, such as speed, power, energy, programmability and portability. In this work, we evaluated platforms representing the above-mentioned design alternatives using five scalable Charm++ and MPI applications: Jacobi, NAMD, NQueens, CG and Sort.

The Intel SCC is a research chip using a many-core architecture. Many-cores like the SCC offer an opportunity to build future machines that consume low power and can run Charm++ and MPI code fast. They represent an intersting and balanced design point, as they consume lower power than heavy-weight multi-cores but are faster than low-power processors and do not have the generality or portability issues of GPGPU architectures. In

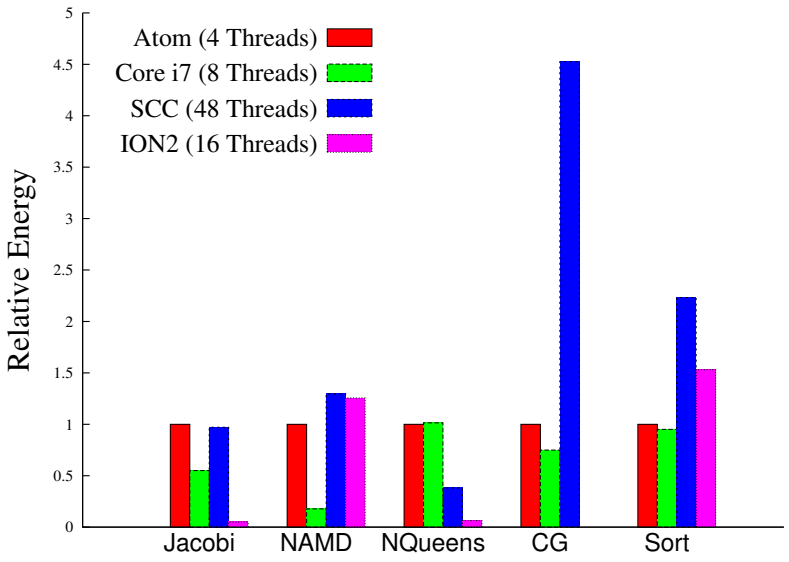


Figure 2.16: Energy consumption of the applications on the different platforms normalized to the energy on the Atom platform.

our analysis of the SCC, we suggested improvements in sequential performance, especially in floating-point operation speed, and suggested adding a global collectives network.

We showed that heavy-weight multicores are still an effective solution for dynamic and complicated applications, as well as for those with irregular accesses and communications. In addition, GPGPUs are exceptionally powerful for many applications in speed, power and energy. However, they lack the sophisticated architecture to execute complex and irregular applications efficiently. They also require a high programming effort to write new code, and are unable to run legacy codes. Finally, as seen from the Intel Atom experiments, we observe that low-power designs do not necessarily result in low energy consumption, since they may increase the execution time significantly. Therefore, there is no single best solution to fit all the applications and goals.

CHAPTER 3

Heterogeneous On-Chip Architectures: Case Study With Object Detection

In the previous chapter, we concluded that heterogeneous architectures, such as the ones with GPU accelerators, can be exceptionally powerful in performance, power, and energy efficiency. However, there are various challenges such as the difficulty of heterogeneous parallel programming. In this chapter, we study the challenges of heterogeneity and illustrate how to improve energy efficiency with less programming effort. Our insights can be used by the designers for more efficient heterogeneous runtime system designs. For this study, we consider a mobile application, which is considerably similar to HPC applications. Our results demonstrate that our techniques are general and applicable to many domains, and are not limited to the HPC domain. This is due to the prevalence of parallel architectures in various domains.

Many computing platforms used by consumers are portable devices such as notebooks, tablets, smart phones and more. Since these devices are usually battery powered, achieving high energy efficiency is a crucial challenge. On the other hand, because of their portability, mobile devices encounter many situations where they are expected to understand their environment in a natural way. For example, many photo applications need to automatically adjust the focal range based on the size of faces looking at a camera. In addition, gestures are frequently preferred to classical keyboard and mouse based input. Furthermore, search engines can allow a query to be formulated using visual inputs without requiring the user to provide the semantic translation of the visual content. Most natural interactions, such as the examples mentioned, require some usage of vision and video analytics algorithms. These tend to be floating-point intensive and computationally demanding, but also regular, which make them good candidates for parallelism.

Such data parallel algorithms adapt well to GPU type architectures, resulting in higher performance and energy efficiency [29]. However, general purpose programming of GPUs requires knowledge of new programming paradigms, such as CUDA and OpenCL, which decreases programmer productivity.

Traditionally, the GPU has been a peripheral component, used as a computational aid to the CPU (which is needed for latency-oriented functions such as the operating system). However, deploying stand-alone GPUs may not be desirable (or even practical) for portable platforms for different reasons. First, using an extra chip increases the system design and implementation cost significantly. Second, the extra chip, along with its associated overheads such as power supplies, increases the power and energy consumption. Third, the off-chip connection between the CPU and the GPU may have high performance and energy overheads.

A reasonable alternative for deploying a GPU is to put it on the same chip as the CPU, and create a heterogeneous on-chip architecture. Advancements in system-on-chip design and increases in the number of available on-chip transistors has made hybrid architectures practical. Emerging examples such as Intel Ivy Bridge [30], AMD Fusion [31] and NVIDIA Tegra 250 [32] have implemented this idea.

For this study, we have chosen an application, object detection using ViVid [33], as a representative of vision applications. The domain of our study is on-chip hybrid architectures, which are most predominantly found in mobile platforms. We believe that object detection is a representative mobile application because it is fairly computationally demanding and it processes streamed visual input from a camera. Similar to our study, most vision applications that would be utilized in mobile devices (e.g. recognition, tracking, stabilization) consist of a pipeline of small number of kernels, where kernels are the core compute intensive components of an application. Of course, there is a large variety of kernels across the spectrum of vision applications. However, from a computational perspective, the pipeline in this chapter provides a good mixture of kernels efficient on GPU, CPU or both. In addition, object detection is an important application for future portable devices, which has not yet been realized beyond basic face detection. Notice that our focus on one application allows us to go deeper into the details of individual kernels. We describe and evaluate the steps one might take to improve performance and energy efficiency: (1) Code optimization, (2) Mapping strategies, (3) Dynamic Voltage and Frequency Scaling (DVFS) [34] and (4) Algorithmic tradeoff of accuracy. We report the lessons learned, which would give insight to application developers and system designers.

In this chapter, we evaluate and analyze different programming paradigms and strategies for energy efficiency. We implement and execute (on the Ivy Bridge architecture) four

different code versions of ViVid using 1) OpenCL, 2) OpenMP + auto-vectorization, 3) OpenMP + vector intrinsics, and 4) the OpenCV vision library. The OpenCL version runs on both the CPU and the GPU, while the other versions only run on the CPU. Our experimental results show that OpenCL does not deliver the performance that can be attained when using lower level interfaces (e.g. vector intrinsics on CPU), but provides a reasonable performance (Section 3.3). The OpenCL code processes 40 frames per second (fps) for accurate object detection (Section 3.4), so it can be used for applications that require real-time object detection (33fps). Notice that the performance of our OpenCL implementation is superior or similar to recent works using much more capable discrete GPUs [35, 36].

We also show that mapping each kernel to the device (CPU or GPU) where it executes more efficiently and overlapping the computation of the kernels is the best approach. Our results show that with these heterogeneous platforms it is possible to find mappings that, while executing relatively faster, are less energy efficient (this is discussed in Section 3.4). In addition, it is possible to gain better energy efficiency by sacrificing a small amount of accuracy algorithmically. For our application, we can reduce 20% of the energy consumed at the cost of an increase of only 1% miss-rate on image detection (Section 3.4.3).

Note that manufacturers do not know how to design hardware and software of future portable devices to support new interfaces (e.g. for human interaction). For instance, specialized hardware accelerators and optimized vision libraries are considered. We show that using a unified programming paradigm (e.g. OpenCL), vision applications can deliver the required performance (for a typical number of frames per seconds) and energy efficiency on heterogeneous on-chip architectures. To the best of our knowledge, the literature only considers large discrete GPUs with very different trade-offs in performance and energy efficiency for these applications.

The rest of this chapter is organized as follows. Section 3.1 describes our application and experimental setup briefly. Section 3.2 evaluates and analyzes different optimizations for our kernels using OpenCL for the CPU and the GPU. Next, Section 3.3 compares the performance and programming effort of the OpenCL paradigm to others for the CPU. After that, Section 3.4 evaluates the performance and energy consumption of different kernels on either the CPU or GPU. It also optimizes the full application's performance and energy consumption using different mapping methods. Finally, Section 3.5 reviews some related work and Section 3.6 concludes the chapter.

3.1 Environmental Setup

3.1.1 ViVid

We focus our study on an object (e.g., face) detection algorithm [33] for finding objects with a specific shape or appearance in unconstrained visual input. This object detector is analogous to most practical approaches [37, 38] to this problem, which follow a common work-flow called "sliding window object detection". This process involves describing the visual information inside small rectangular regions of the image or video frame hypothesized to contain the object, and applying a decision function that yields a binary output indicating the presence or absence of the object in each of such rectangles. Sliding window detection is the most established approach for the unconstrained object detection problem. Other popular methods include generalized voting frameworks [39] or contour matching [40]. In all cases, object detection is a very computationally demanding application because image information needs to be evaluated densely over all the potential locations which may contain the object. ViVid's sliding window approach breaks up the problem into two distinct parts: 1) describing the image information, and 2) classifying it. The image information is described by correlating the gray-scale image with numerous 3×3 patterns and summarizing these correlations in terms of spatially local histograms. The classification is achieved by processing these histograms through linear support vector machines [41].

Other than object detection, there are numerous applications of computer vision on mobile devices including video stabilization, panorama stitching, gesture recognition etc. However, the data description followed by a data association work-flow is a common pattern. Typically, the data description part touches every pixel at least once and builds a summarization of structures of interest (e.g. colors, gradients, textures). The data association part measures the distance between the data summaries against stored exemplars. In classification applications, these can be templates for objects, and in segmentation applications these are usually cluster centers. The computational stages in a mobile computer vision application may be computationally balanced or particular stages may give rise to performance bottlenecks. In our selected object detection algorithm, the data description and data association steps are well balanced in terms of their computational load. Therefore, we believe it comprises a good case study with challenges in both stages.

To build our object detector pipeline, we use the ViVid library[1]. ViVid includes several atomic functions common to many vision algorithms. We have used ViVid successfully in event detection applications [42, 43].

[1] http://www.github.com/mertdikmen/vivid

For the purposes of this work, we extended ViVid by adding OpenCL equivalents of several kernels. We use the C++ interface to orchestrate the calls to these OpenCL functions or kernels.

3.1.2 Blockwise Distance

This kernel needs to find the maximum response (normalized cross correlation) of 100 filters on a small square image patch (in this application, 3×3) centered at every pixel of the image, while remembering which filter delivered this maximum at every pixel. Algorithm 1 outlines the overall algorithm.

```
for each 3 by 3 image patch centered at a pixel do
    for each filter j of 100 filters do
        response = 0;
        for each coefficient i of the 9 coefficients of filter[j] do
            response += filter[j][i]*pixel[i];
        end
        if response > max_response then
            max_response = response;
            max_index = j;
        end
    end
end
```

Algorithm 1: Filter kernel.

3.1.3 Cell Histogram Kernel

Cell histogram kernel is the second stage of data description, where the low level information collected by the filter kernel is summarized for small, non overlapping square blocks of the image. A 100 bin histogram is populated for each of these blocks by accumulating the "max_response" values in their respective bins (given by "max_index") from every pixel inside the block. Note that this operation is different from well known image histogramming problem, for which many parallel implementations exist. Our approach differs in two important aspects: (1) the histogram bins represent a weighted sum (not a simple count) and (2) we build many local histograms not a single global one.

3.1.4 Pairwise Distance

This kernel is the data association step in our application. It finds the Euclidean distance between two sets of vectors, where one vector corresponds to the histogram previously generated and the other vector represents the template. This kernel measures how close each descriptor is to the template of the object of interest. If the distance is small enough, it shall output a detection response.

The kernel is structurally similar to the matrix multiply operation, which finds the dot product between every row of one matrix and every column of another one. However, in pairwise distance, we compute the square of the two values' differences, instead of just multiplying them.

3.1.5 Ivy Bridge Architecture

For the experiments reported in this chapter, we use the two different platforms shown in Table 3.1, both based on the Intel Ivy Bridge architecture. The first one is a 3.3 GHz quad-core used for Desktops and the second one is 1.7 GHz dual-core used for Ultrabooks. Both platforms have an integrated GPU that can be programmed using OpenCL[2]. GPUs exploit Single Instruction Multiple Thread (SIMT) type of parallelism by having an array of Compute Units (CUs). Each CU is assigned a work-group, where work-items in each group run in lock-step, executing the same instruction on different data. GPUs are designed to efficiently exploit data parallelism. Branchy codes may run poorly on GPUs, as all the different paths in a control flow need to be serialized. Note that the Ivy Bridge's GPU is simpler than Nvidia [44] or AMD/ATI [45] GPUs. It has a small number of compute units and simpler memory hierarchy, for instance.

The Ivy Bridge CPU contains multiple cores, where each core supports Advanced Vector Extensions (AVX) that apply the same instruction on multiple data simultaneously. AVX supports 256-bit wide vector units that allow vector operations to operate on 8 floating-point numbers simultaneously. Unless otherwise stated, the experiments reported in the chapter use the Ultrabook platform. For comparison purposes, we have also run experiments on the Desktop platform and an Nvidia Fermi GPU.

For the evaluation, we use Intel SDK for OpenCL [46] to run OpenCL codes. We also wrote OpenMP code with and without vector intrinsics that we compiled using the intel ICC compiler and /O3 compiler flags. Table 3.2 summarizes the software environment we use for the experiments. OpenCL is a unified programming paradigm, used to write programs

[2]http://www.khronos.org/opencl/

Table 3.1: Intel Ivy Bridge (Core i5 3350 & 3317U) processor specifications.

Platform	Desktop	Ultrabook
Processor Number	i5-3550	i5-3517U
# of Cores	4	2
Base Clock Speed	3.3 GHz	1.7 GHz
Max Turbo Frequency	3.7 GHz	2.6 GHz
Base CPU peak	105.6 GFLOPs	27.2 GFLOPs
Max CPU peak	118.4 GFLOPs	41.6 GFLOPs
Cache Size	6 MB	3 MB
Lithography	22 nm	22 nm
Max TDP	77 W	17 W
Intel HD Graphics	2500	4000
GPU Execution Units	6	16
GPU Base Frequency	650 MHz	350 MHz
GPU Max Dynamic Frequency	1.15 GHz	1.05 GHz
Base GPU peak	31.2 GFLOPs	44.8 GFLOPs
Max GPU peak	55.2 GFLOPs	134.4 GFLOPs

for heterogeneous platforms. OpenCL programs can use an address space qualifier (such as __global for global variables or __local for local variables) when declaring a variable to specify the memory region where the object should be allocated. The OpenCL implementations for the GPU in the Ivy Bridge accesses memory through the GPU-specific L3 cache and the CPU and GPU Shared Last Level Cache (LLC). Accesses to global variables go through the GPU L3 cache and the LLC. Accesses to local memory (also referred as shared local memory because this local memory is shared by all work-items in a work-group) is allocated directly from the GPU L3 cache. Thus, GPU L3 cache can be used as a scratch-pad or as a cache. The size of this memory is 64KB (obtained using the standard "clGetDeviceInfo()" OpenCL call) for both platforms, the Desktop and the Ultrabook. The CPU does not have hardware support for local memory, so in principle codes running in the CPU do not benefit from using local memory. Additional details can be found in [47].

3.1.6 Evaluation Methodology

For the experimental results, we measure the time of thousands of iterations of the application and report the average. This is realistic for many vision applications, which are expected to perform analysis (e.g. detection) over a continuous input of frames, fed from the device

Table 3.2: Software environment used for experiments.

Operating System	Windows 8 Build 9200
GPU driver	Intel 9.17.10.2867
OpenCL SDK	Intel 3.0.0.64050
Compiler	Intel ICC 13.0.1.119

camera. This setup is especially important for the Ivy Bridge GPUs, since the running times have high variance in the first few iterations, but stabilize after some "warm up" iterations. For all the experiments reported here, our input image size is 600 by 416 pixels.

For power and energy measurements, we use hardware energy counters available in the Ivy Bridge architecture [48]. They measure three domains: "package", "core" and "uncore". *Package* means the consumption of the whole chip, including CPU, GPU, memory controllers, etc. *Core* is CPU domain and *Uncore* is the GPU domain. For power measurement of the whole system, we plug a power meter to the machine's power input.

The new Intel Turbo Boost Technology 2.0 [49] makes the measurements complicated on this architecture. In a nutshell, it accumulates "energy budget" during idle periods and uses it during burst activities. Thus, the processor can possibly go over the Thermal Design Power (TDP) for a while. It takes it a few seconds to reach that limit and several seconds to go back to the TDP limit. This can change the performance and power of the processor significantly. One might turn this feature off for accurate measurements. However, it is an advanced strength of the architecture that can enhance the user experience significantly (e.g. for interactive use), so it should not be ignored. For our measurements, we run each program for around 10 seconds (which seems to be a valid common use case) and average the iteration times and power consumption.

We used the machine peak performance numbers reported in the Intel documentation[3]. However, those values are computed using the maximum frequency value and AVX vector units, but, as mentioned, the processor cannot be at the maximum frequency for a long time. Thus, in many cases, peak performance numbers are upper bounds of the actual peak performance.

[3]http://download.intel.com/support/processors/corei7/sb/core_i7-3700_d.pdf

3.2 Optimization of Kernels in OpenCL

In this section, we describe the optimizations we applied to the OpenCL kernels described in Section 3.1. Then, in Section 3.2.4, we analyze the performance impact of each optimization. The OpenCL codes run in both the CPU and the GPU, but it is possible that an optimization that works well for the GPU would hurt the performance when running on the CPU or vice versa. From now on, we will refer to the Blockwise Distance Kernel as *filter*, the Cell Histogram Kernel as *histogram*, and the Pairwise Distance kernel as *classifier*.

3.2.1 Filter Kernel

Here, we describe the optimizations that we applied to the filtering algorithm shown in Figure 1.

Parallelism

We exploit parallelism by dividing the image across multiple work-groups with several work-items. Then, each work-item runs the 100 filters on its image block. We use 16 by 16 work-group size following Intel OpenCL SDK's recommendation (considering also our working set memory size). In addition, we use the Kernel Builder (from the Intel OpenCL SDK) tool's work-group size auto-tuning capabilities to make sure this is the best size.

Loop Unrolling

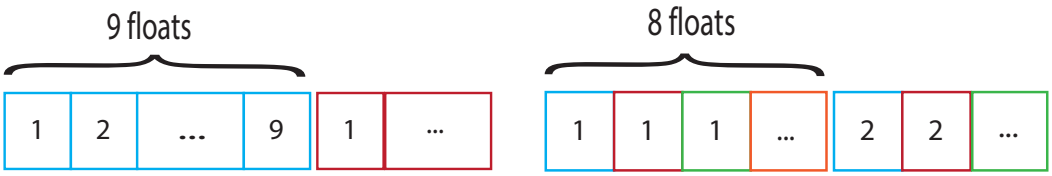
We completely unroll the inner loop, which has 9 iterations.

Vectorization

This transformation tries to exploit the CPU's AVX vector units. Without vectorization, this kernel calculates the response of every filter on a three-by-three image patch, keeping track of the maximum one. This requires nine multiply-add operations, followed by an update guarded by an *if* statement. In this form, the inner loop cannot be fully vectorized. Since *AVX* supports 8 operations at a time, we can vectorize eight of the multiplies and partially vectorize the sum reduction, but still need to run one sequentially. Thus, to enable efficient vectorization, instead of working on one filter at a time, one can consider eight of them at the same time. Note that the number of filters (100) is not a multiple of eight so we need to

handle the last four filters separately. Each pixel value needs to be replicated (broadcast) in a vector to participate in the vector operations.

This transformation needs a reorganization of the filter coefficients' data structure in the memory. Originally, a filter's nine coefficients are located in consecutive memory locations (Figure 3.1(a)). However, we need the first coefficients of eight filters to be together to be able to load them in a SIMD vector (Figure 3.1(b)). Figure 3.1 illustrates these layouts using different colors for different filters, and numbers for different elements of a filter. Thus, effectively, we are transposing each 8×9 sub-matrix of eight filter coefficients to an 9×8 one. This transformation is generally useful for vectorizing various filters of different sizes since the number of coefficients most probably does not match the SIMD size. Note that this transformation can be thought of as a customized instance of the Array of Structures (AoS) to Structure of Arrays (SoA) transformation.



(a) original coefficient layout (b) vectorized coefficient layout

Figure 3.1: Change of coefficient data layout for vectorization.

Local Memory

This optimization is specific to the GPU. The filter kernel operates on an image and the 100 filter coefficients. The filter coefficients occupy 3.5KB that we copy (using all the work-items in parallel) to the local memory. Each work-group also copies the image block it needs. This optimization may hurt performance when the code runs on the CPU due to the copying overheads. This is evaluated in Section 3.2.4.

3.2.2 Cell Histogram Kernel

The parallelism is achieved through a scatter operation. Every work-item in a work-group accumulates a subset of the values inside the image block to their respective histogram bins. Note that this is a potential race if two or more work-items in the same work-group

try to increment the same histogram bin. This race can be avoided if the language and the hardware allow for "atomic_add" directives for floating point numbers. However, these atomic operations serialize memory accesses and can hurt the performance significantly.

We allow this race in our OpenCL kernel because our Monte Carlo simulations have shown that the probability of such a race is low given the distribution of filter indexes in natural image patches. Therefore we do not expect the race conditions to change the shape of the histograms drastically, and we have validated this through experiments. Unlike scientific applications, media programs do not need full accuracy in many cases, and we should exploit this for better performance and energy efficiency.

3.2.3 Classifier Kernel

Parallelization

Parallelizing this code is similar to a tiled matrix multiply, where a work-group is responsible for a tile of the output matrix (as with the filter, we use 16x16 tiles).

Loop Unrolling

We manually unroll the innermost loop, which has 16 iterations.

Vectorization

Vectorizing this code is easy as operations are done in an element by element fashion, with elements in consecutive memory locations. After accumulating differences in a vector, a sum reduction is required (which we implement as a dot product with an identity vector).

Local Memory

All the work-items load the two blocks of elements they want to work on in parallel in the local memory.

3.2.4 Performance Evaluation

In this section, we evaluate the performance impact of each of the optimizations. Figure 3.2(a) shows the execution time for *filter* when running on the CPU or the GPU. The bars

show the cumulative impact of the different transformations. Thus, Unroll+Vec+LocalMem corresponds to the execution time after all the optimizations have been applied. As Figure 3.2(a) shows, after applying all the above optimizations, this kernel runs more than 10 times faster on the GPU than the original non-optimized code. It now takes only 8.46ms. It also made it 6.4 times faster on the CPU (takes 25.5ms for the same image). Loop unrolling speeds up this kernel for both the CPU and the GPU. Vectorization speeds up *filter* for the CPU significantly. Also, even though the GPU does not have CPU-like vector units, execution times decreases by about 16% (discussed later). We also note that the use of the local memory for the filter coefficients does not have a significant overhead on the CPU. Thus, the same kernel code can be used for both architectures.

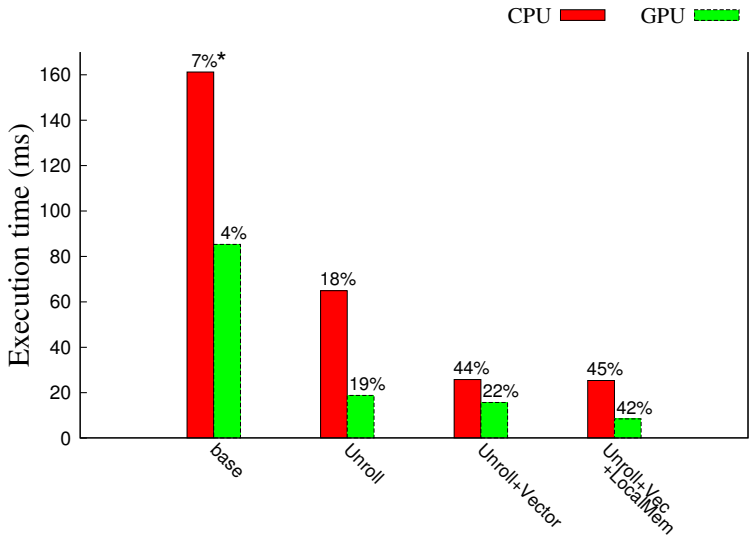
Figure 3.2(b) shows the results for the classifier. As the figure shows, both unroll and vectorization improve the performance significantly. However, the use of local memory degrades performance for both devices. Hence, we did not use the local memory for *classifier* and, again, the same code is used for both the CPU and the GPU.

To assess the utilization of the CPU and GPU, we measured the MFLOPs for both *filter* and *classifier*. Numbers on top of the bars in Figures 3.2(a) and 3.2(b) show the performance of each code as a percentage of the peak performance of the machine. Our numbers show that *filter* runs at 45% and 42% of the peak performance on the CPU and GPU, respectively. *Classifier* runs at 49% and 9% on the CPU and GPU, respectively. Thus, filter utilizes both the CPU and GPU very well, while *classifier* only has that level of utilization on the CPU. The inefficiency of GPUs for certain workloads has been discussed in related work [50]. However, 9% utilization might be considered high for certain workloads on the GPU.

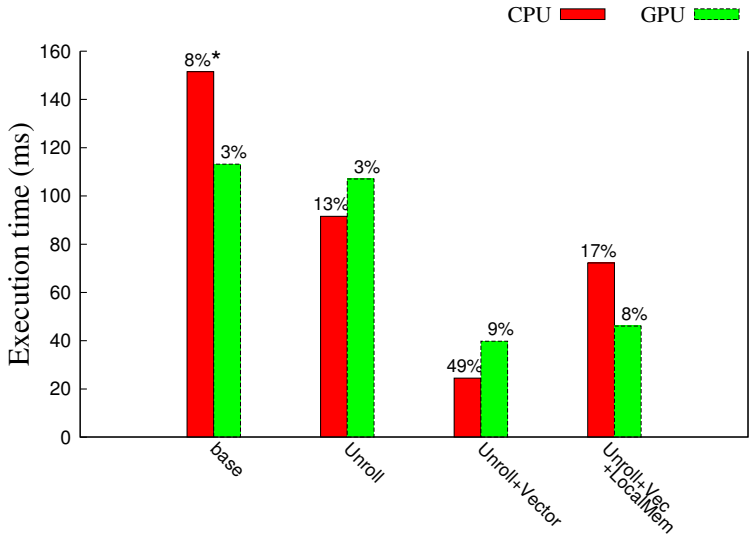
Note that a more optimized code usually results in faster and more energy-efficient execution. In fact, in our experiments we observed that the programs were consuming the same or similar power before and after the optimization. Thus, decreasing execution time almost directly results in lower energy consumption in this case.

Table 3.3 summarizes our results about which optimizations were effective on each device. Note that the OpenCL filter kernel that we use in the following sections uses the local memory optimization, since it does not hurt performance, and this allows us to have a single code version for both devices.

Next, we discuss the performance results, specifically the impact of using local memory, branching effects, and vectorization in more detail.



(a) Filter kernel



(b) Classifier kernel

Figure 3.2: Execution time of kernels with different optimizations (on Ultrabook); *- percentage of peak performance.

Local Memory Usage

Our results in Figures 3.2(a) and 3.2(b) show that the use of local memory is important for the filter kernel but undesirable for the classifier.

Using the local memory is essential for the filter kernel on the GPU. The reason is that a

Table 3.3: Effective optimizations for filter and classifier kernels on Ultrabook.

Kernel	GPU			CPU			Same code?
	Unroll	SIMD	Local-Mem	Unroll	SIMD	Local-Mem	
filter	yes	yes	yes	yes	yes	no	yes
classifier	yes	yes	no	yes	yes	no	yes

small set of constant data (the filter coefficients) are needed for the whole execution (all the iterations of the outer loop). Relying on the GPU L3 cache is not effective because the data from the image that is being accessed at the same time might replace the filter coefficient in the cache.

On the other hand, using the local memory is detrimental for the classifier kernel on the GPU of our Ivy Bridge Ultrabook. However, using the local memory for the classifier improves the performance on the smaller GPU (HD Graphics 2500 device) of the Desktop platform by 35%, even though the architecture is essentially the same (the Desktop GPU has only 6 CUs, while the Ultrabook GPU has 16 CUs).

To understand the differences in performance (in the absence of performance counters), we used the memory streaming micro-benchmark of uCLbench package [51] that measures the effective bandwidth to memory. This benchmark allocates arrays in memory (either local or global), that are accessed by all the work-items repeatedly. Our experimental results show that the effective bandwidth of local memory is less for the Ultrabook GPU than for the Desktop GPU (7.8 GB/s for the Ultrabook vs. 10.3 GB/s for the Desktop) when local arrays are accessed. On the other hand, the effective bandwidth of global memory is about the same for both machines (7 GB/s for the Ultrabook vs. 7.4 GB/s for the Desktop) when global arrays are accessed. Notice that the working set of the classifier is just the data that we are placing on the local memory and fits in the 64KB of the GPU L3 cache. Thus, since the Desktop has a higher effective bandwidth when accessing the data in the local memory, the local memory optimization reduces execution time. However, in the Ultrabook the bandwidth is similar and the use of local memory introduces some copying overheads.

Using local memory for the code running on the CPU introduces some extra copying overhead. While this overhead is not visible for *filter* because of the small size of the filter coefficients data structure, it adds a significant overhead to the classifier kernel, due to the larger size of the data structure allocated in local memory.

Loop Unrolling and Branch Overhead

Unrolling results in a significant performance improvement in both kernels, *classifier* and *filter*, for both the CPU and GPU. In the CPU unrolling decreases loop overhead and increases Instruction Level Parallelism. In the GPU, unrolling reduces the number of branches.

Branches on the GPU can have a significant impact on performance, specially in the case of divergent branches where work-items (threads) of a CU take different paths, and each branch path has to be serialized. On the other side, non-divergent branches, where all the work-items follow the same path, are usually fast.

To assess the impact of non-divergent branches on the Ivy Bridge integrated GPU, we modified the *filter* kernel, and replaced the "if" condition that finds the maximum filter response with additions that sum the filter responses (notice that this branch, although data dependent, is mostly non-divergent, as work-items execute on neighboring pixels that tend to be similar and hence the maximum response filter is mostly the same for all the work-items). This change made this code run 13% faster on the integrated GPU. We also ran both codes (with and without the "if" statements) on the Fermi Nvidia GPU and found that the code without the branches had only 3% improvement. In addition, we used the "branch overhead" benchmark of uCLbench package [51] to assess the difference in performance between divergent and non-divergent branches. In this benchmark, different cases of branch divergence are compared. For example, a branch might be taken by all the work-items, a subset of them or only one. The experimental results show that the Ivy Bridge's integrated GPU is performing much better for non-divergent branches, as benchmarks can be up to 10 times slower on the Ivy Bridge's integrated GPU when branches are divergent.

Overall, our experiments show that non-divergent branches have a higher effect on the Ivy Bridge GPU than on a Fermi GPU. Thus, loop unrolling (that removes non-divergent branches) is an important optimization for this platform. Other non-divergent branches, such as the "if" associated with the max operator cannot be removed with loop unrolling, and would benefit from a better hardware support for non-divergent branches.

Vectorization

Vectorization speeds up both the codes for the CPU, as it makes it easier for the compiler to generate code using the AVX vector extensions in the Ivy Bridge. When running on the GPU, classifier is about 2.8 times faster with vectorization, despite the fact that vector units need to be emulated on the GPU, which might have some overheads. One reason is that the vector code has more unrolling on the GPU implicitly. Thus, to assess the effect of further

unrolling, we unrolled the non-vectorized code's outer loop as much as it is beneficial (and "jam" it into the inner loop, which is already unrolled). This code runs faster, but still 1.8 times slower than the SIMD version. The other reason for the difference in performance is found by looking at the code generated by the compiler for both versions (with and without SIMD). For the code with SIMD, the compiler generates different memory load instructions with better alignment, which is important for performance.

As mentioned, filter kernel runs only slightly (13%) faster on the GPU when vectorization is applied.

3.3 Comparison with Other Programming Paradigms

In this section, we assess if OpenCL is a suitable paradigm for the CPU, since it is desirable to have a single programming paradigm for both types of devices.

For that, we compare the programming effort and execution times of the OpenCL *filter* code versus implementations of the same code written with other programming models for the CPU. *Filter* code is chosen for the comparison because it is a compute intensive kernel, based on a convolution operation used by many computer vision applications.

We run the experiments of this section on the Desktop's CPU, since it is more powerful and will reflect the effects better. In addition, the Ultrabook's CPU does not support SSE vector instructions. Note that for all the experiments we use 4 byte "float" precision numbers (which are enough for the filter kernel).

3.3.1 OpenMP with Compiler Vectorization

Since OpenMP is well suited to exploit data parallel computation in multicores, we compare the OpenCL code with an OpenMP implementation. Although one could expect perfect speedups, our results show an overhead of 8% with respect to perfect scaling. This is due to the overhead of spawning and joining threads for every loop invocation on a different image.

To exploit the machine's potential, we need to exploit the CPU's vector units. The simplest way is to have the compiler do this task. The Intel compiler that we use (Section 3.1.5) can vectorize this code, but needs the "/fp:fast" flag, to enable optimizations that can cause minor precision loss in vectorization of reductions. In addition, by looking at the assembly code, we realized that it did not generate aligned loads, which was fixed by using Intel compiler intrinsic function (*__assume_aligned()*).

Furthermore, with the hope that the compiler would generate better code, we generate another code version where we applied, at the source level, the transformation we applied to vectorize the OpenCL filter kernel (Section 3.2.1).

3.3.2 OpenMP with Manual Vectorization

We vectorized the code manually using vector intrinsics that map directly to assembly instructions. A disadvantage of this approach is that the code is not portable as it is tied to a specific machine's instruction set and a compiler. Furthermore, it is close to the assembly level and hence, the programming effort including code readability and debugging will suffer. Nonetheless, if the performance difference can be very high, one might prefer paying the cost. We wrote three versions: using AVX and SSE, using only SSE and using only AVX.

AVX+SSE

The Ivy Bridge architecture supports the AVX and SSE instruction sets. AVX instructions can work on eight floating point elements, while SSE ones can only handle four elements. We use SSE, since AVX does not have an instruction equivalent to SSE's "_mm_comigt_ss" (that compares two values and returns a 1 or a 0 depending on which one is larger), which simplifies the coding. Thus, we use AVX for multiply and add operations and SSE for conditional comparisons. Note that mixing AVX and SSE instructions can have significant translation penalties on Ivy Bridge [52]. However, we use "/Qxavx" flag to ask the compiler to generate AVX counterparts whenever possible. In addition, we use Intel vTune Amplifier to make sure these penalties are avoided. Since this kernel needs to find which filter resulted in the maximum response value, we compare the max_response against each response value. A sample comparison is shown below, where we permute the result vector and compare the lowest index element using the "_mm_comigt_ss" intrinsic.

```
1 __m128 p_tmp = _mm_extract_ps(response1, 0x1);
2 if(_mm_comigt_ss(p_tmp, max_response)) {
3       max_response = ptmp;
4       best_filter = filter_ind+1;
5 }
```

Note that we provide code snippets to be able to compare the complexity of different methods. We refer the interested reader to Intel's documentations to fully understand the details.

SSE

We implemented an SSE version to evaluate AVX versus SSE and measure the effect on performance of SIMD width.

AVX

We also implemented a version that only uses AVX instructions. The implementation compares all the responses in parallel, gathers the sign bits in an integer mask and examines each bit separately. If the maximum response needs to be updated, we use a permutation instruction to broadcast the new maximum to the register, repeat the comparison and update the integer mask. There is a small complication because of "_mm256_permute_ps" instruction's semantics. Since it can only choose from each four element half of the register separately, we need to consider each half of the responses separately and copy it to the other one. Thus, the initialization code for comparing four elements of responses is shown below:

```
// low 128 half
// copy low to high
__m256 response1 = _mm256_insertf128_ps(
 response, _mm256_extractf128_ps(response, 0), 1);
__m256 cpm = _mm256_cmp_ps(
   response, max_response, _CMP_GT_OS);
int r = _mm256_movemask_ps(cpm);
```

After that, we will have four tests of the mask with possible updates similar to the one below:

```
if(r&(1<<1)) {
      best_filter = filter_ind+6;
      int control = 1|(1<<2)|(1<<4)|(1<<6);
      max_response = _mm256_permute_ps( response1, control);
      r=_mm256_movemask_ps( _mm256_cmp_ps(
        max_response, max_response, _CMP_GT_OS));
}
```

3.3.3 OpenCV Library Calls

OpenCV [53] is an open source library consisting of many low level image processing algorithms, as well as many high level algorithms frequently used in computer vision. It is by far the most utilized common code base for vision research and applications. We constructed

the object detection algorithm using standard library data structures and function calls to OpenCV in order to compare what is achievable in terms of performance using the standard C++ interface. We link against the standard distribution of the OpenCV binaries, which is not multithreaded[4].

The *filter* kernel can be constructed simply by 100 calls to the OpenCV 2 dimensional filtering function. Between each OpenCV filtering function call, we do a pass over the output array to determine if the current filter response value is higher than the maximum value observed and replace the assignments in the output array accordingly. The *classifier* kernel can be simply constructed by taking the norm of the differences for every pair of rows in the two input matrices. Row isolation, vector differencing and vector norms are all standard library calls in OpenCV. The histogram kernel is not achievable through standard library calls, but the ViVid call can be used on standard OpenCV data types with minimal changes.

Our experimental results show that the OpenCV implementation of the filter kernel runs 15 times slower than the OpenCL version of ViVid running on the CPU of the Ultrabook. Notice that our OpenCL code runs in parallel (two cores of the Ultrabook), while the OpenCV code runs sequential. Also, the OpenCV code that we use has hard-coded SSE2 intrinsics (we verified by looking at the library code), while our OpenCL code uses the AVX vector instructions (when running on the CPU). However, these two points still do not justify the big difference in performance. This substantial difference in performance appears because the OpenCV code does not take advantage of locality, as we need to re-load the image 100 times. In our OpenCL code, each image pixel is loaded only once, as the 100 filters are applied to each pixel before moving to the next one. Notice that experimental results for *classifier* show that the OpenCV code is also significantly slower than our OpenCL code. The reason is that while the OpenCV library has an efficient matrix multiplication call, what we need is a customized operator (the square of the two value's differences or Euclidean distance), which needs to be realized in an inefficient manner (as mentioned above).

While the code using the library calls is very concise and straightforward, the non-perfect adoption of OpenCV library primitives for our algorithm results in performance degradation. This is because each library call has some overhead, and no optimization is possible across library primitives. Vision applications can benefit significantly from specific low level optimizations based on the expected input and output structures, as well as computational patterns of individual kernels. Thus, current vision libraries are unable to solve the entire parallel programming problem for vision applications, as the resulting code is not fast enough

[4]Some OpenCV functions can be made multithreaded by linking against Intel Thread Building Blocks [54]

for production use. However, ease of use has made these libraries, such as OpenCV, very good candidates for application prototyping and development.

3.3.4 Performance and Effort Comparison

Performance Comparison

Figure 3.3 shows the execution time and the percentage of peak performance obtained by different schemes running on the Desktop platform (results in the Ultrabook are similar as these experiments are mainly concerned with vectorization). In the schemes evaluated, *orig-novec* corresponds to the baseline OpenMP code where compiler vectorization is disabled; *orig-auto* corresponds to the OpenMP code auto-vectorized by the compiler; *trans-auto* is the OpenMP code transformed for vectorization at the source level (like in the OpenCL code in Section 3.2.1) and automatically vectorized by the compiler; the three code versions using intrinsics are labeled *SSE*, *AVX+SSE* and *AVX*; *OpenCL* corresponds to the OpenCL code optimized as discussed in Section 3.2.1.

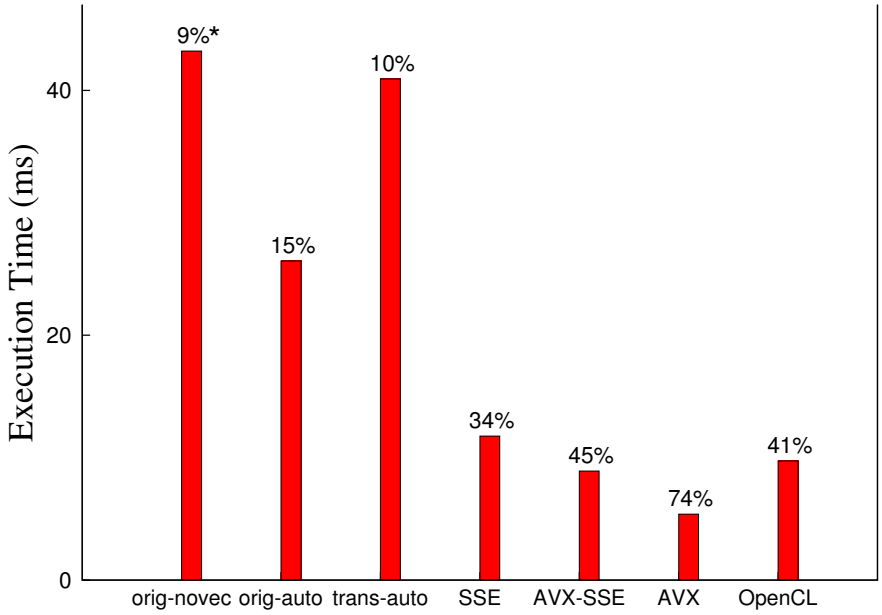


Figure 3.3: Performance comparison of filter kernel in different paradigms (on Desktop); *- percentage of peak performance.

As the figure shows, the auto-vectorized codes (*orig-auto* and *trans-auto*) run significantly slower than the OpenCL code (26.06 ms and 40.93 ms versus the 9.74 ms time of the OpenCL

code). Checking the generated assembly code of *orig-auto*, we see that the compiler has generated vector code but it is not as efficient as our manual code since it could not perform our transformation automatically (which is expected). The figure also shows that the performance of *trans-auto* improves only slightly with respect to *orig-novec*, because, although simple, the compiler cannot analyze the transformed code. Thus, auto-vectorization, although easy to use, is not a good solution in terms of performance, even for the simple loops of our *filter* kernel.

With respect to the vector codes using intrinsics, *AVX* is the fastest code, and is 46% faster than the OpenCL code. As expected, *SSE* is the slowest of the codes using intrinsics, due to the shorter vector units. However, the SSE code is the shortest, as the number of filters (100) is now an exact multiple of the SIMD width (4 elements). Thus, wider SIMD units increase the overheads of handling boundaries. Finally, the *AVX+SSE* is only 8% faster than the OpenCL counterpart.

For comparison purposes, we implemented, using AVX, a filtering kernel without branches and comparisons (that does not find the index of the best filter). It uses a small "reduction tree" method to perform fewer SIMD max operation to find the maximum from the eight responses. Our *AVX* version is just 5% slower than this one, showing that we have alleviated most of the comparison overheads by comparing in parallel and pushing other instructions inside the "if" statements.

Programming Effort Comparison

To quantify the programming effort of each paradigm, we use the Halstead's productivity metrics [55–57]. In Halstead's Elements Software Science [55], a program is considered as a string of tokens, which can either be *operands* or *operators*. The operands are either constants or variables, while the operators are symbols or their combination that can affect the value or ordering of operands. Let η_1 be the number of unique operators, η_2 the number of unique operands, N_1 the number of occurrences of operators, and N_2 the number of occurrences of operands. Derived metrics that can be used to quantify programming complexity are defined as follows:

Program Volume: $V = (N1 + N2)\log_2(\eta_1 + \eta_2)$

Program Difficulty: $D = \frac{1}{2}\frac{\eta_1 N_2}{\eta_2}$

Programming Effort: $E = DV$

Table 3.4 shows these metrics for different implementations of the filter kernel. The last column of Table 3.4 shows the number of Source Lines of Code (SLOC). The Table shows that performance is correlated with effort; higher performance requires more programming

effort, which is to be expected. *AVX* has the highest performance and effort, while compiler auto-vectorization has the least of both. From these numbers, OpenCL Programming Effort and Program Volume metrics are similar to those of AVX-SSE; both deliver also similar performance. The table shows that η_1 is almost the same for all the code versions, while η_2 is 61 for orig-auto and around 100 for all the others. These additional variables appear as a consequence of unrolling, that has been applied to all code versions but orig-auto. N1 and N2 are also larger in *AVX* because it needs some code to handle the leftovers after loop unrolling. In addition, the code to compute the index of the filter with the maximum response is more complex, as described in Section 3.3.2.

Notice that all these metrics do not fully capture the complexity of each code. They are based on the number of operators and operands, but do not take into account the complexity of each operator. For example, addition is much simpler than a vector intrinsic function of AVX. Thus, these metrics may be highly optimistic for the vector implementations.

Table 3.4: Software metrics for different implementations of the filter kernel.

Paradigm	η_1	η_2	N_1	N_2	V	E	SLOC
orig-auto	23	61	230	214	2838	114504	68
SSE	23	102	494	467	6694	352458	133
AVX-SSE	23	101	682	645	9228	677725	187
AVX	24	106	903	836	12211	1155752	212
OpenCL	22	99	691	625	9105	632307	162
OpenCV	12	25	59	63	635	9609	15

Overall, OpenCL provides a good balance in programming effort and performance. It is much faster than the auto-vectorized versions and it is close to the low level intrinsics versions. It is 1.8 times slower than the *AVX* code with "ninja optimizations" but the effort is significantly less (1.8 times less Halstead Effort). Therefore, programming in OpenCL is effective for the CPU as well as the GPU, and bringing the GPU on the die did not impose significant programming effort (since the same code runs on the CPU as well). Thus, OpenCL has the advantage that a single programming model can be used to run in both CPU and GPU. It is possible that the code versions that run the fastest will be different among platforms, but the programming effort does not increase significantly, because the different versions need to be tried in both platforms in the optimization and tuning process anyways.

Note that we do not claim OpenCL is performance portable across platforms in general. We believe that given the data parallel nature of vision algorithms, in many cases, the

same baseline algorithm can be written for CPU and GPU in OpenCL. However, tuning transformations need to be evaluated separately for each device. For this study, our target of the OpenCL tuning was the GPU, but the experimental results show that the transformations also worked for the CPU, resulting in the same kernel codes.

3.3.5 Possible Hardware and Software Improvements

Vision and video analytics (and their filtering kernels) are important applications for heterogeneous on-chip architectures. Thus, we list a set of possible improvements to the hardware and system software that vendors might consider for this class of applications.

The first one is related to the algorithms that the compiler can recognize and vectorize automatically. We observed that neither the Intel compiler nor the OpenCL compiler can generate efficient vector code for the max reduction (and finding the index corresponding to max) used in the filter kernel. When we examined the assembly code, we found out that the OpenCL compiler generates permutations and comparisons similar to our *AVX+SSE* version. However, the compiler should be able to automatically generate more efficient code [58, 59], following a similar approach to the one in the *AVX* code evaluated in Figure 3.3.

The second one deals with a common operation in this type of kernels. We have observed that multiply and add operations are used together extensively. Thus, Fused-Multiply-Add can improve the performance significantly. The "FMA" vector extension addresses this point, which is available in some new processors (such as the ones using Intel Haswell micro-architecture).

Our transformation optimizes filter kernels significantly but they could become even faster with more hardware support. Finding the maximum and corresponding index in a vector is a *reduction* across the elements of a single SIMD vector, or a *horizontal max* operation (in Intel's terminology). In current SSE and AVX standards, there are a few horizontal operations, such as an addition that just reduces two adjacent elements. This could be further extended to perform a full reduction, which will improve multimedia applications in general [60–62]. In fact, to estimate how much improvement we can achieve with a reduction instruction, we replaced the instructions to find the maximum response in our *AVX* kernel with just a horizontal add instruction. This improved the performance by more than 34%. Thus, more targeted hardware support can lead to significant improvements in future machines.

3.4 Application Performance and Energy

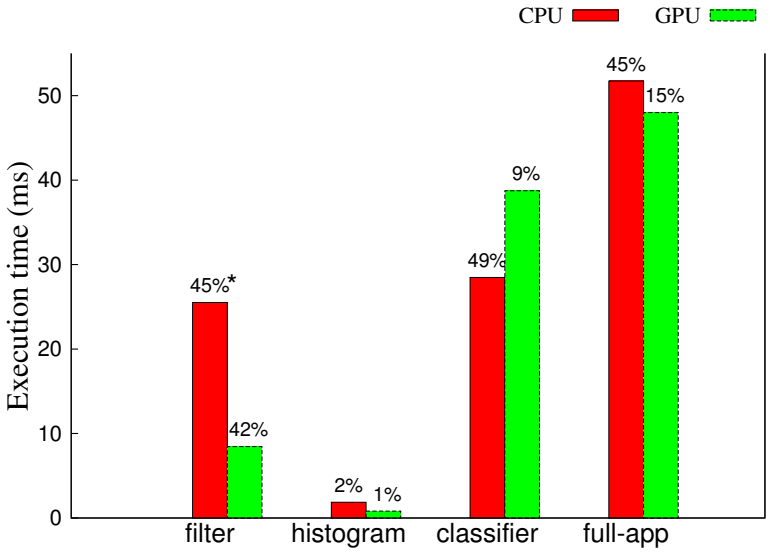
This section evaluates and analyzes the execution time and energy consumption of the kernels described in Section 3.1.1 and optimized in OpenCL in Section 3.2. We also evaluate and analyze different mappings for our application on CPU and GPU for better performance and energy efficiency.

Figure 3.4(a) shows the execution time of each kernel on the CPU and GPU. *full-app* shows the results for all the three kernels running on either the CPU or the GPU. The GPU is about 3 times faster for the filter kernel and 2.3 times faster for the histogram one. However, it is more than 1.3 times slower for the classifier kernel. Note that *classifier* performs less floating point operations per data element (see related work for analysis of data-parallel kernels on CPU and GPU [50]). For the full application, the GPU is slightly faster (less than 8%) than the CPU.

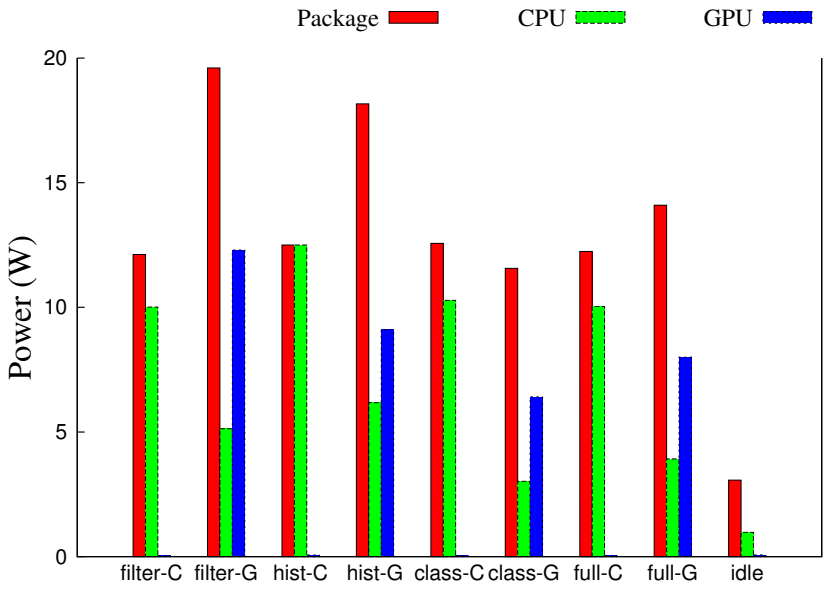
Figure 3.4(b) shows the power consumption of the processor for each individual kernel and for the full application running on either the CPU or the GPU. Each setting is labeled after the running code and the architecture it is using. For instance, "class-G" means that classifier kernel is running on the GPU. Each setting has three power consumption bars. We also show power numbers in idle state. The red (left) bar is the power consumption of the whole processor chip (CPU, GPU, memory controller, etc.), while the green (middle) bar is just the CPU's consumption and the blue (right) one is just the GPU's. Note that we report the average power consumption over a period of execution (see Section 3.1.6).

We mostly consider the power consumption of the whole package (the red bar), as it corresponds to the cost one would pay. However, the power breakdown can give insights about some important aspects of the system. For instance, when the code is running just on the GPU, the CPU is still consuming considerable power. The reason is that the CPU and the ring interconnect are in the same voltage and frequency domain [49] and the interconnect cannot be idled, since the GPU needs to connect to the last level cache (LLC). Addressing this issue may lead to significant savings in power consumption when the application is only using the GPU. The reason is that, for instance, the CPU domain consumes 3W (with probably a notable part contributed by the cores) from the 11.5W total package power when the classifier is running on the GPU. On the other side, it consumes only 0.7W in idle state.

As shown in Figure 3.4(b), the GPU consumes more power than the CPU in all cases (e.g. comparing left bars of filter-G and filter-C) except classifier, which is not unexpected since GPU has higher peak performance as well (See Table 3.1). However, GPU's power consumption varies depending on the workload. For instance, classifier consumes around 11.6W, while the filter consumes about 18.2W (around 36% difference). This is because the



(a) Execution time



(b) Power consumption

Figure 3.4: Execution time and power consumption of kernels (on Ultrabook); *- percentage of peak performance.

filter keeps the GPU almost fully occupied while the classifier does not have full utilization. On the other hand, the CPU's consumption has less than 0.5W variation across the board, even with its complex architecture and power management schemes. The reason is that

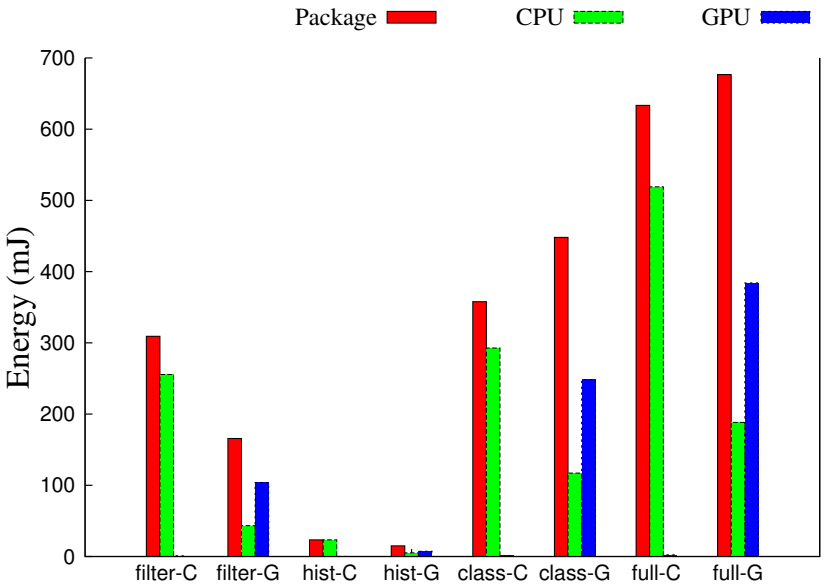


Figure 3.5: Energy consumption (on Ultrabook).

all the kernels can keep it occupied, partly because of its adapting architecture and partly because it is not very powerful.

So far, we have seen that the GPU is faster but it consumes more power. Since one factor is in favor of energy but the other is against it, we need to look at the energy metric. Figure 3.5 compares the energy consumption of the different kernels and full application on the CPU and the GPU. For the full application, the package consumes slightly more energy when running on the GPU (less than 7%), while it varies across different kernels. If we look at the package consumption, the GPU consumes 36% less energy for histogram kernel and 46% less for the filter kernel. However, it consumes 20% more energy than the CPU for the classifier kernel. This, as pointed by others, contradicts the general belief that the GPU architecture is more energy efficient for every highly parallel kernel [50]. The net energy is in favor of the CPU, since the classifier kernel is time consuming on the GPU.

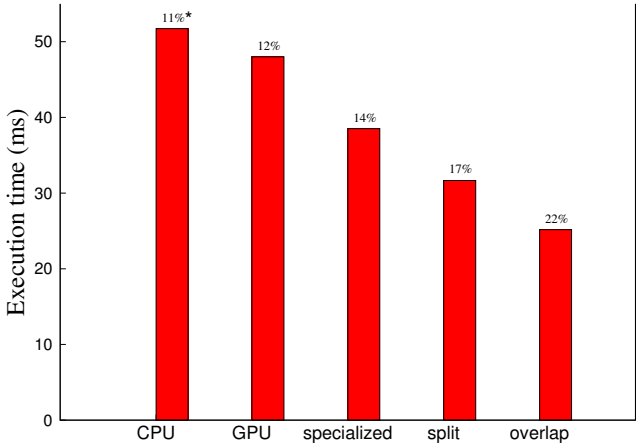
In a nutshell, running the three kernels in ViVid on the integrated GPU of the Ivy Bridge Ultrabook is faster but consumes more power and energy. On our Desktop system, with the same input size, the GPU is about 5 times slower and 3 times less energy efficient than the CPU for *full-app*, because it is small (so not very powerful in computation) but keeps the resources of the system busy. Thus, the balance of the system needs to be considered for portable devices that run vision applications. Comparing across platforms (but same processor type), the CPU of the desktop machine is about 2.5 times faster than the Ultrabook one for the *full-app*, but 19% less energy efficient.

3.4.1 Mapping Strategies

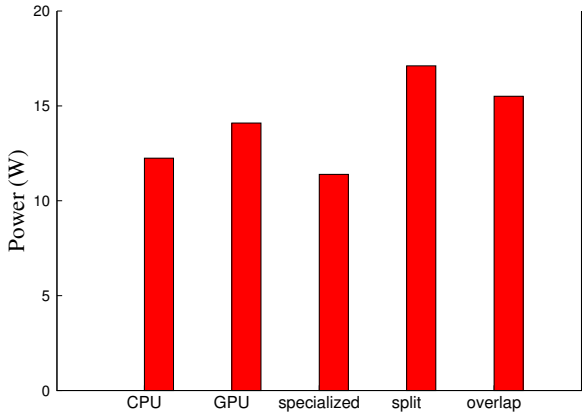
After understanding the different trade-offs between GPU and CPU for each kernel, the natural question is how to utilize the heterogeneous system for an application to achieve better performance and energy efficiency. Other than just running the code only on the CPU or the GPU, one could also try to map different kernels to the device where they run more efficiently. Figures 3.6(a), 3.6(b), and 3.6(c) show the execution time, power and energy of different approaches, respectively. *CPU* and *GPU* correspond to running all the kernels in the CPU or in the GPU. *Specialized* corresponds to an execution where the filter and histogram are mapped to the GPU (where they run faster and more energy efficient) whereas the classifier is mapped to the CPU (where it is faster and more energy efficient). In *specialized*, when the GPU is executing filter or *histogram*, the CPU is idle and vice versa (when the CPU runs the classifier, the GPU is idle). *Overlap* corresponds to an execution similar to software pipelining. It can be applied to streaming applications where parallelism can be exploited across multiple input images or frames, like multiple frames of a video. When using *overlap*, filter and histogram form the first stage of the pipeline operating on a frame in the GPU, while classifier is the second stage of the pipeline running on the CPU and operating on the GPU's results. Note that with *overlap*, since the CPU's work takes around 3 times more than the GPU, the GPU will be idle for about two-thirds of the execution time. Note that the strategies so far will under-utilize either the CPU or the GPU because of data dependencies. Therefore, one could split the image between the CPU and the GPU for maximum utilization, shown as *split* in the figures. Since the execution time of the application is almost the same for the CPU or the GPU, we split the image in half for our experiments.

When analyzing these strategies, one needs to keep in mind that this architecture has a dynamic power management scheme (Intel Turbo Boost 2.0 technology). It determines a fixed power budget at each time based on the temperature and assigns frequencies to the CPU and the GPU accordingly [49]. Thus, for example, the CPU and the GPU are slower when they are running together as opposed to when the other is idle.

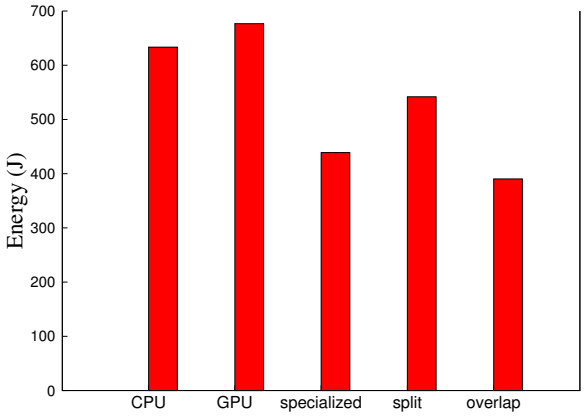
Figure 3.6(a) shows the execution time of different strategies for the full application. *Specialized* is more than 25% faster than just running on the CPU (20% faster than the GPU), as one would expect. *Split* is about 39% faster than *CPU*, but it could be up to twice faster if the system did not have dynamic power management. *Overlap* obtains the best performance by running the kernels on the best type of processor, but trying to keep them more busy by software pipelining. It should be noted that, for our Desktop system, *split* did not result in any performance improvement comparing to *CPU*. This is because the



(a) Execution time



(b) Power consumption



(c) Energy consumption

Figure 3.6: Running full application on CPU or GPU or utilizing both using different approaches; *- percentage of peak performance.

GPU is much slower (5 times than the CPU) and the overheads of using it dominate. Thus, the balance of the heterogeneous systems seems important for these applications.

Figure 3.6(b) illustrates the power consumption of different strategies. *Specialized* has the least consumption, while *split* consumes the most. As one expects, *overlap* consumes more than *specialized*, but less than *split*, because its resource utilization is in between the two. Note that power consumption does not necessarily correspond to execution speed here.

Figure 3.6(c) shows the energy consumption of each strategy for an input image. *specialized* and *overlap* consume the least energy because they run each kernel where it runs the best. On the other hand, using only the CPU or the GPU is not energy efficient. Note that *split*

is a very fast method but it consumes much more power also, so it is not the most energy efficient in the end. *specialized* and *overlap* are 35% and 42% more energy efficient than GPU only method respectively. They are also 19% and 28% more energy efficient than *split* respectively.

Summary Overall, our results show that to minimize energy consumption in these heterogeneous devices, one should try to exploit parallelism across devices and each kernel should be mapped to the device where it is more energy-efficient. Execution time should not be the only factor used to determine how to map an application, because the different devices have different power consumptions, resulting in different overall energy (e.g. *specialized* is slower than *split*, but more energy-efficient). Thus, our *overlap* approach where parallelism is achieved through software pipelining seems the best strategy for these type of on-chip heterogeneous architectures. However, a couple of points need to be considered when choosing this strategy. First, it is desirable to have pipeline stages with similar execution times, as the execution time of this scheme is determined by the execution time of the longest kernel. Note that our application's stages do not have similar execution times but this strategy is still the best. Second, this approach requires to have more on-the-fly data. In our case, since the pipeline only has two stages we have two frames on-the-fly (as opposed to one). In addition, since kernels execute in different devices, the frames need to move from device to device (in contrast with the *split* mechanism, where the data always stay in the same device). Since most of the vision applications (including ours) are very compute-intensive, data movement usually is amortized easily by the numerous computations required per data element.

Our performance is superior or similar to recent works using much more capable discrete GPUs [35,36,63]. However, notice that real time vision applications need to run at a certain number of frames per second. For instance, we can run at around 40 frames per second (fps) with *overlap* and 31 fps with *split*, while 10 fps might be enough for many object detection purposes. Applications requiring real-time object detection (33 fps) can use the OpenCL code on this architecture. The extra available computation power can be used for more analysis or for other applications (e.g. if vision is only the interface for some other purpose). Note that when maximum performance is required (e.g. needed fps cannot be reached), one might need to trade energy efficiency for performance (e.g. *specialized* versus *split*, when *overlap* cannot be used).

One might need more compute-power for future applications. Our experiments show that scaling the number of GPU's CUs is effective. As noted in Section 3.1.5, the Ultrabook's GPU has 16 CUs, while the Desktop's GPU has 6 CUs, with similar architecture and frequencies.

For all the kernels (as well as the full application), we see more than twice speedup on the Ultrabook one, which supports the scalability of the architecture for these applications.

3.4.2 Saving Energy with DVFS

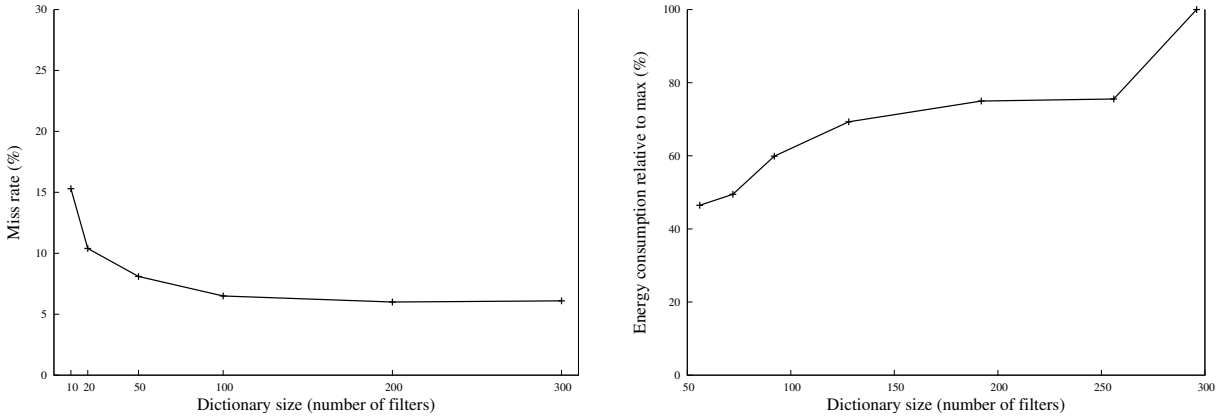
We saw that we can reach a detection rate that is more than enough for many applications. Thus, one might consider Dynamic Voltage Frequency Scaling (DVFS) for saving energy. However, Ivy Bridge processor's DVFS does not seem to be effective for these compute-intensive codes. We applied DVFS to our application and we could only save at most 5% of the energy, while sacrificing 9% performance. The reason is that it makes the runtime so much longer (for compute-intensive codes) that it offsets the power savings. Thus, running the application for a while and then idling the processor seems to be the best solution for saving energy. In this case, savings will depend on sleep and wakeup latencies of the processor in the specific usage.

However, we expect DVFS support to improve significantly in future devices, as vendors consider it in earlier steps of the processor design. For instance, when Near Threshold Voltage (NTV) processors become available, DVFS will save much more energy [64]. This will be very important for energy efficiency of many vision applications similar to ours.

3.4.3 Trading Accuracy for Energy

The visual descriptor we use is based on a model where the appearance of each 3×3 patch is characterized by finding its closest neighbors in a pre-determined dictionary of 3×3 patch templates (filters). Naturally, larger dictionaries can capture wider varieties in appearances of patches. In the case of detection problems, this results in an increased modeling power for discriminating the appearance of the objects of interest, versus the appearance of all other structures in natural images. However, as the size of the dictionary grows, more training samples are necessary to fully utilize the dictionary's modeling potential. Thus for a given dataset, one can expect the detection performance to saturate at a large enough dictionary size, which we observe at around 150-200 item dictionaries in our example application. Figure 3.7(a) shows the miss rate of our object detection algorithm as a function of dictionary size (from previous work [33]). We chose 100 filters (dictionary size of 100) in this chapter since it provides enough accuracy for most applications [33].

However, since energy is a major constraint in portable devices, one might want to trade some accuracy for energy savings when the battery charge is low. In our application, accuracy



(a) Object detection miss rate

(b) Relative energy consumption

Figure 3.7: Accuracy vs. energy consumption.

is determined by the dictionary size (number of filters) as mentioned. Furthermore, the work of the algorithm also depends on the dictionary size. Thus, the number of filters might be a "knob" for the system to save energy according to the energy status of the device (battery charge) at runtime.

Figure 3.7(b) shows the relative energy consumption when using different dictionary sizes. From this figure and Figure 3.7(a), one can conclude that we can save approximately 20% energy by going from 100 filters to 70 filters, which increases miss rate only by around 1%. This is a good tradeoff of energy and accuracy for many situations.

In a nutshell, we have considered four different techniques for better energy efficiency: 1) item Program optimization, 2) Heterogeneity, 3) DVFS, and 4) Decreasing accuracy.

3.5 Related Work

Previous works have shown that the use of heterogeneous architectures can improve performance and decrease energy consumption [29]. In addition, mapping algorithms have been studied for heterogeneous systems [65–68]. Other forms of heterogeneity, such as off-loading virtual machine services (e.g. garbage collection) to smaller cores [69], has also been studied. However, the focus is mostly on mapping across different applications. In addition, integrated GPUs have not been considered.

A few programming paradigms such as OmpSs [70] or starPU [71] provide a unified programming paradigm for the CPU and the GPU and automatically perform load balancing

and move the data as needed between the different nodes and GPUs. We restricted ourselves to OpenCL, since it is the only one supported by the integrated GPU in the Ivy Bridge machines. In addition, these works focus on programmability and performance, but not energy efficiency. Also, they focus on large systems, rather than on-chip heterogeneous systems. Moreover, they do not focus on pipeline applications. Furthermore, as we have shown, automatic vectorization does not achieve high performance in our case.

The new architectures with on-chip GPUs are becoming increasingly more popular in industry. These platforms include Intel's Ivy Bridge [30], AMD APU [31], and NVIDIA Tegra 250 [32]. Evaluation studies also show their advantages in performance and energy efficiency [72–75].

With regards to computer vision, it is known that GPU is very effective [63,76–79], because of the data parallel nature of most vision computations. However, as shown, integrated GPU's have different trade-offs and a GPU-only solution is not efficient here [50]. Our code has very high performance comparatively, and we gain much better or similar fps detection rate compared to recent works on object detection, which use much more capable discrete GPUs [35,36]. For example, 41 fps had been reported using a desktop machine with an Nvidia GTX 260 GPU card [35], while we achieve 40 fps on a portable device with an integrated on-chip GPU (although comparison is complicated, since the algorithms and machines are different).

Furthermore, trading accuracy for energy or performance has been considered, but in different contexts [80,81]. For example, Bergman [81] shows how to limit the processing times for rendering graphics by an OpenGL API library. This method sacrifices frame rate or image quality for less energy consumption. In addition, Sharrab and Sarhan [80] adapt the video rate for computer vision applications considering both accuracy and power consumption. To gain insight about the accuracy of object detection algorithms, we encourage the reader to go through surveys on similar topics [82–84].

3.6 Conclusions

Driven by user demand, the computer industry is focused on battery operated portable devices, which are energy constrained. In addition, better user experience requires natural interfaces using vision and video analytics applications. However, energy efficient execution of these compute-intensive workloads is challenging.

We showed that heterogeneous on-chip architectures can be very effective, using a visual object detection application. We optimized each kernel for CPU and integrated GPU of

the Ivy Bridge architecture using different techniques. For example, we vectorized the filter kernel using a data layout transformation.

Furthermore, we showed that a unified programming paradigm such as OpenCL provides a good balance between performance and programmer productivity. This is because the same code runs efficiently on both the CPU and the GPU.

In addition to productivity and performance, energy efficiency is a main concern. By comprehensive evaluation, we showed that it is best to map each kernel where it runs the best. Thus, existing methods, which only use the GPU or try to gain maximum utilization of both the CPU and the GPU naively, are inefficient (even for highly parallel vision workloads). In a nutshell, running each kernel on the best processor type, and using software pipelining is both faster and more energy efficient. This is because these heterogeneous on-chip architectures have a fixed chip power budget, which is allocated by a dynamic power management scheme to each processor. If parallelism through software pipelining is not possible, splitting the input among CPU and GPU might be faster, but specializing each processor for suitable tasks can be more energy efficient.

CHAPTER 4

Adaptive Cache Hierarchy Reconfiguration in Adaptive HPC Runtime Systems

Power- and energy-related issues are of growing concern in computer systems. The number of transistors on a single chip has already surpassed one billion and continues to increase. Although semiconductor processes can give us ever more transistors, thermal dissipation and broader power and energy constraints will limit their use.

The cache hierarchy has potential for many power and energy efficiency innovations. A significant fraction of the power used by a processor chip is consumed by the cache hierarchy. For example, caches in IBM's POWER7 consume around 40% of the processor's power [85]. Yet, the caches may not be utilized equally in various Computational Science and Engineering (CSE) applications and even across different phases of a single application.

To exploit this fact, we have developed a scheme that saves energy by using the runtime system (RTS) to selectively turn off parts of the caches. Our approach takes advantage of common characteristics of workloads found in CSE. We also leverage adaptive RTSs that include an introspective component that is aware of both the current hardware status and the application.

In this chapter, we characterize HPC platforms and applications and find common patterns of cache utilization. We then use these patterns to develop a novel, adaptive RTS-based scheme that automatically turns parts of the caches on or off to save energy. Our scheme also switches the cache to a streaming organization depending on the application's behavior. In this case, the runtime reconfigures the streaming parameters for best performance and energy efficiency.

Our scheme addresses major limitations associated with other methods that reconfigure the caches. It uses *persistence* and formal language theory to express the application's

pattern, and the Single Program Multiple Data (SPMD) model to find the best configuration concurrently. This approach is practical since it only requires minor hardware support.

We evaluate our scheme using cycle-level simulations of a chip multiprocessor running the Mantevo mini-apps suite [86] and real applications such as NAMD [19] and MILC [87]. Our results indicate that 67% of cache energy can be saved on average, with only a slight performance penalty. We also demonstrate that adaptively switching to a reconfigurable streaming organization for the L3 cache (prefetching cache lines for detected memory-access streams) can improve both performance and energy efficiency with various tradeoffs. For example, performance can be improved by 30% while saving 75% of cache energy consumption.

The contributions of our work can be summarized as follows. We analyze the memory access characteristics of common HPC applications using the inherent properties of scientific domains and algorithms. In contrast to previous HPC characterization works [88–91], we consider all the relevant aspects from algorithms to hardware. We also examine the capabilities of HPC runtime systems, and the related energy reduction possibilities in caches. Taking into consideration all the involved system components, we propose a novel cross-layer solution for adaptive cache reconfiguration. We also propose a software-controlled reconfigurable streaming scheme that can improve performance and energy efficiency for many common applications. Our proposals are highly practical since the RTS is easy to change, and the hardware complexity does not increase significantly. To the best of our knowledge, this is the first work to use HPC runtime systems to reconfigure the cache hierarchy for energy efficiency.

This chapter is organized as follows: Section 4.2 and Section 4.3 discuss the necessary background and common patterns in HPC applications and architectures. Next, Section 4.4 introduces our RTS-based scheme. Section 4.5 explains our evaluation methodology and presents the results of our scheme for cache reconfiguration. Section 4.6 explains our reconfigurable streaming scheme and presents the results and their analysis. Section 4.7 discusses the related work and we conclude the chapter in Section 4.8.

4.1 Background and Motivation

Cache reconfiguration (*cache tuning*) has been extensively studied, because of the high energy consumption of caches [92–97]. Dynamic hardware-based methods need to 1) monitor the application to predict the future, and 2) find the best cache configuration effectively. Both of these stages have considerable performance and energy overhead [98], eliminating the benefits of reconfiguration. For example, the hardware could monitor some system metrics

such as Instructions Per Cycle (IPC) and cache miss rate in a short interval and choose a new configuration. However, there is no guarantee that the interval is representative of the application execution, especially for HPC applications, which typically have long iteration times. In general, phase change detection is known to be challenging. Furthermore, the chosen configuration may not be the best possible, and good design space exploration heuristics are difficult to design for complex modern processors. In Section 4.6, we analyze a typical case where the miss rate is decreased but the performance is degraded due to different, complex factors of a modern speculative processor. A survey by Zang and Gordon-Ross [98] explores the challenges. Because of these issues, these hardware methods have not found their way to modern processors, and some recent processors, such as Angstrom [99], rely on software to reconfigure their caches.

Cache reconfiguration by the compiler has also been proposed [100, 101]. Many assumptions are made for the required footprint analysis, such as having only simple nested-loops and affine functions for array indices (only constants and loop index variables are allowed). However, large-scale HPC codes are usually more complicated. Moreover, the hardware complexities mentioned previously can prevent the compiler from choosing good configurations. We argue that the RTS can perform this task much more easily and effectively.

As an example of current practice on modern HPC machines, let us consider the simulations that were used for a recent scientific discovery at Illinois. Researchers used NAMD to simulate an HIV molecular system with 64 million atoms. Considering the maximum possible cache utilization with all the read only data (e.g. structures of atoms) and transient data (multicasts of force calculation results), only about 400 bytes per atom are needed. Therefore, the application uses only 25.6 GB of data in the working set. Note that in NAMD, the main data being updated in each iteration are the position and velocity of the atoms, which need only 48 bytes per atom combined. A typical simulation uses about 4000 Cray-XE nodes of Blue Waters (each containing two AMD Interlagos processors) with a total of 256 GB in L2 and L3 caches. Thus, more than 90% of the cache capacity was not used for the simulation. Each simulation took more than 16 days of wall clock time, which translates to a huge waste of power in the caches. Section 4.2 explains why the algorithms of this particular class of HPC applications do not need large caches.

4.2 HPC Systems

4.2.1 Provisioning Practices

Machines in HPC data centers are used very differently than non-HPC ones. Usually, there is no multi-programming or time-sharing of different jobs. In addition, there is no co-location of different jobs on the same nodes. Therefore, each node is dedicated to a single job at a time.

Furthermore, there is no migration of jobs across nodes. A set of nodes is dedicated to a single job for its entire execution time, which is usually much longer than the execution times of non-HPC jobs. Note that capability supercomputers usually try to run long jobs with large allocations (especially on the full machine) to facilitate new and significant scientific discoveries. On the other hand, non-HPC data centers run short and small jobs (e.g. search queries) that can be migrated using virtual machines. Thus, HPC machines are simpler to analyze and there is much more predictability and persistence in HPC data centers that can be exploited.

The processors in current supercomputers are often commodity chips. The reason is that designing and manufacturing a processor is a large investment that needs larger markets. Thus, most processors used in HPC are designed for commercial workloads in various environments, which can be very different than HPC workloads. This can result in inefficiencies of both power and performance in HPC environments. However, as we demonstrate in this chapter, these can be overcome with minor support for HPC.

4.2.2 Applications

Common HPC applications are usually iterative and *persistent*, meaning that their computation and communication patterns tend to persist over time. They perform roughly the same (or very similar, at least from a memory access pattern perspective) computations and communications in each iteration. Each simulation consists of thousands to millions of these iterations (each iteration might be structured and have phases in itself, which is discussed later). For example, to simulate a bio-molecular system (e.g. in NAMD [19]), forces need to be integrated for every one (or few) femtosecond(s) of simulated time. Therefore, a one microsecond simulation of a bio-molecular system takes one million iterations. Even though the simulation is dynamic and the molecules and atoms might move across regions, the computations are roughly the same. Thus, many scientific and engineering applications follow the *principle of persistence*. This means that the computation and communication tends to

persist or change slowly over time. This principle allows the RTS to predict the future of the application and has led to many successful features, such as measurement-based load balancers [10]. All the mini-apps in the suite we consider are iterative and highly persistent.

Some HPC applications (e.g. stencils and matrix-vector multiplies) are memory-bound and have lower temporal locality but higher spatial locality than other workloads. This property is studied extensively in the literature [88, 90, 91, 102]. For example, a sparse matrix vector multiply (SpMV) kernel sweeps the matrix and vector linearly and there is a high chance of accessing neighboring values. However, if the matrix and vector inputs are larger than the largest cache, the data will not be present in the cache for the next iteration. For example, a physical domain with 100^3 grid cells per processor[1] and 240B of data per cell (for different attributes such as velocity, energy, mass, their derivatives, etc.) will occupy 230MB of memory, which should be updated in every iteration.

On the other hand, some applications have high temporal locality as well as spatial locality, but their working set (in typical execution runs) is usually much smaller than the cache hierarchy. Many Molecular Dynamics (MD) applications, such as NAMD, usually fall into this category. For example, 1000 atoms per processor with 80B of data per atom takes only 78KB of memory, which is only a small fraction of a typical Last Level Cache (LLC).

Most HPC applications follow similar memory access patterns. To study these patterns, we use the Mantevo suite's mini-apps as representative of common HPC applications. Mini-apps are simplified versions of applications that are of interest to the CSE community. They are designed to be similar to real applications from a computational perspective, and more representative than micro-benchmarks. The mini-apps of the Mantevo suite can be divided into three classes:

1. Stencil computations (CloverLeaf and MiniGhost)
2. Sparse Linear Algebra (HPCCG, MiniFE and MiniXyce)
3. Particle simulations, such as molecular dynamics (MiniMD and CoMD)

Stencil computations, which are key kernels of many structured grid applications and their PDE solvers, have limited data reuse. These kernels sweep through domain data structures that are typically much larger than caches and fill a sizable fraction of the main memory. Thus, they do not benefit from caches to the full extent. For example, Figure 4.1 illustrates a 5-point 2D stencil computation. In this example, the update of Point 3 uses Points 1, 2, 3, 4, and 5, which can potentially result in four cache misses. The next update, which is for Point

[1] by "processor" we mean a processor chip in this chapter (not a single core).

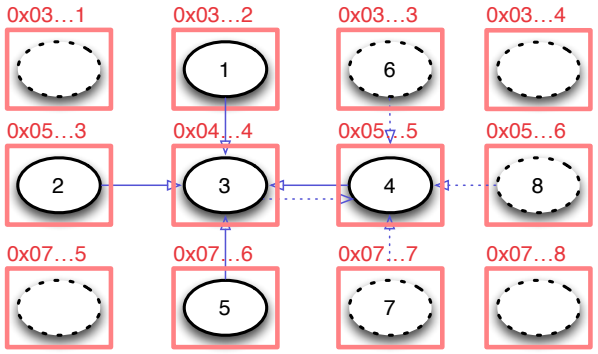


Figure 4.1: 5-point 2D stencil example: boxes represent memory locations, ovals represent stencil data points, and arrows indicate data dependencies.

4 (displayed with dotted lines), can reuse some of the data of the previous update. Therefore, a memory location is reused only a few times after its first use. Similarly, consecutive updates go through the top, middle and bottom rows of the data domain in the figure. From the memory hierarchy point of view, three different address ranges are being read ("streamed") with few reuses.

In essence, these classes of applications have much more spatial locality than temporal locality. Thus, streaming buffers (or other forms of block transfers) can be more effective than typical caches for stencil computations. Streaming strategies can capture more of the available spatial locality in stencil codes to hide memory latencies. Also, spatial locality is partially captured through the cache line in caches, and a smaller cache could be just as effective.

Note that cache tiling (blocking) optimizations reduce the dimension sizes, increasing the number of reuses. However, the block size needs to be tuned and does not need to fit the whole LLC, as previous work demonstrates [103]. Furthermore, many legacy HPC codes do not incorporate these optimizations, and the required programming effort is a burden. The applications and mini-apps we evaluate do not use tiling for stencils for the same reason. Programming paradigms such as Hierarchically Tiled Arrays (HTA) [104] can alleviate this issue.

Sparse Linear Algebra computations in HPC applications also have limited cache utilization if the domain is large enough. The HPCCG, MiniFE, and MiniXyce mini-apps use sparse linear algebra methods, such as Conjugate Gradient (CG) and Generalized Minimal Residual (GMRES). Most of the execution time of these methods is spent in matrix-vector and vector-vector operations. These kernels stream data from main memory without much reuse. For example, matrix-vector multiply kernels read the matrix only once in every itera-

tion. The vector accesses might also not have much data reuse depending on the structure of the matrix (e.g. a matrix from a regular 2D grid). In addition, if the matrix is large enough, the vector is evicted from the cache. Thus, consecutive addresses from a few address ranges are read regularly (from the memory hierarchy) for these kernels. Therefore, they have high spatial locality, similar to stencils, and the same arguments apply.

Many Molecular Dynamics (MD) and other particle interaction kernels are different than the previous categories and can have high temporal locality (as well as spatial locality). The reason is that the previous categories usually represent discretized points in the physical domain, while particle kernels represent entities. Each entity can have many interactions with other entities, while a point is fine-grained and usually interacts only with its neighbors. For example, in many particle kernels each particle (an atom in MD or a star in astrophysics) interacts with all other particles within a cut off distance. Hence, while each memory location is accessed $O(1)$ times in the other two classes of applications, it is accessed $O(n)$ times in particle applications (n is the number of particles in a cut-off), resulting in high data reuse. On the other hand, the data size for practical runs is typically smaller than caches of modern machines, especially the LLC. One reason is that the order of the computation time is roughly the square of the data size in the cut-off, making large input sizes impractical. Thus, large caches are not exploited to their full potential in many members of this class of applications either.

As we discussed earlier, cache effectiveness of common HPC applications is highly related to their per-processor working set size. Therefore, even a single application can have different cache utilization profiles depending on the input size and the number of processors used. Thus, there is no single cache hierarchy configuration that could fit all cases, providing the highest performance and energy efficiency.

4.2.3 Runtime Systems

In HPC environments, the RTS mainly mediates the communication and provides parallel services, such as message passing in MPI. This parallel management, in addition to applications' persistence, empowers the runtime system to provide other important features such as load balancing [10], fault tolerance [11], efficient parallel I/O [9, 105], and power management [12, 13]. Therefore, an adaptive RTS orchestrates a control system [106].

Our approach is based on the management of Sequential Execution Blocks (SEBs), which we define as sequential computations between two communication calls (e.g. MPI calls). The RTS has control before and after each SEB, but it cannot usually interrupt it. These SEBs

are repeated every iteration and they perform roughly the same computation (especially from a cache access perspective). For example, in most stencil codes the processors iteratively exchange the boundaries and update their values in an SEB. Thus, we try to adapt the caches to the SEB that is about to execute.

4.3 Cache Hierarchy

4.3.1 Cache Structure

Modern processors have multiple levels of very large caches to hide memory latency as much as possible. For example, the Intel Xeon E7-8870 [107] has 30MB of L3 cache in SRAM technology and IBM POWER8 [108] has 96MB of L3 cache in eDRAM technology. Architects try to incorporate larger caches to accelerate different workloads, while meeting the area, power, and latency budgets (e.g., it is critical in many designs to have only one cycle latency for L1 caches). Therefore, a large fraction of the silicon area is used by caches.

The cache hierarchies are designed for a diverse set of applications and hence, a fixed design might not be best for every workload. Furthermore, most supercomputers use commodity processors, which are designed for other (non-HPC) markets with different workloads. These factors result in immense waste of power and energy in supercomputers. For example, "big data" applications such as graph analysis might not have any locality because they are unstructured in their memory accesses (e.g. *pointer chasing* pattern). Thus, a level of adaptivity is needed to match the running application without too much hardware overhead.

4.3.2 Cache Power

Caches consume a large fraction of processor chips' power budget. For example, even with many advanced hardware power reduction techniques in place, caches in POWER7 consume around 40% of the total power [85]. The power consumption of caches depends on the technology, but our approach can help in most cases. SRAM technology has high leakage but is faster. On the other hand, eDRAM has much less leakage and higher capacity but needs to be refreshed [109]. Our approach can help with either technology.

Turning off ways of caches, used in our approach, can save the power consumption of various caches differently. In conventional caches, tag lookup and data access are performed in parallel for faster access. Therefore, considering that a large fraction of dynamic energy is consumed in data arrays, turning off ways of the cache saves significant dynamic energy

in addition to leakage (static) energy. On the other hand, the caches that are not on the critical path of the processor can be made so that tag and data accesses are done sequentially, accessing only one way after tag lookup [110]. Thus, turning off ways can only save leakage energy in this case. This usually applies to Last Level Caches (LLC), which consume a lot of leakage energy.

4.3.3 Architectural Opportunities

Modern set-associative caches are partitioned into multiple sub-arrays for performance reasons, and only minor hardware modifications are required to turn them off. Previous work proposes to do so, through simple changes to the cache controller and the addition of a register to let software turn ways off [110].

Most recent processors (and proposals) incorporate advanced features which let the software control various aspects of the processor similar to what we need, so our proposal is practical. The Angstrom architecture [99], which has been proposed for extreme-scale computing, allows the cache size to be changed by turning off ways and banks of the caches in software. The RAPL (Running Average Power Limit) [111] interface of recent Intel processor lets the software limit the power consumption of the chip among other features. The architectural support we need for our approach seems to be much simpler than RAPL.

4.3.4 Streaming

Some related proposals focus on streaming strategies as a cache efficiency technique for scientific applications [112]. In short, streaming relies on the spatial locality of HPC applications to load more data to reduce memory access latency. This can improve both performance and energy efficiency. In this work, we propose a unique software-controlled reconfigurable streaming scheme.

Streaming schemes strive to recognize the memory accesses with simple patterns (*streams*) and prefetch them using prefetch buffers. When a memory access misses in the cache, a stream is allocated and the cache blocks are prefetched starting from the missed target. Thus, subsequent memory accesses of the stream will have the data available in the cache. This method is simple, but it is usually effective for HPC applications because of the common patterns discussed in Section 4.2.

Since most HPC applications usually access multiple arrays in each loop (e.g. pressure and temperature at each grid point), using more than one stream is useful (i.e., *multi-way*

streams). When a memory access misses in the cache (i.e. it is not prefetched by the other streams), an old stream is flushed and a new stream is allocated starting at the miss address. We assume a Least Recently Used (LRU) policy to select the stream to be deallocated.

The *depth* of the stream is an important parameter. Streams should be deep enough to hide the memory latency for the subsequent accesses, but not *too* deep. If a stream is too deep, it competes with other useful accesses, potentially delaying them. Also, it can evict useful blocks from the cache. Thus, it can waste memory bandwidth and energy. We evaluate the reconfiguration of the depth of streams in Section 4.6.

It is important to filter isolated references, as allocating streams for them can waste memory bandwidth and energy. This is done with a small *history buffer* that stores the addresses of recent misses. A stream is allocated only when a miss to a block occurs next to a previously missed block (e.g. a and $a + l$). This also facilitates the detection of non-unit strides that are prevalent in some HPC applications (e.g. accesses to addresses of a, $a + d$, and $a + 2d$, for $d > 1$). A stream can have unit strides or non-unit strides depending on the application's memory access pattern.

In this chapter, we reconfigure the cache's (used as the streaming buffer) size and prefetch depth automatically to improve performance and energy efficiency significantly. This needs some hardware support (e.g., a small stream detection table), but it is relatively low cost. For the streaming implementation in this work, we use a previous work [112] that identifies a few streams of constant stride. However, we do not use a separate stream buffer and prefetch to the LLC instead (more details in Section 4.6).

4.4 Reconfiguration in Adaptive Runtime Systems

In this section, we introduce our approach for automatic way reconfiguration of the cache hierarchy. First, we present our baseline approach, which is enough for many common applications. Then, we formulate the general problem of application pattern recognition to incorporate more applications.

4.4.1 Overview of Our Approach

The RTS can easily identify common iterative patterns. This can be done by monitoring communication calls (e.g., MPI_Recv, MPI_Barrier etc.) to see if they are repeated regularly with similar arguments. The time between the matching calls is the iteration time and should be reasonably consistent. Even complex HPC patterns can also be expressed using Formal

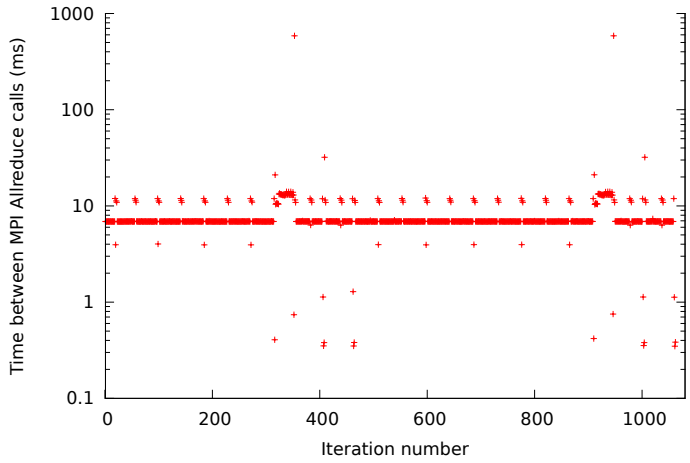


Figure 4.2: Time between calls to Allreduce in MILC.

Language Theory, as discussed later. Alternatively, some applications have calls to the RTS marking the end of each iteration, which are used for other purposes such as load balancing and fault tolerance. For example, calls that mark the best place for checkpointing are usually made between iterations.

Figure 4.2 presents the time between successive calls to MPI_Allreduce in MILC [87], which is a prominent code for Quantum Chromodynamics (QCD), running on a BlueGene/Q system. The *Allreduce* collective is called in the Multi-CG solve phases of MILC, which designates the CG steps. As can be seen, there is a clear regular pattern even for this sophisticated application. After removing the outliers, the average is 6.893 ms with a standard deviation of only 0.045 ms. In addition, the patterns for other phases of MILC are also regular (but with nearest-neighbor communication instead of collectives). Thus, even a sophisticated application such as MILC has a repetitive and regular pattern that is recognizable by the runtime system using the the time between communication calls.

Note that, just like other runtime adaptation mechanisms such as load balancing, we ignore the initialization and some iterations in the beginning of the execution. However, since initialization is usually a small fraction of the execution time, using the best configuration is not essential.

After identifying the iterative pattern and reconfiguration units (i.e., Sequential Execution Blocks (SEBs)), the RTS should ensure the iterations and their SEBs are the same across different processors. For this purpose, the runtime gathers some *characteristic information* about the execution of each SEB on different nodes, including execution time, instruction-based samples [113], and key performance counters. Then, reconfiguration is applied if the attributes are within a threshold on all nodes. This can be accomplished by collective calls

(e.g. Allreduce) that determine if the collected attributes are statistically similar (e.g. the minimum and maximum of the attributes are not too far from the average).

After finding the persistence pattern, the best cache sizes need to be found for each reconfiguration unit (SEBs or whole iteration). For example, the (2,1,2,4) configuration specifies that 2 ways need to remain active for L1D cache, 1 way for L1I cache, 2 ways for L2 cache and 4 ways for L3 cache. This is accomplished by applying and benchmarking different configurations on different processor in parallel to find the best one. We can map the configurations to sequential numbers and each processor can use its number (e.g. derived from MPI ranks) to know which configuration it needs to try. Different processors measure the execution time and energy consumption for some number of iterations and the minimum is reported by collective calls (e.g. Allreduce). The best configuration is then used on all the processors.

When the best configuration is applied, the RTS observes the execution of future iterations until the attributes become significantly different. Then, our method is invoked again to adapt to the change. Note that if the variation of the application is so high that our method could degrade performance, we switch to the default configuration (full size caches and normal policy), which is the "safest." In our experiments, we found that the runtime would not need to switch to the default configuration very often, but this is possible in the general case.

Our approach in the RTS can be summarized as follows:

1. Determine iterations (and relevant SEBs)
2. Ensure the SEBs are the same across processors
3. Run different configurations on different processors and find the best in performance and power/energy efficiency
4. Apply the best configuration to all processors
5. Observe the execution and repeat if behavior changes

Note that we depend on the fact that SEB characteristics are the same or similar on different processors. This follows from the Single Program Multiple Data (SPMD) paradigm assumed in most distributed memory parallel languages, such as MPI.

4.4.2 Generalization

Most scientific applications are structured: they can have multiple phases in each overall iteration, but these phases are also often iterative, forming a "hierarchical" iteration structure. For example, Figure 4.3 depicts different phases of MILC on four processors. This is a timeline diagram, where different phases (e.g domain updates with nearest neighbor communication, and CG solve) are color-coded differently. Note that the executions of four processors are stacked, but they appear very similar.

Using Formal Language Theory, the hierarchical iterative structure of an HPC application can be expressed as a *Regular Language*. We define each unique SEB as a symbol a of an alphabet Σ. Each application execution might have a different number of iterations and hence, is a *word* of the language.

Theorem 4.1. *A hierarchical iterative pattern is a regular language.*

Proof by construction. Each execution is a number of repeated iterations. Therefore, the pattern can be written as a regular expression of this form: $(a_0, a_1, ..., a_d)^*$, where each a_i is a regular expression for each (possibly iterative) component of each iteration. The regular expressions a_i can also be constructed in the same way, since each component is iterative as well. Following this procedure, in a finite number of steps, the whole regular expression can be constructed recursively. Hence, the language is regular, since it has a regular expression. □

The general problem of finding the application's pattern (to use for phase change detection) is a pattern recognition problem. Using our formulation, it can be modeled as a classical Formal Language Theory problem: *learning a regular language from text* [114, 115]. During the application profiling, we collect a stream of symbols that are from a regular language, and we need to infer the language.

In the profiling phase, we gather a string of symbols (*Sample S*) of the language by monitoring the SEBs. We need to infer the grammar to build a deterministic finite automaton (DFA). Recall that a DFA is a tuple $(\Sigma, Q, q_\lambda, F, \sigma)$ where Σ is a finite alphabet, Q is a finite set of states, q_λ is an initial state ($q_\lambda \in Q$), F is a set of final states ($F \subseteq Q$), and σ is a transition function ($\sigma : Q \times \Sigma \to Q$). For example, Figure 4.3 can be rewritten as a list of symbols: $a_0a_1a_2a_3...$

A simple solution is to use a *prefix tree acceptor (PTA)* [115, 116]. A PTA is a tree-like DFA that has all the prefixes of the sample as states, and is *strongly consistent* with the

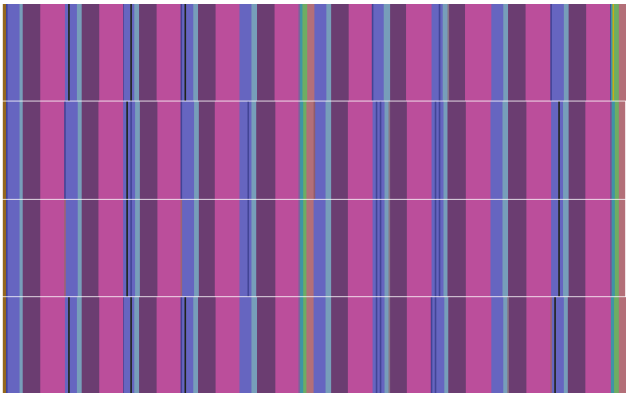


Figure 4.3: Timeline view of phases of MILC: time is on x axis and four processors are stacked on y axis. Colors represent different computations. This figure illustrates the regular iterative pattern of MILC.

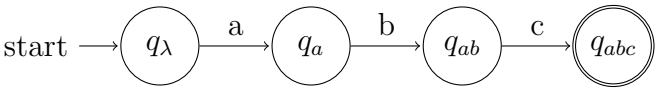


Figure 4.4: PTA for sample abc.

sample, which means that it only accepts the sample[2]. Algorithm 2 demonstrates how a PTA can be built from a sample. In essence, the PTA has one state for each prefix of the sample. For example, if the sample is abc, the PTA has a state for a, ab, and abc. The transition function has only one transition per state, which goes to the state representing the next longer prefix. For example, the state for a only goes to the state for ab. Figure 4.4 illustrates the PTA that is built by this algorithm for sample abc.

Algorithm: Build-PTA
Input: Sample S
Output: DFA A=$(\Sigma, Q, q_\lambda, F, \sigma)$
$F \leftarrow \emptyset$;
$Q \leftarrow \{q_u : u \in PREF(S)\}$;
for $q_{u \cdot a} \in Q$ **do**
    $\sigma(q_u, a) \leftarrow q_{u \cdot a}$;
end
$F \leftarrow F \cup \{q_S\}$;

Algorithm 2: Build PTA from sample.

Learning from text by a PTA can be challenging since the number of states can grow large. However, in practice, the number of SEBs that execute in the profiling stage is small.

[2]We have simplified the definitions and the algorithm for our purpose but in general, there can be multiple *positive* and *negative* samples of the language to learn from.

Furthermore, the number of DFA states can be reduced easily. For example, the application might have 1000 relaxation steps followed by 1000 CG steps in each overall iteration. This translates to 2001 DFA states since there these many prefixes in the string of the iteration pattern in our formulation. To reduce this number, we combine all of the CG steps together to form only one symbol since the same SEB is repeated. This fits our purpose since similar SEBs will have the same cache configuration. In this way, our example will have only three states in its DFA. Note that there are *state merging techniques* that can be used to merge *compatible* states (please refer to the references [114, 115]). However, for practical cases, the number of states is already very small after applying our technique.

The inferred DFA (equivalent to a regular expression) will be used for the rest of the application execution by the RTS to predict the future of the application. In this formulation, predicting the future of the application is similar to simple pattern matching of regular expressions. For instance, when the RTS is in state a and the next state is ab, the RTS predicts that the SEB that b represents will be executed next, so it changes the configuration to the best one for b before its execution.

Using our approach, the patterns of sophisticated HPC applications can be expressed by simple regular expressions. For example, NAMD performs three force calculation steps (a), before an FFT for long range force calculations (b). Therefore, the regular expression $(a^3b)^*$. MILC's pattern illustrated in Figure 4.3 seems more complicated, but it can be expressed as the regular expression $((a_0a_1a_2a_3)^5b_0b_1b_2b_3)^*$ using our method.

Some HPC applications consist of multiple potentially very different regular modules, but they can be handled similarly. Although the application might seem more complicated, the resulting pattern is still a regular language due to the following lemma:

Lemma 4.2. *The concatenation of multiple regular languages is a regular language.*

Therefore, the RTS will construct a single DFA, encompassing the execution of all the modules, and our approach is applied without any change.

In some applications, the processors are divided into logical groups that perform different computations, and this *application heterogeneity* needs to be taken into account to be able to apply our scheme in these cases. For example, in a climate simulation application, some processors might simulate parts of the physical domain that is inside a storm. Therefore, they might be performing different computations than the processors that do not simulate a storm. Hence, the cache hierarchy requirements might not be the same for all of the processors. In general, programmers (mostly in the SPMD programming model) can divide the processors into logical groups to perform different computations. To incorporate these cases, when the SEBs are not the same across different processors, the RTS can run a parallel

clustering algorithm that identifies similar processors based on their SEBs. For example, the processors simulating the storm might be running SEB a_2, while the others are running SEB a_1. In addition, the runtime can monitor the calls that are usually used by programmers for this purpose (such as MPI_Comm_split) as a hint. Then, the test of configurations phase of our approach is applied to each group separately. Also, each group can have a different DFA for pattern matching. Note that we do not require all the processors to have the same behavior. We simply need the pattern of each group to be regular.

In some applications, the phases can be slightly different across different overall iterations. As a hypothetical example, suppose the application has a CG phase in each overall iteration, and the number of CG iterations required for convergence can be between 995 and 1005. This is non-deterministic from the RTS point of view and can be modeled using a *Probabilistic Finite Automaton (PFA)* or similarly as a Hidden Markov Model (HMM) (please see references [117, 118]). However, this general formulation will increase the complexity of the problem and is not needed in our setting. To handle these cases, the runtime only needs to choose a "conservative" cache configuration when the next DFA state is not known. In our example, assume that the next phase needs a larger L3 than the CG phase. Therefore, the runtime chooses a cache configuration with a larger L3 for the last few iterations of CG (from iteration 995) because it does not know exactly when the next phase starts. This technique avoids performance degradation of the next phase.

The conservative cache configuration is constructed by examining the possible future states and setting the size of each cache level to the largest of the configurations. In our experiments, we found that this has negligible impact since only a small fraction of the execution time would need conservative configurations. Note that in some rare scenarios, the structure can slightly change (such as two SEBs running in switched orders in Charm++ [119, 120]), but these variations can be handled conservatively as well. Moreover, the runtime goes to a conservative cache configuration when there were too many mistakes in the DFA's predictions. This avoids overheads for the uncommon applications that are not persistent. In depth study of these cases is beyond the scope of this chapter.

4.4.3 Practical Details

The overheads of our scheme are very small compared to the overall application runtime and also can be measured and controlled easily by the RTS. The overhead of checking the configurations is negligible since it only impacts a few iterations. For many common applications, reconfiguration is done only once. For most others, the runtime reconfigurations

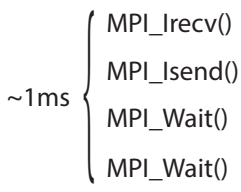


Figure 4.5: Different communication calls are combined if too close in time.

are rare due to persistence. Note that scientific applications usually run for a long time and for many iterations. For example, according to the available data (for several months) of the BlueGene/P installation at Argonne National Lab (Intrepid), the average runtime of a job was 5176 seconds (6817s for jobs larger than 8k core jobs, which are less likely to be test runs). Slowing down a few iterations (usually in the hundreds of milliseconds range) is therefore negligible.

The dominant hardware overhead for reconfiguration is invalidation traffic ("flushing") in the caches, which is added to the software overheads (system calls, calculations, table lookups, etc.) caused by the RTS. Only the modified cache lines of inactive ways need to be flushed before turning them off, which typically takes less than a microsecond (a few thousand CPU cycles). In addition, using an experimental module in the Charm++/AMPI system, we found the software overheads to be negligible. Therefore, the total overhead is usually on the order of microseconds, while SEBs are usually hundreds of microseconds. The programmer usually ensures that SEBs are long enough to amortize communication overheads, so SEBs are usually coarse enough for reconfiguration. Therefore even frequent reconfiguration can be practical.

Some communication calls need to be "combined" (considered as one call with no SEB in between) if they are too close together in time, since they are logically one communication step and they do not represent different SEBs.

For example, in a particular phase MILC repeats the pattern illustrated in Figure 4.5 for each neighboring processor before doing the actual local computation: These calls need to be considered as one call, since all of them happen in a short one millisecond interval, and there is negligible computation in between.

In general, the unit of reconfiguration should be selected judiciously. At one extreme, it can be as coarse-grained as the whole iteration (and hence reconfiguration is done only once). On the other hand, it can be as fine-grained as each SEB (or even finer than that if possible).

In practical settings, there might be minor timing variations of the SEBs on some proces-

sors (e.g. due to correctable ECC errors sometimes occurring). Therefore, the RTS needs to average SEB attributes of multiple iterations to smooth out these minor temporal effects. Moreover, the SEB timings and attributes do not need to match exactly. They only need to be within a threshold.

4.5 Evaluation of Runtime Cache Reconfiguration

4.5.1 Methodology

In this chapter, we use a diverse set of common scientific applications for evaluation of our scheme. This is important as previous work has demonstrated the importance of benchmark selection for cache access analysis [91]. Instead of micro-benchmarks, we use the Mantevo mini-app suite [86] and real applications, including NAMD [19] and MILC[3] [87][4]. We have confirmed that all of these applications follow the patterns we described in Section 4.2. Furthermore, all the processors execute the same or very similar SEBs (from the cache access point of view).

We also add an FFT benchmark to complement the molecular dynamics mini-apps, since their computation is usually simplified and includes only the time consuming short range force calculations. However, the long range force calculations can become significant depending on various parameter values. In NAMD, those forces are integrated every four timesteps using an FFT kernel. We use the NPB-FT benchmark to represent that FFT kernel.

For simplicity, we assume the MPI+OpenMP programming paradigm, which means that OpenMP is used for parallelization across each processor's cores. However, runtime systems of pure MPI programs and other paradigms can easily apply our method at the processor chip level as well.

For all the experiments of this section, we use SESC [121], which is a cycle accurate simulator. We simulate each unique SEB with different configurations, and find the wall clock time of each iteration for the whole application. The simulated system's parameters are chosen to be similar to real processors, and are presented in Table 4.1. We use CACTI [122] for modeling the power and energy consumption of the caches. We assume that the ways of the L1 and L2 caches are activated in parallel for each access (for less latency), while only one way of the L3 cache is activated for each access, since L3 is not in the critical path of

[3]We use su3_rmd in the MILC collection, which is usually used for benchmarking.

[4]These two applications are used by thousands of scientists on large-scale supercomputers, and were among the three applications used for the acceptance test of Blue Waters at Illinois.

the processor. Thus, turning off the ways of the L1 and L2 will save dynamic energy, while it will only save leakage energy in the L3 cache.

Table 4.1: Simulated processor's parameters.

Chip	8 Core CMP
Core	MIPS32, 4 issue out-of-order processor
Instruction L1 (L1I)	32 KB, 2 way
Data L1 (L1D)	32 KB, 4 way, WT, private.
L2	256 KB, 8 way, WB, private.
L3	16 MB, 16 banks, 16 way, WB, shared
Technology node	32 nm
Frequency	3.4 GHz

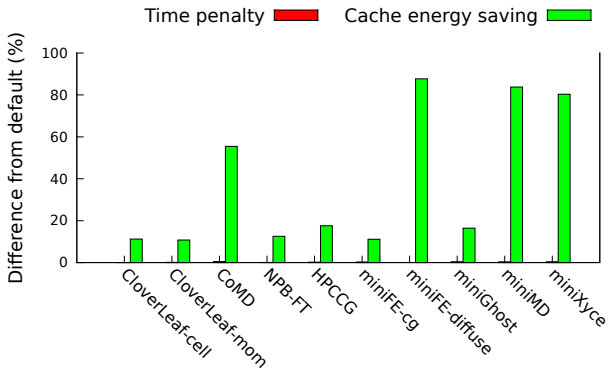
In this work, we consider the properties of the application domains for our selection of the input sizes. For example, in stencil codes each element represents a point in the physical domain and the iteration's computation is linear in the input size. Consequently, large sizes are more common and practical. On the other hand, large input sizes are less common in molecular dynamics since the force computation in each iteration is not linear in the number of atoms and molecules. Table 4.2 presents the input size per processor of each application in our experiments. These sizes are small compared to weak scaling runs that fill the node's main memory, but they are used for typical strong scaling runs. In addition, input sizes larger than the LLC usually behave similarly because of common streaming patterns discussed in Subsection 4.2.2. We study the effect of input size more extensively in different experiments.

Table 4.2: Application domain sizes.

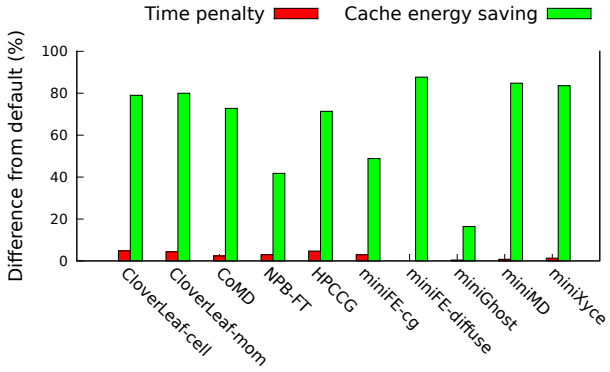
Mini-App	Input Domain Size per Processor
CloverLeaf	960×960 grid
CoMD	2744 boxes (including halo)
NPB-FT	$128 \times 128 \times 32$ grid
HPCCG	$60 \times 60 \times 60$ grid
miniFE	$50 \times 50 \times 50$ grid
miniGhost	$100 \times 100 \times 100$ grid
miniMD	6083 atoms (including halo)
miniXyce	602 variables

4.5.2 Results

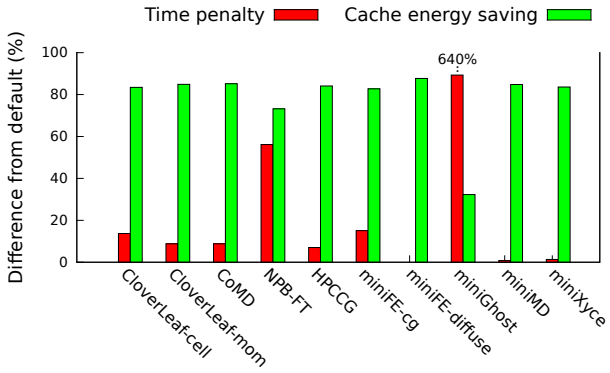
Table 4.3 presents the cache configurations that result in the best energy efficiency, with only slight execution time penalty (0.5% penalty threshold). As can be seen, in most cases, half of the first level instruction cache and three quarters of the first level data caches were



(a) 0.5% Threshold



(b) 5% Threshold



(c) No Threshold

Figure 4.6: Time penalty and cache energy saving of reconfiguration with different time penalty thresholds.

turned off for the best energy efficiency. The reason is that turning off ways of L1 caches can save a lot of energy, since they are the closest to the processor and have many more accesses. However, naive shutdown of ways of L1 caches can be detrimental, since they are critical for performance and increasing their miss rates can hurt performance significantly. In our simulation results (not presented here), some configurations with small L1 caches and not enough capacity in other caches resulted in more than one order of magnitude slow-down. Thus, the other levels need to have enough capacity to back up lower level caches, and configurations should be selected carefully.

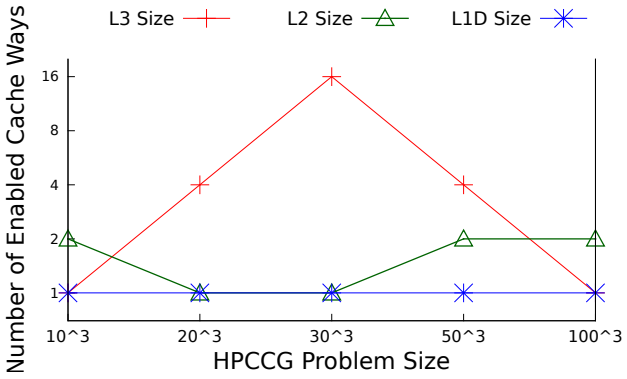
The only configuration with multiple L1D ways enabled is for miniMD. The reason is that the working set (data structures of atoms) fits in the L1 cache. Because of the high computation per data element in molecular dynamics programs (discussed in Section 4.2), the benefit of having them in L1 exceeds the power saving of turning off its ways.

Filtering Configurations We try all the configurations exhaustively since there are only a few SEBs but many processors in a supercomputer. For small scale (down to one processor) runs, one could try only the configurations that are more likely to achieve better performance and energy efficiency. Table 4.3 shows that the set of high performing configurations is not diverse and only a few configurations can be the best for different applications. More investigation at the small scale is left for future work.

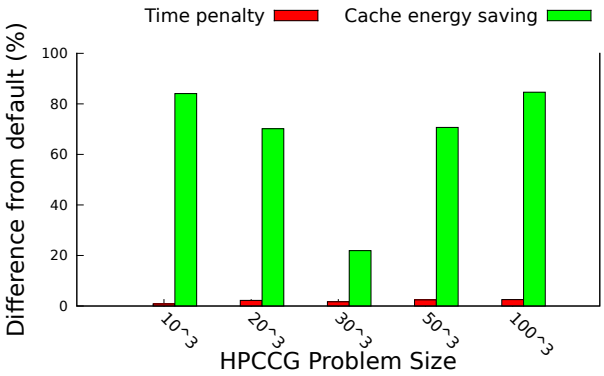
Table 4.3: Best configuration found with lowest energy but without performance penalty. Format: (number of cache ways on)/(total number of ways).

Mini-App	L1D	L1I	L2	L3
CloverLeaf-cell	1/4	1/2	2/8	16/16
CloverLeaf-mom	1/4	1/2	2/8	16/16
CoMD	1/4	1/2	2/8	8/16
NPB-FT	1/4	2/2	4/8	16/16
HPCCG	1/4	1/2	2/8	16/16
miniFE-cg	1/4	1/2	2/8	16/16
miniFE-diffuse	1/4	1/2	1/8	1/16
miniGhost	1/4	1/2	2/8	16/16
miniMD	2/4	1/2	2/8	1/16
miniXyce	1/4	1/2	4/8	1/16

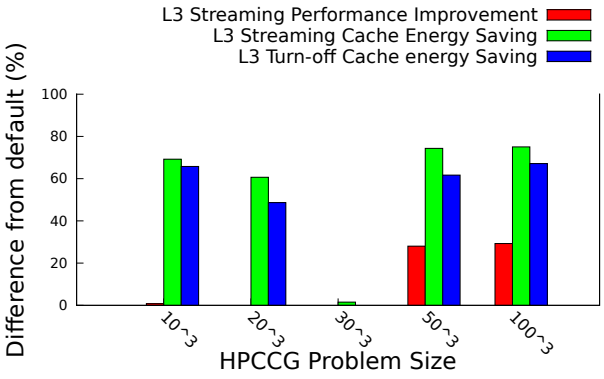
Figures 4.6(a) to 4.6(c) present the execution time penalty and energy savings of different mini-apps due to reconfiguration, with different performance penalty thresholds. Note that some mini-apps have more than one significant kernel (presented separately, such as miniFE-cg), while others are simple enough to take the whole iteration as reconfiguration units. From this figure, it is evident that with negligible change in execution time (less than 0.5% performance penalty threshold, 0.2% average actual penalty), very significant cache energy



(a) Number of Active Ways



(b) 5% Threshold Energy Saving



(c) L3 Streaming v.s. Simple Turn-off (0.5% Threshold)

Figure 4.7: Reconfiguration with different input sizes.

savings (up to 88%) are possible. On average, about 40% of cache energy consumption can be saved by just turning off ways of caches, without a significant performance penalty.

Furthermore, a small sacrifice in performance (less than 5% threshold, 2.4% average actual penalty) can result in more cache energy savings (about 67% on average). These small performance differences in the computation may not result in any performance degradation for many HPC applications because of inter-node communication. Moreover, minimizing cache energy without considering performance degradation results in more savings (about 78% average savings), but it can result in a very high penalty in some cases (6.4 times slowdown for miniGhost). This happens for miniGhost because its data fits in the L3 cache, but this method is trying to turn L3 ways off to save leakage energy. This is clearly a suboptimal decision from the energy standpoint as well, because other energy consumption sources, such as extra memory transfers, have not been considered. One should consider other energy sources if available for measurement consequently or cap the performance penalty.

Figures 4.7(a) to 4.7(c) illustrate the behavior and effectiveness of our approach for different problem sizes. Figure 4.7(a) illustrates that our approach initially increases the cache size (mostly L3) to incorporate the working set, which is the most energy efficient decision. However, a larger cache is not very useful for very large working set sizes and decreasing the size adaptively is the best strategy. Figure 4.7(b) is consistent with the previous one, demonstrating that when the working set fits in the cache, less energy savings are possible (since that energy is consumed in a useful manner, following our discussion in Section 4.2).

Figure 4.7(a) also demonstrates that our algorithm sometimes prefers to have more than one way of the L2 cache active, which is consistent with the results of Table 4.3 but seems counter-intuitive in some cases. Our insight is that, especially when there are fewer L3 ways on, at least two L2 ways are needed to reduce the conflict misses in both L2 and L3. These complicated scenarios are difficult to handle by methods that do not test different configurations and only rely on system metrics.

4.6 Reconfigurable Streaming

Based on the memory patterns of HPC applications, it is beneficial to use a streaming strategy for two of the three application classes we identified (Section 4.2). Following the discussion of Section 4.3.4, we propose an RTS-controlled reconfigurable streaming strategy. However, in our proposal, the cache organization is not changed and the system prefetches to the L3 cache instead of a specialized streaming buffer. When RTS switches to the streaming strategy, a streamer starts prefetching to the L3 cache. The streamer is a small structure

which issues extra memory requests (similar to the CPU requests). Therefore, switching only involves turning the streamer on and off, which takes only a few cycles. The implementation details of the streamer hardware is similar to previous work [112]. Note that the streamer accesses are treated in the same fashion as the processor accesses (same cache line size, etc.). In this section, we use the SESC cycle-accurate simulator to evaluate our streaming approach.

There are two important parameters of the streaming strategy that need to be tuned based on the application and its input size. First, the size (number of ways) of the L3 to be used for streaming (as the streaming buffer) needs to be decided. A small streaming buffer can potentially harm performance because useful data might be evicted prematurely, increasing the cache miss rate. On the other hand, a larger than necessary streaming buffer will waste energy. Second, the best streaming depth needs to be determined carefully. Prefetching should bring enough data to the cache to hide memory latency, but too much extra data can evict useful data and waste memory bandwidth and energy. The RTS can tune these parameters dynamically. In general, choosing the streamer configuration is done in the same manner as choosing the cache configuration, and most of our previous discussions are directly applicable here as well.

The hardware implementation of software-controlled reconfigurable streaming is simple. The hardware for prefetching usually includes an adder that generates the next address to be prefetched from the previous address. The input of that adder can be exposed to software as a system register. Our approach does not add repetitive prefetch instructions (as in compiler prefetching approaches), so it avoids significant overhead.

Continuing with our HPCCG example, Figure 4.7(c) presents the results when the runtime only tunes the LLC cache size for streaming. The prefetch depth is fixed at four cache lines. The results demonstrate that streaming can improve performance significantly for larger input sizes of HPCCG, while saving more energy than basic reconfiguration of the L3 cache. For the 100^3 grid size, performance is improved by 30%, while saving 75% of cache energy consumption (relative to the default configuration).

Tuning the prefetch depth seems is more challenging, and the RTS is the best agent for this task. Figure 4.8 presents the runtime of HPCCG with different prefetch depths and cache sizes. In addition, various statistics of the system for these configurations are presented in Figure 4.9. As can be seen, the performance is better with more cache ways enabled, but the extra energy consumption might not be worth the slight performance increase in some cases. For example, having all 16 ways on improves performance only slightly compared to using 8 ways, but the energy cost is considerably higher as revealed in Figure 4.9(a). Using eight ways of the LLC with prefetch depth of 32 seems to be a good performance-oriented

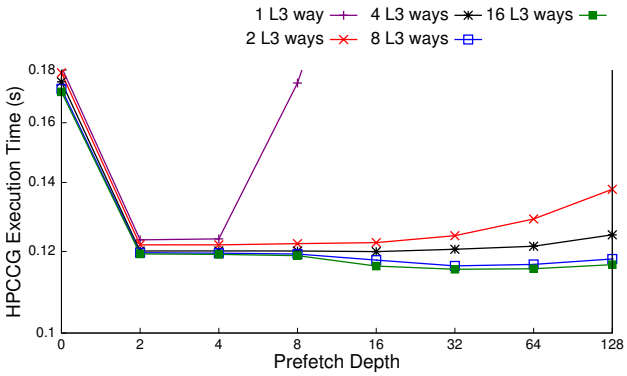


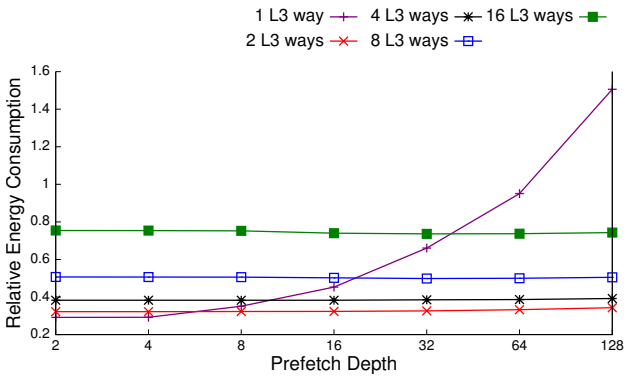
Figure 4.8: Performance of different streaming configurations for HPCCG (input size 50^3).

tradeoff, which improves performance by 32%, while saving 50% of the cache energy. If energy is the main factor, using only one way with prefetch depth of two can save 71% of cache energy, while improving performance by 28%.

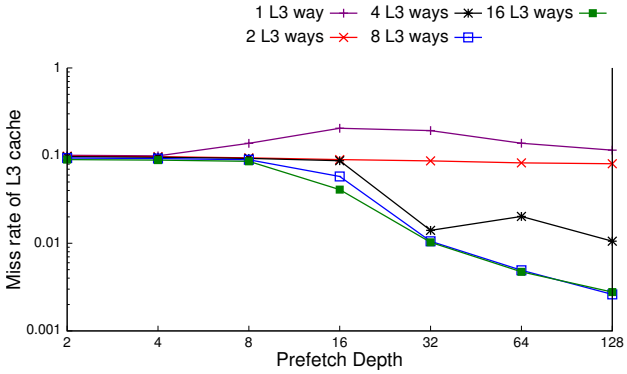
Analyzing the performance behavior of streaming strategies is complex in modern processors because of intricate cache hierarchy interactions and out-of-order/speculative execution. Figure 4.9(b) illustrates that in many cases, LLC miss rate decreases with very deep prefetching, but Figure 4.8 indicates that the performance becomes worse. More analysis reveals the reason: deeper prefetching reduces the memory delay for the mispredicted speculative paths, causing it to interfere more with the correct execution path. Figure 4.9(c) presents evidence for this conclusion: the number of instructions issued for the exact same computation increases with deeper prefetching. This means that the mispredicted speculative paths are making more progress and issuing more instructions, while the useful instructions committed are the same. Their excessive memory accesses evict useful data from various cache levels, harming application performance. This example demonstrates that tuning these parameters based on simple system metrics such as cache miss rate will not necessarily improve performance, and higher level software control in the RTS is needed.

4.7 Related Work

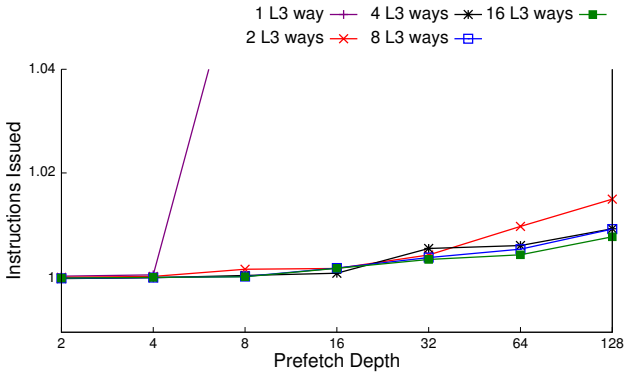
The Exascale Computing Study report [2] presents energy consumption as the main challenge for future systems, with data transfer within the memory hierarchy being a large component. Other previous studies have also characterized scientific applications [89–91], mostly establishing that scientific applications can be incompatible with common memory hierarchies. Cicotti et al. [123] evaluate the potential of cache reconfiguration for HPC ap-



(a) Energy



(b) L3 Miss Rate



(c) Instructions Issued

Figure 4.9: Statistics of different streaming configurations for HPCCG (input size 50^3).

plications (without proposing a practical solution), and suggest that significant savings are possible. Our results get close to their predictions. Thus, cache hierarchy reconfiguration is promising.

Automatic cache hierarchy reconfiguration in hardware has been explored extensively [92–97]. A survey by Zang and Gordon-Ross [98] summarizes the literature on cache adaptation. However, it is hard to predict application's phase changes and behavior in hardware and the hardware's "window" might be too small to capture the whole iteration. Also, predicting the best configuration in hardware is difficult, and incorrect hardware reconfiguration might result in extreme application performance slow-down. Moreover, automatic cache reconfiguration in hardware makes the hardware even more complicated and might also increase the energy consumption (due to additional structures, tables, etc.).

Compiler directed cache reconfiguration has also been explored [100, 101]. However, the compiler's analysis is usually limited because of the lack of runtime information. For example, array indices can be complicated in HPC application, inhibiting the required analysis. Thus, the RTS is the best agent to drive the cache reconfiguration. To the best of our knowledge, our work is the first to introduce a RTS-based adaptive cache reconfiguration in the of context HPC systems.

4.8 Conclusion

Caches consume a large fraction of a processor's power, but a fixed cache configuration does not fit every application. We exploit the regular structure of HPC applications and the partitioned structure of caches to reconfigure the caches (turn on/off ways of the cache) in the RTS, and save a large fraction of cache energy. The RTS is the best agent to direct the reconfiguration, since it can recognize the application's pattern easily (as we showed using formal language theory), without programming effort or hardware implementation overheads. Using the SESC cycle-level simulator, we demonstrated that 67% of the cache energy is saved on average, while incurring only a 2.4% penalty in sequential computation. Assuming that 70% of the total power of an HPC system is consumed by its processors, and that 40% of each processor's power goes to its caches, 19% of the total power is saved using our approach. This power can be used to turn on more compute nodes and further improve performance for over-provisioned systems. Moreover, we established that the change of cache strategy to reconfigurable streaming can save up to 75% of the cache energy and also improve performance by 30% in some cases.

CHAPTER 5

Power Management of Extreme-scale Networks with On/Off Links in HPC Runtime Systems

Large-scale parallel computers are becoming much larger in terms of the number of processors, and larger interconnection networks are being designed and deployed for those machines. The reason is that the demand for performance of supercomputers is escalating, while single-thread performance improvement has been very limited in the past several years. Moreover, the many-core era with on-chip networks is rapidly approaching, which will add another level to the interconnection network of the system [124]. These immense networks are a key factor in the performance and power consumption of the system.

Modern networks are over-provisioned in resources (e.g. links), in order to provide good performance for a range of applications. Since networks with lower latency and higher bandwidth, in comparison to existing popular networks (such as 3D Torus networks), are necessary for some applications executing on multi-petaflop/s systems, higher radix network topologies such as multi-level directly connected ones [125–127] and high-dimensional tori [128] are being proposed and used. Although these networks are designed to provide enough bisection bandwidth for the worst case (e.g. all-to-all communication in FFT), not all applications make use of the abundant bandwidth. Furthermore, the intention is to provide low latency for all applications, hence the network provides small hop count and low diameter for any given pair of nodes. However, the set of communicating node pairs of different applications vary, which may leave some part of the network unused in each application. As evidenced in this chapter (Section 5.2), the net result is that many applications do not use a large fraction of links, especially for high radix networks.

Saving network power is crucial for keeping HPC systems within a reasonable power budget. Power and energy consumption are major constraints for HPC systems and facili-

ties [129], especially at the high end. Interconnection networks are often among the major power consumers for different systems, and many researchers have reported on their power consumption. For example, routers and links are expected to consume about 40% of some server blades' power, which is the same as their processors' power budget [130, 131]. For current HPC systems, using an accurate measurement framework, Laros et.al. [132] report more than 25% total energy savings by shutting off some of the network resources of a Cray XT system. In future systems, especially because of the increasing number of cores per chip, and aggressive network designs, the network is expected to consume 30% of the system's total power [133]. From this power consumption, up to 65% is allocated to the links and the resources associated with them [131] (and the remaining 35% is mostly consumed by routers). In addition, up to 40% of the many-core processor's power budget is expected to go to its on-chip network [131].

In contrast to processors, the network's power consumption does not currently depend on its utilization [131], and it is near the peak whenever the system is "on". For this reason, while about 15% of the power and energy is allocated to the network in many current systems [134], it can go as high as 50% [135] when the processors are not highly utilized in data centers. While, for *HPC* data centers the processors are not usually as underutilized, they are not fully utilized all the time either and energy proportionality of the network is still a problem. Therefore, it is essential to save the network's power and make it energy proportional [135], i.e. the power and energy consumed should be proportionate to the usage of the network.

An effective approach to address this problem and improve energy proportionality is to turn off unused links. Thus, we propose addition of hardware support for on/off control of links (links that can be turned on and off), which can be used by the runtime system to save the wasted power and energy consumption. We show how the runtime can accomplish that by observing the applications' behavior. Note that adaptive runtimes have also been shown to be effective for load balancing and power management (using DVFS) [12]; our approach makes use of the same infrastructure. We also discuss why the hardware and compiler cannot perform this task effectively, and why network power management should be done by the runtime system.

Our contributions can be summarized as follows:

- We have evaluated the communication patterns of different HPC applications' and benchmarks' with respect to extreme-scale high-radix networks. The applications and benchmarks we evaluated include NAMD [136], MILC [137], ISAM [138], Stencil benchmarks (representing nearest neighbor communication patterns) and some of NAS Par-

allel Benchmarks [20].

- We have proposed a runtime system based approach to adaptively turn off unused links, which has various advantages over the previously proposed hardware and compiler based approaches.

- We have developed a theoretical model of link utilization of HPC applications, which provides insights about the applications and networks.

- We present a case study demonstrating that system design alternatives (e.g. mappings) with similar performance can have very different power consumption profiles.

Using our basic approach, for commonly used nearest neighbor applications such as MILC [137], 81.5% of the links can be turned off for a multilevel directly-connected network (around 16% of total machine power, assuming 30% network power budget), and 20% for 6D Torus (Sections 5.2 and 5.3). Moreover, we demonstrate that approximately 20% of the machine power can potentially be saved for most applications on these networks (Section 5.5) using a smarter scheduling approach. All these can be realized if the system allows the runtime system to turn off some of the links.

Sections of this chapter are organized as follows. Section 5.1 establish the background by discussing the related work, extreme-scale networks and applications' communication patterns. Section 5.2 demonstrates, via empirical evidence, that many links are never used on different high-radix networks and proposes a basic approach to turn them off in the runtime system. Section 5.2.2 of this section presents a case study, which shows how much our basic approach can save for two different design alternatives (which have the same performance). Section 5.3 discusses the implementation of our approach in a runtime system, and methods to handle practical issues that arise. Section 5.4 develops a theoretical model to estimate the power and energy that can be saved for an application, running on a high-radix network. The insight from this model helps us generalize the idea into a more practical scheduling approach in Section 5.5, which also considers the on/off transition delay. We conclude the chapter in Section 5.6.

5.1 Background and Motivation

5.1.1 Related Work

Power consumption of interconnection networks in supercomputers, distributed systems and data centers has received special attention in recent times. Several techniques have been proposed for reduction of network power in non-HPC data centers [134, 135, 139]. Intelligent power-aware job allocation and traffic management schemes form the basis of many of these approaches. Laros et.al. [132] present results on potential power saving using CPU and network scaling, by post processing the data collected from the monitoring system of Cray XT machines. Their work, using real systems (instead of simulations and projections) and real applications, shows the importance and potential of network power management for supercomputers.

Among hardware based approaches, power management of interconnection networks using on/off links has been studied [131, 140, 141]. On/off links, which refers to shutting down communication links that are not being used, has shown to be a useful method to save power. However, dependence on hardware for power management may cause considerable delay for some applications. Additionally, hardware does not have enough global information about the application to manage network power effectively.

Soteriou et.al. [142] show severe possible performance penalty of hardware approaches, and propose the use of parallelizing compilers for power management of the links. However, parallelizing compilers are not widely used because of their limited effectiveness, and most parallel applications are created using explicit parallel programming models [143]. Furthermore, compilers do not have information about input dependent message flow of an application, and cannot manage the power effectively for such applications.

New programming paradigms to perform power management inside the application (by giving more information about the communication) has also been proposed [144]. Although the programmer has more information about the application, involvement of the programmer compromises productivity. In addition, such approaches cannot be applied to legacy code easily. Thus, automatic approaches seem more practical.

As an alternative to hardware, compiler and application driven power management, we advocate network power management by the runtime system. Limited network power management by the runtime system, such as for collective algorithms, has been proposed in the past. Power management using on/off links in the runtime system has also been studied [145]. However, that approach is limited to management of network links only during collective operations in MPI. In this chapter, we propose the use of an adaptive runtime

system to manage the power of network links using on/off control, taking into account all of the communications performed by an application.

5.1.2 Network Power Management Support on Current Machines

Unfortunately, network power management support on current HPC machines is very limited. For example, it is possible to reduce link and node injection bandwidth on a Cray XT system (effectively turning off some portion of the links), but it requires a reboot of the whole machine [132]. Thus, using this feature is impractical. Other recent machines such as IBM Blue Gene/Q, Cray XE6, and Cray XC30 do not seem to have any feature for dynamic network power management either. However, techniques such as on/off links have been implemented before, and it seems feasible to include them for HPC machines as well. For instance, some commercial systems [1] can disable some of the board-to-board and box-to-box links to save power. Currently it takes 10,000 cycles to turn the links on/off, although even this can be improved much further [131].

5.1.3 Extreme-scale Networks

In this section, we briefly describe n-dimensional tori and multilevel directly-connected networks, which have been used in recently developed supercomputers that are predominant in the Top-500 list [146].

n-dimensional tori have been used in many supercomputers such as IBM Blue Gene series, Cray XT/XE, and the K computer. Given an n-dimensional mesh, a torus is obtained by adding wrap around links in every dimension, i.e., by adding links that connect nodes at one end of a dimension to the nodes at the other end. Tori are symmetric in the sense that the number of links out of every node is the same, with each node being connected to two other nodes in every dimension. An n-dimensional torus strikes a good balance in terms of bisection bandwidth, latency, and the link cost, and have been shown to be scalable. In the past few years, most vendors have increased the torus dimensionality from three (as it is in IBM BlueGene/P and Cray XT/XE) to five (IBM BG/Q) and six (the K computer). This shift is necessary in order to keep latency low, with possible increase in the bisection bandwidth. We present analysis and results for link utilization of an n-dimensional torus, with n varying from 3 to 10.

[1] Motorola MC92610 WarpLink 2.5 Gb/s Quad SERDES Transceiver, Motorola Inc., www.motorola.com

Multilevel directly-connected networks have been proposed by IBM (PERCS network [125]), the DARPA sponsored Exascale study report [129] (Dragonfly topology [147]), and Cray (Aries network [127]). In all of these proposals, similar multi-level directly connected networks have been described. In these networks, nodes are logically grouped together at multiple (currently two or three) levels. In each level, nodes (or the grouped entities from previous level) are connected in an all-to-all manner. Hence, in the first level a clique of nodes is formed, and in the second level, a clique of cliques (from the first level) is constructed and so on. The resultant network, with its large number of links, boasts of a large bisection width. At the same time, the latency of the entire system is low (few-hop connectivity between any pair of nodes). Currently, these networks are used in some large-scale IBM Power 775 machines [2] and in the Cray XC30 machines. In this study, we use the parameters of PERCS as an instance of multilevel directly-connected networks, since they are readily available. However, the conclusions will apply to other multilevel directly-connected networks as well.

In Figure 5.1, we present a prototype of the PERCS network (two-level directly connected), in which the nodes are grouped, and connected in an all-to-all manner to form supernodes. These supernodes are further connected in an all-to-all fashion to obtain the entire system. We present link utilization results for these networks as well.

We observe that the two topologies we present results on, multilevel directly-connected networks and tori with high dimensionality, have a large number of links. The presence of these links provides an opportunity for high performance, as well as a challenge for power and energy proportionality.

5.1.4 Application Communication Patterns

Interconnection networks are designed to serve a range of communication patterns, in terms of bandwidth and latency. High radix networks, such as multilevel directly connected ones, provide a large number of links to support demanding communication patterns such as all-to-all and varying demands of different applications. In addition, in order to maintain low latency and fewer hops between every node pair, a large number of links are required. However, each application has its own communication pattern, so many node pairs of a system may not communicate during execution of a common application, leaving a large fraction of the network unused.

Figure 5.2 shows the communication pattern of some of the applications we use in this

[2] www.top500.org

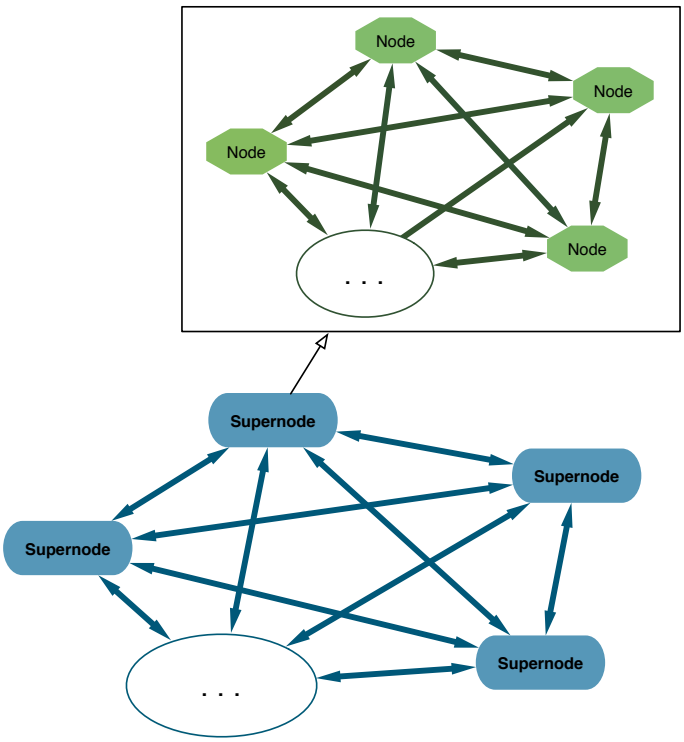
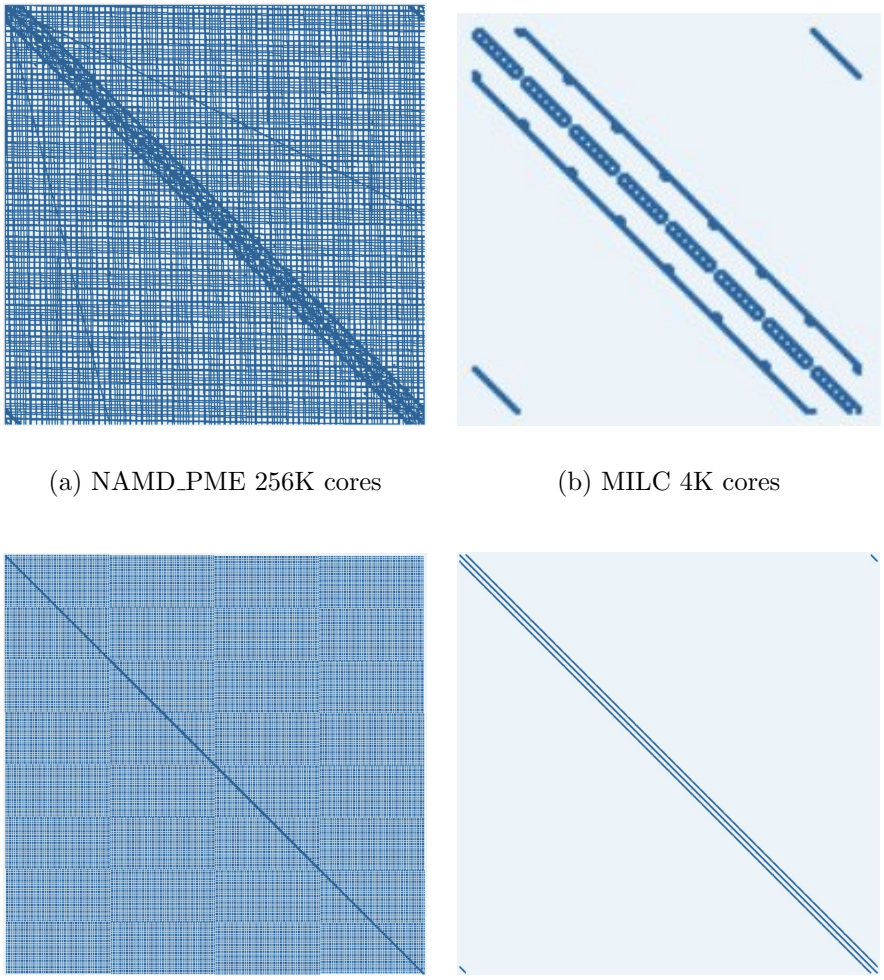


Figure 5.1: IBM PERCS - a two-level directly-connected network.

chapter. NAMD [136], implemented in Charm++ [16], is a prevalent parallel molecular dynamics code designed for high-performance simulation of large bio-molecular systems. Its localized communication pattern represents the pattern of many common particle interaction HPC applications. MIMD Lattice Computation (MILC) [137] is widely used to study quantum chromodynamics (QCD). Similar to many HPC applications, it has a near-neighbor communication pattern. We use CG from NAS Parallel Benchmarks (NPB) [20] to represent expensive many-to-many communication pattern found in some applications and MG from NPB to represent the communication pattern of common numerical solvers.

Both the vertical and horizontal axes of Figure 5.2 represent the nodes in a system. A point (x, y) is marked if the node y on the vertical axis sends a message to the node x on the horizontal axis, during the execution of an application. Each marked point has been enlarged for better illustration. It can be seen that many of the node pairs never communicate during execution of various applications. Moreover, the number of pairs that communicate varies significantly with the application. For instance, in NAMD_PME and CG, the number of node pairs that communicate is much larger than in MILC and MG.

Most of the communicating pairs in NAMD_PME are due to the FFT performed in the PME phase, which is done once every four iterations. Without the PME option, NAMD



(a) NAMD_PME 256K cores

(b) MILC 4K cores

(c) CG 64K cores

(d) MG 64K cores

Figure 5.2: Communication patterns of different applications.

has a near neighbor communication pattern, which can be seen in the dense region around the diagonal of Figure 5.2(a). CG, on the other hand, has a more uniform and dense communication pattern. Applications like NAMD_PME and CG, that have large number of communicating pairs are more likely to use most of the network.

On the other hand, the number of communicating pairs in MILC (Figure 5.2(b)) and MG (Figure 5.2(d)) are few, and concentrated near the diagonal. As such, these applications are expected to make use of a small fraction of the available network links. These applications represent a large class of applications in science and engineering, such as the ones following the nearest neighbor pattern [20].

All illustrated cases have a dense region close to the diagonal of their communication

graph, suggesting that nearest neighbor communication constitutes a major part of many applications' communication. This can be used as a clue in understanding a network link's usage. We use Stencil, decomposed in two, three and four dimensions, to study network's link usage for near neighbor communication patterns. From this discussion, there are reasons to expect that there is an extensive opportunity to save the power of the network links in higher-radix topologies in many common cases, since they are designed for the worst cases with many communicating pairs (such as random access benchmark or FFTs).

In summary, there are applications that have intense communication patterns such as all-to-all, but many applications have only nearest neighbor pattern. Additionally, *embarrassingly* parallel applications that essentially do not rely on the network during their computation (e.g. ISAM [138]) represent an extreme set. Since they do not use any of the links, the link power can be saved easily.

5.2 Potentials of Basic Network Power Management

In the previous section, we observed that the number of communicating pairs for many applications is not large, which indicates that a sizeable fraction of links may be unused. In this section, we discuss and evaluate a basic power-saving approach for links, implemented in an adaptive runtime system. In this approach, the runtime monitors a few iterations and turns off the links that are never used during execution.

Our methodology of evaluation is to emulate an application at scale using BigSim (which has been validated for these networks before [148, 149]) and capture the communication traces. These traces are then used to simulate the target network and mark the links that are used.

We assume default mapping for placing processes (ranks) onto processors for all the networks. For a 32 cores per node case, it means that the first 32 ranks are mapped to the first node, next 32 ranks are mapped to the second node and so on. Only direct routing is considered for multilevel directly-connected networks in this section (before Section 5.2.2), which means that the messages are sent directly to the receiver, instead of going through an intermediate supernode (i.e. indirect routing). Effect of different mappings and indirect routing are discussed and evaluated in Section 5.2.2. For tori, we assume minimal dimension order routing, which is used in many of the current supercomputers. The results are easily extensible for adaptive routing as we discuss later.

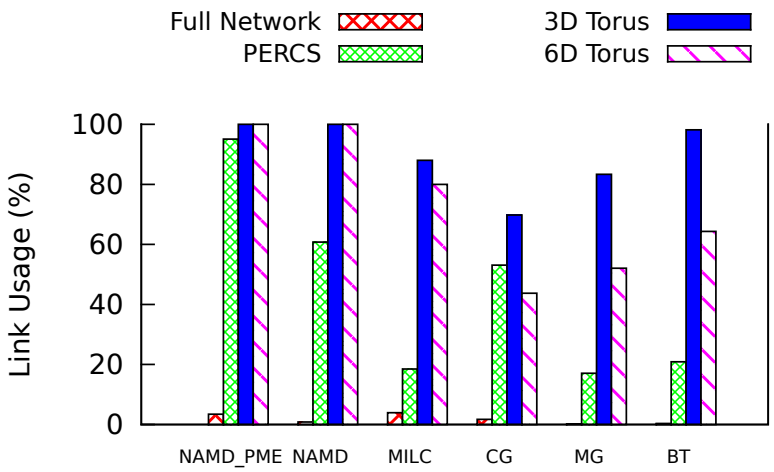


Figure 5.3: Fraction of links used during execution of various applications.

5.2.1 Link Usage of Modern HPC Networks

Figure 5.3 and Figure 5.4 show link usage of different applications and benchmarks for a fully connected network (a link between every pair of nodes), 3D Torus, 6D Torus and PERCS (two-level fully-connected). In the context of this section, we consider a link as "used" if it is used *at any time during* execution of an application. With this assumption, the specifics (e.g. link bandwidth) of each network other than its topology do not make any difference. These used links may be utilized for only a small fraction of the application execution; we will exploit this property in Sections 5.4 and 5.5. Note that the full network is an asymptotic case that is not usually reached by the large-scale networks. However, for example, small jobs (less than 1k cores for PERCS) running on a two-tier system will have a fully connected network. In this case, most of the links can be shut down according to our results, which saves a significant fraction of the system's power.

As shown in Figure 5.3, link usage of each application is different, and depends on the topology of the system. For example, MILC only uses 3.93% of the links of a fully connected network, while it uses 80% of the 6D Torus links. For most applications, a larger fraction of 6D Torus links will be used in comparison to links on PERCS network; an exception is CG that uses a higher fraction of PERCS links. This shows that analysis of the link utilization of different networks is not trivial and depends on various aspects of the topology and the application.

For the applications of Figure 5.3, from 4.4% to 82.94% of the links are never used during the program's execution on PERCS. In the stencil benchmarks of Figure 5.4, 2D-Stencil uses only 11.91% of the PERCS links, with similar numbers for other stencil dimensions. This demonstrates a great opportunity for the runtime system to save link power of two-level

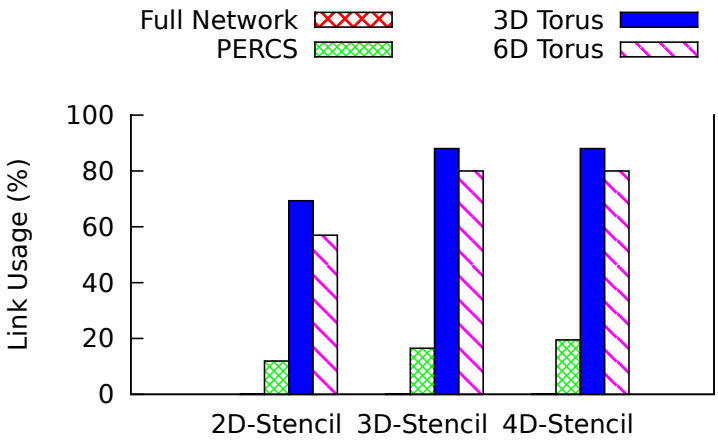


Figure 5.4: Fraction of links used during execution of stencil codes.

directly connected networks.

There is an opportunity on 6D Torus to save energy as well. Even though NAMD uses all of the links, MG leaves 47.92% of the links unused. However, many applications can use most of the links of a 3D torus, which has been one of the dominant topologies in the past and in the current supercomputers. There is potential for saving power in some cases (e.g. 30.67% for 2D-Stencil), but the savings are neither high nor common. This happens even with deterministic routing, which uses fewer links than adaptive routing. This shows that implementing on/off links for those networks is not significantly useful, and probably is the reason that they have not been implemented so far. However, for high dimensional tori and multi-level directly connected networks, the benefits justify the implementation cost of software controlled on/off links. If we take MILC to represent a significant set of common HPC applications (which usually have near neighbor communication), 81.51% of PERCS links and 20% of 6D Torus links can be turned off to save power. Assuming that 65% of network power goes to links and the network consumes 30% of the total machine's power, around 16% of total machines power can be saved for PERCS systems and around 4% can be saved for 6D Torus systems.

In NAMD_PME, the communication intensive PME calculation is usually performed every four iterations (which takes around 16 ms assuming about 4 ms per iteration for ApoA1 benchmark on 2K cores of BGQ [150]). In this case, many links can be turned off after PME communication is complete, and turned back on right before the next PME communication phase begins (scheduling on/offs is further discussed in Sections 5.4 and 5.5).

We observe that even though 3D-Stencil has a 3D communication pattern, when it is mapped to a system with 32 cores per node, the communication between nodes is not an exact 3D pattern anymore. Thus, some fraction of the links (12%) are not used.

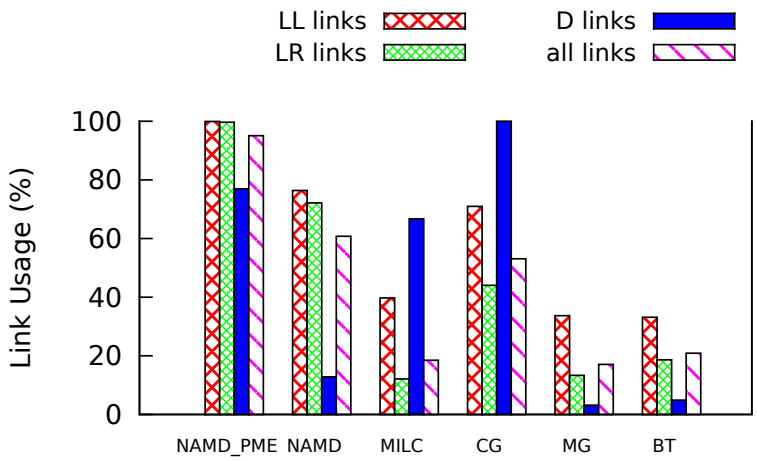


Figure 5.5: Fraction of different links used during execution on PERCS.

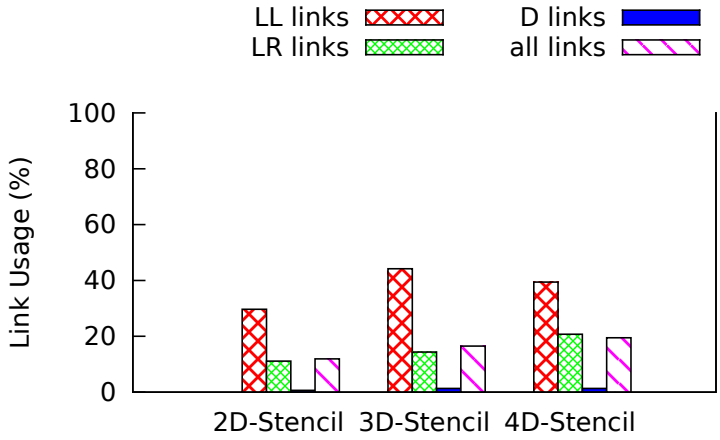


Figure 5.6: Fraction of different links used during execution of stencils on PERCS.

So far, for simplicity, we assumed that all the links of the network are the same and have the same power cost. However, networks are usually made of different links for practical purposes. These links even have different technologies (optical vs. electrical). For example, in PERCS network, global links are called D-links and connect clusters of nodes at second level and they use optical technology [125]. These long links are probably more power hungry than the local ones. On the other hand, LLocal (LL) and LRemote (LR) links connect nodes placed close to each other (LL for nodes in the same drawer, LR for other local links) and use electrical technology. These local links probably consume much less power compared to the global ones.

To find out which type of links are used more often, Figure 5.5 and Figure 5.6 show the usage of different types of PERCS links. Overall, D-links are usually less utilized than the

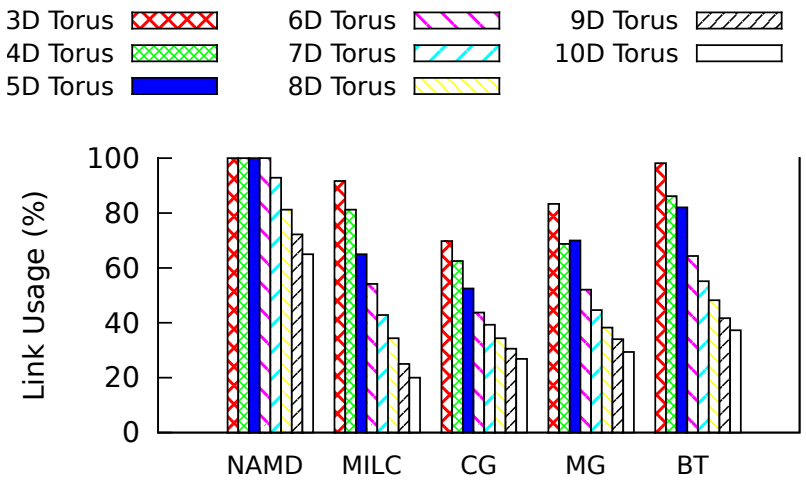


Figure 5.7: Fraction of links used during execution on tori.

local ones. This happens because most of the communication of the applications is either local or near neighbor exchange. Even NAMD_PME, which exhibits limited opportunity for power saving in previous results, does not use 23.09% of D-links; this may improve absolute power saving. MILC shows high usage of D-links because the results are for 4K processors only (4 supernodes), hence there are just 12 D-links. For larger configurations, it should show link usage similar to 4D-Stencil and have a very low D-link utilization. CG is again an exception and uses more of the D-links. This is because its communication is not local but distributed as mentioned earlier. The stencil benchmarks use only around 1% of D-links and most of those links can thus be turned off safely. Thus, using a simple model of same power cost for all links is pessimistic, and the actual savings can be much higher in many cases as the power hungry D-links have less utilization. Note again that these results are with the default (rank-ordered) mapping (we will discuss a case of other mappings in Section 5.2.2).

Figure 5.7 and Figure 5.8 show the link usage of the applications on tori with different dimensions, from 3 to 10. As dimensionality of tori is increased, a smaller fraction of the links are used, which is intuitively expected. For example, 4D-Stencil uses only 53% of the links of a 10D torus network, but more than 80% links are used on a 3D torus. Even NAMD that does not have any savings on low dimensional torus, shows potential for saving power on a torus with sufficiently high dimensionality, starting from 7D. It uses only 65% of the links of a 10D torus, which shows that even such applications have potential of link power savings on high dimensional tori.

Other than these applications, there are cases where the network is virtually unused. Data parallel applications do not have much communication (except during startup, I/O in the beginning and at the end) and do not use the network during execution. For example,

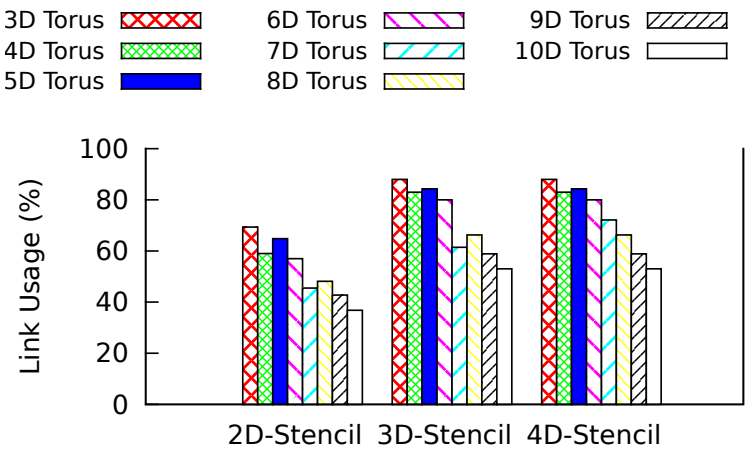


Figure 5.8: Fraction of links used during execution of stencils on tori.

ISAM [138], which is a climate modeling application, only uses stored climate data to do the computation (in its standalone mode). Thus, almost all of the network power can be saved for these applications during the main computation phases (I/O uses the main network in some machines, but it happens usually infrequently).

To summarize, in the common near neighbor applications like MILC, up to 16% of total machines power can be saved (assuming 30% network power budget and 65% of network power associated with links) using a basic power management approach. Since our assumptions are very conservative and only links that are never used (and are not likely to be used) are turned off, the application will not experience a significant performance penalty.

5.2.2 Different Mappings

In the results so far, we assumed default mapping with direct routing for PERCS network, which implies sending each message directly to the destination supernode. However, it had been shown that this configuration might result in contention in few links of the network for some applications [126]. In this case, one might use intelligent application specific mappings, or use other more general alternatives that had been proposed before. A previous study on PERCS network [126] suggests using random mapping or indirect routing to avoid contention on few links and improve performance. Indirect routing uses a random intermediary supernode for each message transfer (dynamically), while random mapping places the processes (e.g. MPI ranks) on random processors (statically). The purpose is to use more links to avoid contention, at the cost of some possible overheads (e.g. due to more hop count).

Figure 5.9 shows the link usage of these schemes (proposed in [126]) compared to the default mapping. Random mapping has higher link usage than default mapping, which is intuitive. It can use 33.18% of the links, which is twice the 16.51% link utilization of the default mapping. However, the overall usage is still low and the possible savings are as high as 67%. Note that this scheme uses many more D-links, which may increase the power consumption significantly. Thus, when choosing among different mappings for future machines, power consumption should also be taken into account, since in addition to performance, mapping can affect power consumption as well.

Indirect routing uses all of the links of the network, since every packet is routed through a random supernode. Therefore, it is very expensive in terms of network power and no energy reduction is possible. On the other hand, random mapping is shown to have similar performance as indirect routing on PERCS networks [126]. Thus, indirect routing should be avoided and random mapping should be used to have much less power consumption but the same performance. The routes in random mapping are statically determined during the mapping phase, while indirect routing dynamically changes the routes for every packet. This suggests that different aspects of hardware and software design can affect power consumption of the network significantly, and power should be considered at every stage of the design.

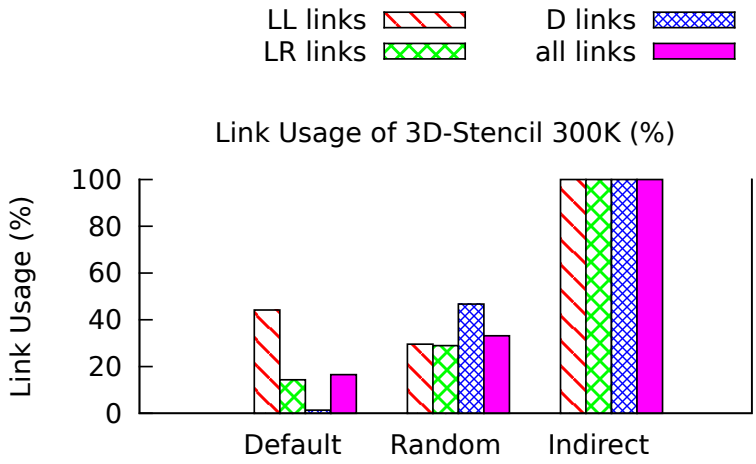


Figure 5.9: Fraction of links used with different mappings.

5.3 Implementation in Runtime System and Hardware

The large amount of unused links, discussed in the previous section, presents opportunities for power optimization and savings. Although past studies suggest hardware and compiler techniques, we believe that this should be done by the runtime system. Hardware and

compiler do not have enough information about the characteristics of the application, hence they may make conservative assumptions or cause unnecessary delays. For example, NAMD's communication depends on the input and previous iterations, and hence the compiler cannot assume any unused links. It is also difficult at the application level since it would hurt portability and programmer productivity.

On the other hand, runtime systems, such as MPI and Charm++, have enough information about both the application and the hardware to make wise decisions. The runtime system obtains this information about the application by monitoring the communication performed as the application executes (with negligible overhead [10]).

5.3.1 Runtime System Support

The runtime system mediates all the communications and computation, so it can instrument the application easily. Runtime systems, such as Charm++, use this information for many purposes such as load balancing [10] and power management [12]. They also obtain characteristics of the network, such as its topology [151]. Our approach requires only a small subset of this data to save network power: the communication graph of the application and the topology of the network. Using this information, our approach can turn off unnecessary links as follows. We assume that each node keeps track of the destinations of its messages. At the network power management stage, each node calculates the route for each of its destinations. It sends a notification message to each of the intermediate nodes to have them mark their used links. At the end, when a node has received all notification messages and marked its own links, it turns off all of its unused links. Note that collectives are mostly implemented as point-to-point messages in runtime, so they can be handled similarly. This algorithm, which is executed by every node, can be summarized as follows:

1. Obtain the destination list of local messages.
2. For each destination, calculate its route.
3. Mark the local links used by local messages.
4. For each intermediate node, instruct it to mark the required links.
5. Wait for all notifications to be received (possibly using a termination detection algorithm).
6. Turn off the local links that are never used either by local or non-local messages.

This algorithm needs to be invoked at appropriate times, which is feasible in most cases since scientific and many other parallel applications are usually iterative in nature. For the common case of static communication pattern, which encompasses all of our benchmarks except NAMD, every iteration follows a constant communication pattern. Thus, one invocation of the power management scheme (e.g. after the first iteration) is sufficient. Note that even in this simple case, the hardware cannot make wise decisions on its own, because it is not aware of the iteration time of the application and its window might be too small. In addition, hardware does not see the global picture of the application's message flow, since it usually works at the packet and flit levels.

For NAMD, the communication pattern between objects is static, but the objects may migrate between processors periodically, under the control of a load balancer. Therefore, the actual communication pattern varies. In this case, the new communication pattern can be determined by the runtime system at (or just after) the load balancing steps. Thus, the network power management algorithm needs to be called at every load balancing step. Even in this case, the switching delay of links is of negligible concern, since the runtime will not make any mistake in switching the links' states. In addition, since load balancing is not performed very frequently (usually once in thousands of iterations), our method will not add significant overhead. Many other dynamic (and phase based) applications, such as Adaptive Mesh Refinement (AMR) [152], can be handled in the same way.

5.3.2 Hardware Support

For our approach, we only require the network hardware to implement links that can be turned off and on (or any other power saving means such as DVS), along with a software interface to control them. Note that the "off" state should usually be implemented as a very low power (but slow) state. Turning the links completely off may increase the switching delay significantly because the links would need "re-training". Not turning the links fully off also ensures connectivity of the network.

In a simple but robust implementation, the runtime provides hints to the hardware to turn a link off, but the hardware turns it back on if a message (packet) needs the link. That message will pay the penalty of switching delay because of the incorrect prediction by the runtime. In this way, we do not strictly require any change in routing and switching tables. Note that the runtime can measure the iteration times and turn off the power management algorithm, or adjust it, if the performance is degraded. This feedback loop ensures "safety" of our approach for performance if anything about the application or system changes. This

safety cannot be provided easily by hardware or compiler approaches.

On some current machines (e.g. from Cray), network interference from other running jobs can decrease the predictability for the runtime and make the power management task more difficult. However, the other jobs running on the machine are most probably also iterative with predictive behavior, so the runtime can take them into account similarly. Note that job interference also has performance penalty [153, 154], and many machines (e.g. Cray systems) are exploring new job schedulers for isolated partitions. Some other machines, such as Blue Genes, already allocate isolated partitions (prisms) for every job running on the supercomputer. I/O interference can cause similar issues if I/O is performed on some I/O nodes that are out of the job's allocation, using the same network as the application. Thus, I/O needs to be considered as well.

Impact of Adaptive Routing (for Tori) For many networks (especially tori), dynamic adjustments, such as adaptive routing for performance and fault tolerance, have been proposed before. The dynamic behavior may hinder our approach because it reduces predictability, resulting in performance penalties. However, the support for adaptive routing is still limited in current machines due to practical restrictions. For example, the K computer has a fixed, minimal dimension order routing [128]. Some machines may support a limited form of adaptive routing, such as routing in "zones" on Blue Gene/Q [155]. For this case, the runtime needs to know the details of the routing protocol (what links are actually used for communication for messages of an iteration). This information is usually already available to the runtime system for communication performance optimization. Note that even for Blue Gene/Q, minimal dimension ordered routing is used for most messages depending on the system's configuration [156]. Adaptive routing is usually used for demanding cases such as all-to-all, in which case, we do not turn links off. Fault tolerance can also be considered easily, since most faults bring down a whole node, calling the runtime system's fault tolerance protocol. Thus, we call the network power management method after every resiliency action of the runtime. It is notable that such dynamic behaviors will not result in disastrous performance penalties, since the runtime can measure the performance and correct itself.

5.4 Power Model for Network Links

Our power management in runtime system approach is very simple but offers substantial link power savings. However, it is useful to know how much saving is possible for each application on a network. Theoretical models that suggest upper bounds of link power savings need to be

developed for this purpose. Furthermore, a simple theoretical model can give insight about the application's utilization of a network and compare different networks. In this section, with these goals in mind, we develop a theoretical model for link utilization of a network while running an application. Our model provides an upper bound on power requirement of an application using a particular network. We make some assumptions that keep the model tractable at the expense of some loss in accuracy. We also suggest some additions to the model to make it more accurate and provide tighter bounds.

Suppose it is possible to switch a link on and off without any delay, and a link has a bandwidth of B. Assume also that there is no zero-size message latency. In this ideal case, each link can be turned on whenever there is some message traffic and can be off otherwise. Thus, a link only consumes power when it has to transfer data. If $B_i = 100\text{MB/s}$ for link i, and a program transfers 100MB of data through this link during its 10 seconds of execution, then link i is used only for 1 second. Thus, only 0.1 of its capacity was utilized according to this simple calculation:

$$\frac{100\text{MB}}{(100\text{MB/s}) * 10\text{s}} = 0.1$$

Let us generalize this formula, assuming that each link i transfers z_i bytes of data during t seconds of program execution:

$$U_i = \frac{z_i}{(B_i t)}$$

In this formula, U_i represents the utilization of link i. We can derive the whole network's utilization by a sum over all the n links:

$$U = \frac{1}{n}\sum_{i=1}^{n} U_i = \frac{1}{n}\sum_{i=1}^{n} \frac{z_i}{(B_i t)} = \frac{1}{nt}\sum_{i=1}^{n} \frac{z_i}{B_i}$$

We also assume that $B_i = B$ are the same for all the links and derive the upper bound of power savings:

$$M = 1 - \frac{1}{(nBt)}\sum_{i=1}^{n} z_i$$

M is the fraction of network link's power that can be saved, given the hypothetical assumptions, e.g. no on/off delays. In this formula, n, B and t can be determined easily, but z_i depends on the application, mapping, network topology and routing algorithm. Note that we assumed on/off links without Dynamic Voltage Scaling (DVS) support. Using DVS for

the links carrying messages that are not on the critical path may result in even greater power savings.

Let us calculate this formula for a simple case of 3D Stencil, running on a 3D torus. Assume that the processes are mapped to processors perfectly, i.e. in a way that communicating processes are neighbors in the network. If each iteration takes 10ms, each message is 2MB and the bandwidth of each link is 1GB/s, we have:

$$M = 1 - \frac{1}{(n * (1000\text{MB/s}) * (10\text{ms})} \sum_{i=1}^{n} (2\text{MB})$$

Resolving the summation we have:

$$M = 1 - \frac{n * (2\text{MB})}{n * (1000\text{MB/s}) * (10\text{ms})} = 80\%$$

Thus, 80% of the links' power can be saved since only 20% of the time the links are being used. By using a perfect schedule for toggling the links, this power can be saved.

Calculating this formula is not usually as simple as this case, since z_i values are not easy to determine in many cases. Therefore, execution or emulation is required for finding the communication volume of each link, while running the application. However, this task is straightforward because the exact time that each link is used is not important for these values. First, communication traces can be obtained, even on a much smaller machine using an emulation approach, such as BigSim [148, 149]. Then, a simple counting program can determine the path for each message and keep track of each link's communication volume. Using this method, we have determined the maximum possible link power saving for each application.

Figure 5.10 shows the network utilization of different applications on 6D Torus and PERCS networks. Except CG on 6D Torus network, the applications utilize less than 10% of the network. Thus, according to our model, more than 90% of the link power can be saved. Our basic approach can realize a significant portion of these savings with low effort for many applications. For example, more than 81.5% of the power can be saved for MILC by the runtime system. Next, we incorporate transition delay in our model and extend our basic approach accordingly for more savings.

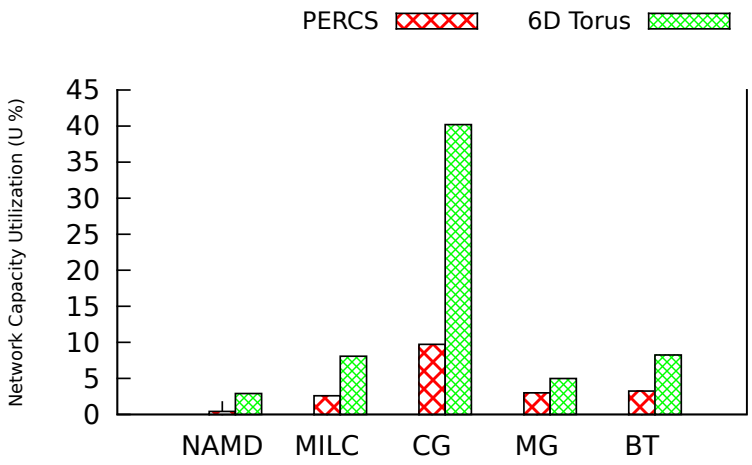


Figure 5.10: Network capacity utilization of different applications.

5.5 Effect of on/off Transition Delay

Our basic model assumes zero on/off transition delay for simplicity. We also do not consider the conservative delay of scheduling in the runtime system to account for system noise and other overheads. We can incorporate this delay if we add one more assumption: *each iteration of an iterative application is divided into long distinct computation and communication phases.* Here, *long* means that a computation stage is considerably longer than the link transition delay, so we can turn the link off in that duration. This assumption is usually valid for common HPC applications, such as the ones with nearest neighbor communication pattern and/or bulk-synchronous parallel (BSP) model.

In addition to this assumption, we also assume that the links consume their full power during their transition and extend our model. If each link i transfers z_i bytes for each iteration that takes t seconds and the transition delay is d, the link consumes full power U fraction of the time:

$$U_i = \frac{z_i}{(B_i t)} + \frac{2p_i d}{t}$$

Here, p_i is a boolean variable that is one if the link is ever used, i.e. $z_i > 0$, and zero otherwise. This variable is needed because if the link is not used at all, it does not need to make any transition. Furthermore, since the runtime system turns the link off after each communication step and turns it back on before the next one, we pay the delay cost twice per iteration. Note that transition delay d should be smaller than half of the iteration time of the application, since utilization cannot be more than one. We define p as the fraction of the network links used for the application and z as total communication volume over links.

Thus, we can derive the full network utilization:

$$U = \frac{1}{n}\sum_{i=1}^{n} U_i = \frac{1}{nBt}\sum_{i=1}^{n} z_i + \frac{2d}{nt}\sum_{i=1}^{n} p_i = \frac{z}{nBt} + \frac{2pd}{t}$$

This formula specifies the network utilization as a linear function of transition delay over the iteration time, so we can quantify the effect of the transition delay for each network. For practical cases, the transition delay is not a problem since the iteration time is much longer and the last term of the equation becomes small. For example, a typical short iteration time is around $10ms$, while some current implementations have a transition delay of around $10,000$ cycles ($10\mu s$ at $1GHz$). In this case, the transition overhead is just 1%.

Our approach may lead to overheads due to the software and hardware. In software, in order to capture the communication pattern and link utilization, the runtime system has to monitor the application. However, this should not have significant overheads since the monitoring is usually performed only once (because the applications are iterative) and its results are stored. Other overheads include the system call overheads (context switching overheads, argument verification overheads, etc.) because currently the runtime system is executed in the user space (which is likely to change in the future). Our experiments suggest that these overheads are limited to $20\mu s$ per call. In the hardware, as mentioned earlier, some current implementations have a transition delay of around $10,000$ cycles ($10\mu s$ at $1GHz$) for turning links on/off, and it is projected that it will improve much further (down to just 100 cycles) [131]. Hence, the overall overheads should be less than $30\mu s$ per call that turns a link on/off.

Figures 5.11 and 5.12 show the link power savings as a function of transition delay (other overheads are also included) for PERCS and 6D Torus networks (using simulation). For many of our applications, we have short iteration times of around $30ms$ and we show the results with up to $15ms$ delay for illustration. Thus, as can be seen from the figures, transition delays and other overheads are not significant problems for our approach.

Note that this approach assumes accurate scheduling of links' on/off transitions by the runtime system, which is achievable since each iteration's message send and receive times are usually very deterministic. To verify this, we ran some of the NPB benchmarks on Blue Gene/P and inspected a sample of processors. We found that the message sends and receives occur with regular intervals and are predictable. The prediction error was usually less than $200\mu s$, while the iteration time is in hundreds of milliseconds. Thus, in our results, we consider $1ms$ conservative delay for the runtime system to incorporate noise and variations in the system. Figure 5.13 summarizes machine power saving potentials of our approaches for different applications on PERCS and 6D Torus networks. As before, this figure assumes

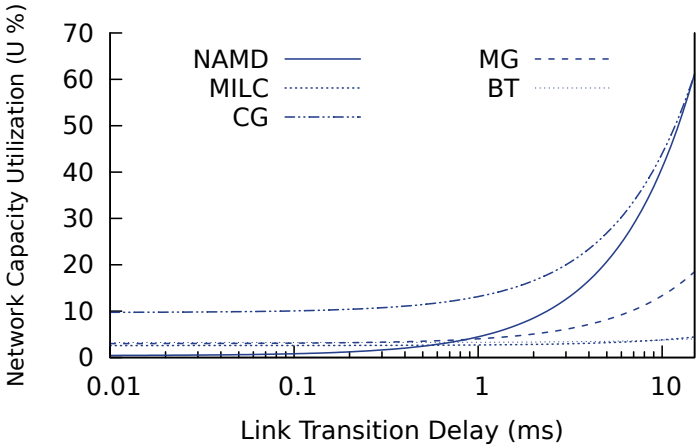


Figure 5.11: Potential link power saving on PERCS network.

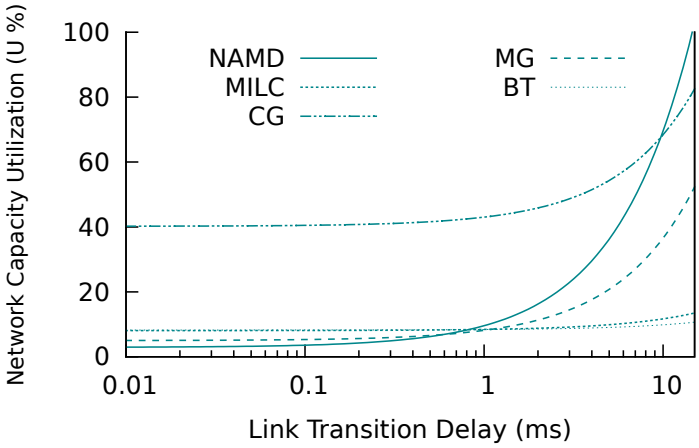


Figure 5.12: Potential link power saving on 6D Torus network.

that 30% of the machine power is consumed in the network and 65% of network power is consumed by the links. As can be seen, for most applications, our scheduling approach can save around 20% of the machine power. Our basic approach can also save significant power, usually more than 15% for PERCS and around 10% for 6D Torus. Note that in the case of NAMD_PME, the basic approach cannot save much and the scheduling approach is required.

5.6 Conclusions and Future Work

With ever increasing communication demands of large-scale parallel systems, multilevel directly connected networks (Dragonfly, PERCS) and high dimensional tori are becoming more appealing. Optimizing the power and performance of these innovative networks presents a new challenge for parallel systems. We showed that many parallel applications do not fully

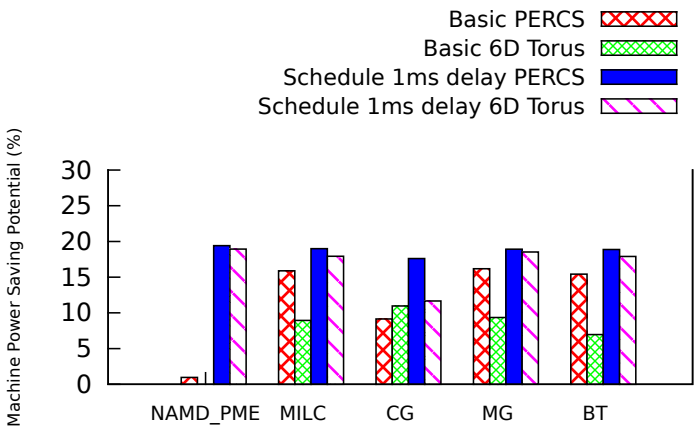


Figure 5.13: Potential total machine power saving for different approaches.

exploit a significant fraction of the network links, which present opportunities for power optimization. Thus, a runtime system can optimize the power consumption of the links by turning off the unused ones, with minimal hardware support. This approach results in up to 20% saving of total system's power for common place applications with near neighbor communication.

For future work, less conservative approaches that turn off more links can be used, which may have some performance penalties. Furthermore, dynamic voltage scaling (or reducing the bandwidth) of the network links can be exploited for the links that do not transfer messages on the critical path. Overall, we suggest that more adaptive power management techniques by the runtime system for the network should be explored further.

CHAPTER 6

Runtime Scheduling in Presence of Process Variation Heterogeneity

Process variation is the deviation of transistor parameters from their design (nominal) values, which is caused by systematic effects (e.g., lithographic inconsistencies) and random effects (e.g., varying dopant concentrations) [157]. Affected parameters include effective channel length, channel width, and threshold voltage. Therefore, transistor characteristics such as switching speed and current leakage can vary widely across the chip. Incorporating very small feature sizes in succeeding CMOS technology generations and lowering the supply voltage, which is necessary for power efficiency [5], exacerbate the process variation problem.

At the architectural level, process variation results in cores and on-chip memories having different frequencies and static power consumption profiles. The reason is that a core's frequency is determined by the switching speed of the transistors on its critical path, which depends on the characteristics of those transistors. In addition, static power has an exponential relationship with the threshold voltage of transistors. Many designers tackle this issue by leaving design margins, but this solution is deemed too wasteful, especially for future generation technologies and many-core architectures. A recent study estimated the within-die frequency variation of many-core chips in 11nm technology generation to be 2.3x for conventional high voltage operation (known as Super-Threshold Voltage Computing) and 3.7x for very low voltage operation (known as Near-Threshold Voltage Computing) [6].

Process variation leads to high heterogeneity in processor chips, which has direct consequences to High Performance Computing (HPC) environments. For example, frequency heterogeneity can slow down most multithreaded applications written in various parallel programming paradigms such as MPI [158] and Pthreads [159]. This is because parallel programmers usually assume different cores/processors have the same speed and assign the

computational load uniformly. In addition, the execution of different processes/threads is synchronized in most HPC applications. Therefore, the slowest core determines the execution time, unless variation-aware load balancing is performed (as we evaluate in Section 6.3).

Furthermore, heterogeneity makes power and energy management harder, since different parts of the chip can have widely different performance and power characteristics. The challenge is to utilize future HPC machines efficiently while staying within the performance, power, and energy constraints. Previous studies have developed scheduling heuristics for multiprogrammed environments [6, 160, 161], but HPC environments are different because usually only one parallel application runs on the whole chip. We strive to solve the performance, energy, and power problem for heterogeneous chips by developing a novel scheduling framework, which can be implemented in intelligent HPC runtime systems. The only requirement is being able to migrate work units and balance the load according to the configuration chosen by our framework. Our solution does not change the programming paradigms and existing codes and has negligible execution time overheads.

For an application running on a processor chip, the runtime has to choose a configuration among potentially billions of options. Each configuration instructs how many and which cores will execute the parallel program, and leaves other cores off. For a chip with n frequency domains, there are $2^n - 1$ configurations[1]. This translates to 67 billion configurations for a chip proposed by previous work with $n = 36$ [6]. The number of frequency domains is expected to be even larger for future Exascale many-core chips, resulting in an enormous number of possible configurations. Testing all of these configurations is infeasible for the runtime system. Since Exascale architectures are also very likely to be over-provisioned [3, 4], variation-aware power management is essential for them to stay within their power budget. We develop a scheduling framework with accurate performance and power models that explores the combinatorial search space efficiently and finds a close to optimum configuration quickly. It only needs a few samples of application execution to build the required models.

Performance modeling efforts for parallel applications have usually assumed that different processor cores have the same speed [162,163], or the system is heterogeneous but there are a few processor types (e.g. GPU and other accelerators) [164,165]. However, process variation causes a new form of heterogeneity that potentially makes all of the cores/processors of the system different. We develop and study the accuracy of four different performance models and apply the most accurate one for making scheduling decisions. Studying performance models also gives us insight into the impacts of heterogeneity on performance and power consumption.

[1] 'all cores off' is not useful

For a large chip with many frequency domains, evaluating even simple models for all the configurations is infeasible. Therefore, we use integer linear programming (ILP) to explore the search space efficiently. Our results show that our ILP-based scheduling provides configurations that perform 25% better on average for a compute-bound application and 16% better for a memory-bound application as compared to scheduling algorithms based on heuristics. In some cases, our framework finds configurations that are up to 2.5 times faster than the baseline heuristic. Furthermore, we demonstrate how different performance and power scheduling constraints can be expressed as linear models to be used by our ILP scheduling framework. Since ILP provides optimality guarantees (assuming that the models are accurate), our framework can be used to evaluate simpler scheduling heuristics as well.

The rest of the chapter is organized as follows. Section 6.1 presents background on process variation. Section 6.2 describes our evaluation setup. Section 6.3 discusses the requirements of programming systems, and evaluates the load imbalance caused by process variation heterogeneity. In Section 6.4, we design and evaluate different performance models. We use these models in Section 6.5 to define an ILP-based scheduling framework. We evaluate this framework versus simple heuristics in Section 6.6. We discuss the related work in Section 6.7, and conclude in Section 6.8.

6.1 Background on Process Variation

Ideally, all transistors of a die should be identical and have the same parameters as designed, but this is hard to achieve in manufacturing. Therefore, there are static, spatial fluctuations of parameters around the nominal values. The variation of transistors across different dies is called *die-to-die variation*, while the difference of transistors on the same die is called *within-die variation*.

Variation affects two critical parameters of transistors: threshold voltage (V_{th}) and effective channel length (L_{eff}). These parameters determine switching speed and leakage of the transistors. At the architectural level, process variation causes some processor cores to run faster or slower than the intended design. This is determined by the speed of the transistors on the critical path of each core. Figure 6.1 illustrates the frequency variability of a hypothetical chip. In addition, the static power consumption of different cores and on-chip memory units is determined by the leakage of their transistor, which varies.

To display a uniform view of the system to the user, the designers usually include margins to cover variations. However, due to growing variations in successive processor generations, researchers believe that this will become too costly [166]. For example, by using all the cores

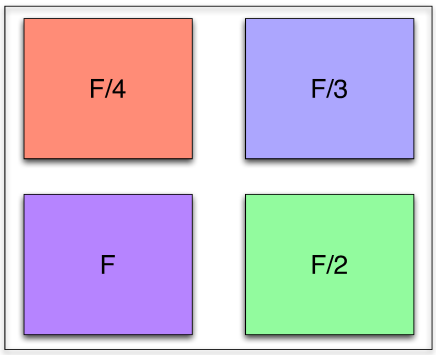


Figure 6.1: An example of core frequency variation on the same chip.

at a very low frequency, one pays the static power cost of all the cores but achieves limited performance. On the other hand, to alleviate power limitations, low voltage operation seems to be required [5], but it will exacerbate the variation issues. Hence, process variation needs to considered for future processor chips.

6.2 Evaluation Setup

For evaluation of our approach, we model heterogeneous chips using the Sniper simulator [167]. We use Sniper's default core model, which is similar to the Intel's Gainestown microarchitecture and has been validated [167].

To model process variation at the micro-architecture level, we use VariusNTV [168]. It models systematic variation by dividing the die into a grid, and assigning V_{th} and L_{eff} to each point by sampling from a multivariate Gaussian distribution. It models both spatially correlated variation and random variation, and its results have been validated against chip measurements [168].

We simulate 12-core or 36-core chips with each core in a different frequency domain, but one voltage for the whole chip. In the following sections, we refer to frequency domains simply as cores for convenience. Some previous works propose multiple small cores (a cluster) in each frequency domain. However, we use a simpler architecture to be able to evaluate our models more accurately and to simulate the possible configurations exhaustively for some cases. We believe that our results extend to other architectures as well. We have verified our simulation setting by comparing the simulation results of a corresponding homogeneous architecture against a 12-core Ivy Bridge machine. Table 6.1 presents the parameters of the modeled system in this chapter.

We use MiniMD and Jacobi3D to represent typical HPC workloads. MiniMD represents

Table 6.1: Simulated processor's parameters.

Sniper parameters	
Chip	12 or 36 Core CMP
Core	x86, 4-wide issue out-of-order
Instruction L1 (L1I)	32 KB, 4 way
Data L1 (L1D)	32 KB, 8 way, private.
L2	256 KB, 8 way, private.
Memory latency (no contention)	75ns
VariusNTV parameters	
Technology	11 nm
Average frequency	2.6 GHz
V_{dd}	0.765
Correlation range Φ	0.1
Total (σ/μ) for V_{th}	15%

molecular dynamics workloads, which are compute-intensive. Jacobi3D represents stencil computations, which are typically memory-bound. Other HPC applications are typically between these two in terms of being compute-intensive or memory-bound. We use instructions per cycle (IPC) as a proxy for application's performance. In each simulation, after initialization, and warm up for two seconds, we run the application for about six seconds of simulated time, and take the average of the several IPC samples collected at the intervals of milliseconds over the period of six seconds. Since the configuration is heterogeneous and frequencies are not the same, we normalize the IPC statistics of different cores by multiplying them with the frequencies of the corresponding cores and dividing by the frequency of the slowest core on the chip. We use McPAT [169] to evaluate dynamic power consumption and VariusNTV to evaluate static power consumption.

6.3 Programming Systems

Some parallel programming systems provide adaptive features such as automatic load balancing. Essentially, a scheduler decides how much work is assigned to each core. For example, an OpenMP runtime system can assign loop iterations to idle cores dynamically. Furthermore, the Charm++/AMPI [119] runtime system measures the load of different processors and balances the load accordingly.

In the context of process variation, runtime scheduling is even more important. For example, the runtime system should assign more work to faster cores and less work to slower ones. Otherwise, a multithreaded application will run at the pace of the slowest core. Figure 6.2 illustrates how load balancing is performed by migrating units of work among

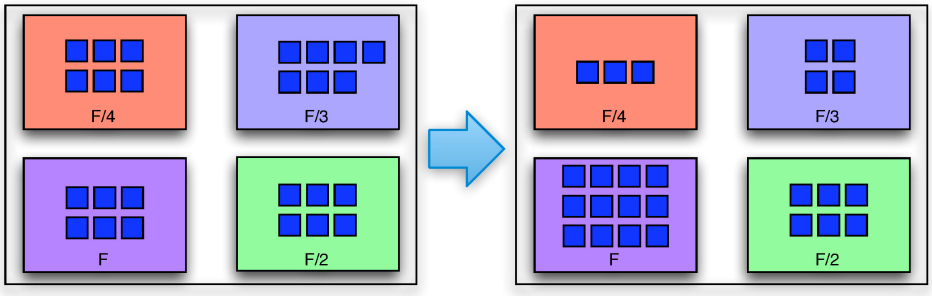


Figure 6.2: An example of load balancing across cores with different frequencies.

different cores. In this study, we assume that the runtime system has a way of assigning units of work to different cores, but our results do not depend on how it is done.

6.3.1 Impact on Load Balance

In this chapter, we assume that the runtime can achieve perfect load balance, but this is not always the case. Since the speeds of cores can be widely different, dividing a fixed number of similar tasks according to their speeds is not always possible. For instance, if there are three cores with frequencies of 1 GHz, 2 GHz, and 3 GHz, 7 equal tasks cannot be scheduled perfectly. In this case, at least one core has more work than the average load, and some other cores will be waiting for it to finish.

The work units cannot always be fine-grained enough to achieve perfect load balance. This is because scheduling and managing very fine-grained work units has high overhead. For example, in Charm++ and AMPI, usually a fixed number of tasks (e.g. *chares* or MPI ranks) are scheduled on different cores. In addition, in shared-memory approaches such as OpenMP and Intel TBB, chunks of iterations are scheduled by the runtime system. In this section, we strive to quantify the load imbalance due to work units not being fine-grained enough to be balanced perfectly on the chips with high variation.

We characterize the impact of the granularity of tasks (for a fixed computation) on load imbalance caused by variation in different configurations of a chip. A configuration is a subset of the cores on the chip, on which the parallel application will be run. We define the overdecomposition ratio as the ratio of the number of tasks to the number of cores. We also define our load imbalance metric using the ratio of maximum load to average load [170]:

$$\mathcal{I} = (\frac{L_{max}}{L_{avg}} - 1) \times 100 \tag{6.1}$$

where, L_{max} is the maximum load of any core in the configuration, and L_{avg} is the average load of all the cores in the configuration. These load values are obtained by appropriately scaling the assigned load with the frequencies of the corresponding cores. We use a greedy algorithm for load balancing, which is outlined in Algorithm 3. The algorithm assigns a *capacity* for work to each core, which is equal to its frequency (lines 2-4). In this way, more work will be assigned to faster cores. It then makes a heap (line 5), which has the core with the maximum available capacity at the top. The main loop (lines 6-14) removes the core with the maximum capacity from the heap, assigns a work unit to it, and adds it back to the heap. If there was not enough capacity on that core, the algorithm increases the capacity of all the cores by their frequencies (lines 8-10). This load balancing problem can be modeled as a variable size bin packing problem [171]. Theoretical analysis of this algorithm is left for future work.

Algorithm: VariationAwareGreedyLoadBalancing
Input: N work units, C cores
for *each core* $c \in C$ **do**
    c.capacity $\leftarrow$ c.frequency;
end
$H \leftarrow$ MakeMaxHeap(C);
for *each work_unit* $n \in N$ **do**
    $c \leftarrow$ RemoveMaxHeap(H);
    while $c.capacity < n.load$ **do**
        IncreaseAllCapacities(C);
    end
    c.workUnits $\leftarrow$ c.workUnits $\cup$ n;
    c.capacity $\leftarrow$ c.capacity $-$ n.load;
    AddMaxHeap(H,c);
end

Algorithm 3: Greedy variation-aware load balancing algorithm.

Figure 6.3 summarizes the load imbalance on various configurations of a 12-core chip. We simulated Algorithm 3 with different over-decomposition ratios for all the $2^{12} - 1$ configurations of enabled and disabled cores exhaustively, assuming that the work units are of equal size. The two bars correspond to the average load imbalance across all the configurations and the maximum load imbalance for any configuration, respectively. As illustrated, more over-decomposition reduces the load imbalance rapidly. With an over-decomposition ratio of 16, the average load imbalance across all the configurations is 2%, while the maximum is 6%. Since an over-decomposition ratio of 16 is already in use for some current Charm++ applications [172] and seems extensible to most applications, we conclude that load imbal-

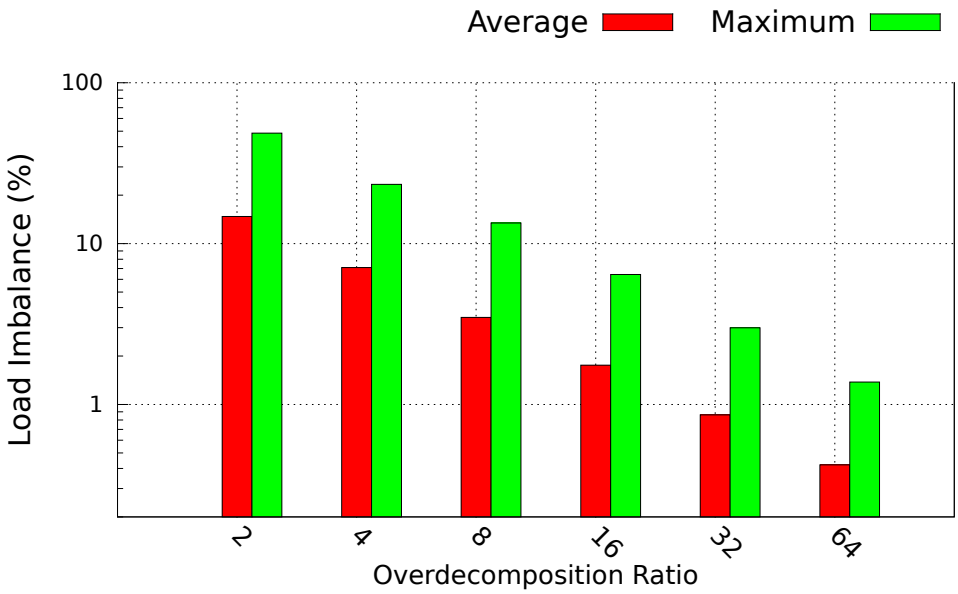


Figure 6.3: The average and maximum load imbalance across all configurations with different over-decomposition ratios.

ance is not a fundamental problem for our approach. Therefore, in later sections, we assume that the runtime system can balance the load almost perfectly on any configuration chosen by the scheduler.

6.4 Performance and Power Modeling

To be able to find acceptable configurations, we need power and performance models that can evaluate each configuration accurately. Most models in the literature assume homogeneous chips [162, 163] or heterogeneous ones with only a few different processor types (such as GPUs and other accelerators) [164, 165]. Therefore, new models need to be developed for chips with high variation and heterogeneity. Building the models should require only a few (e.g. $O(n)$) samples, to be feasible for the runtime system to collect that many samples. In this section, we assume that the runtime system can balance the load perfectly.

Figures 6.4 and 6.5 compare the performance of miniMD and Jacobi3D with different number of cores and frequencies of a 12-core homogeneous systems. We use the insights from these plots to develop our models. For example, compute-bound applications, such as MiniMD, scale well with more cores and/or higher frequency. However, memory-bound applications do not scale beyond a certain point, since memory bandwidth becomes the bottleneck.

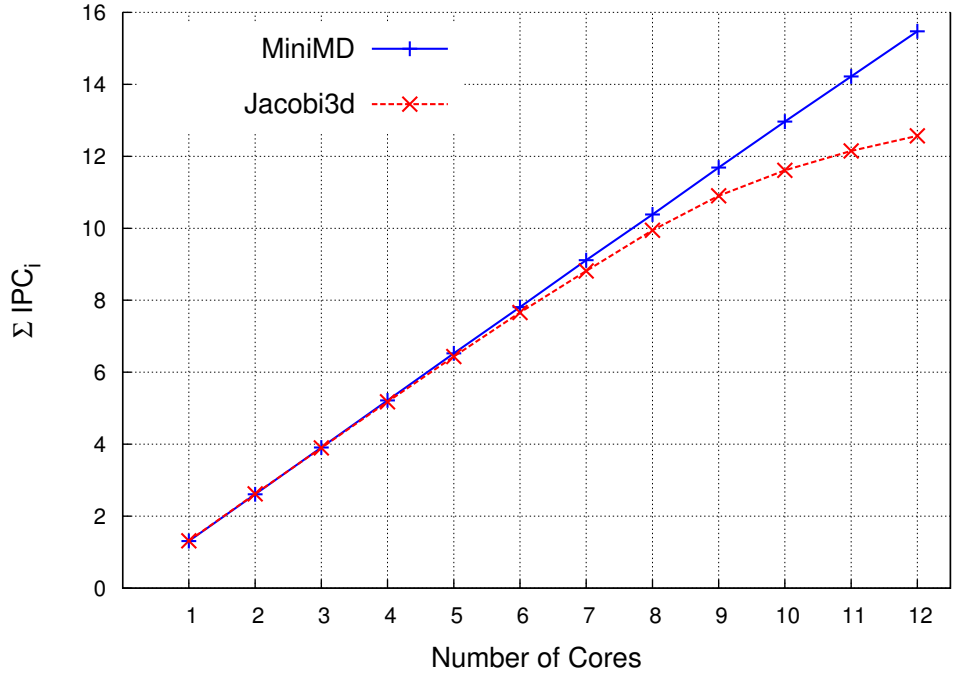


Figure 6.4: Performance scaling with cores.

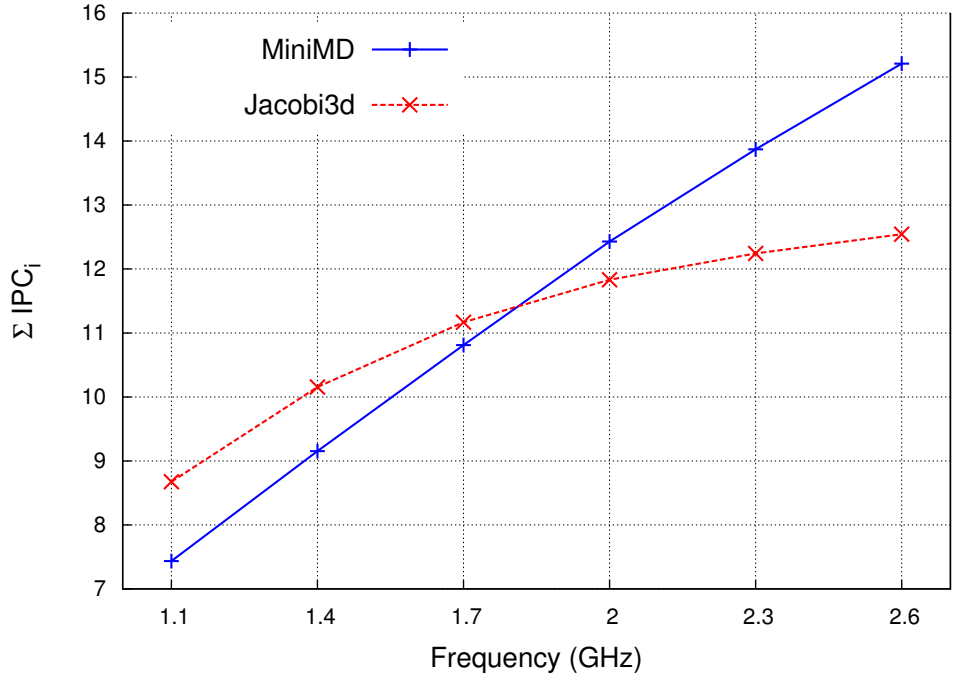


Figure 6.5: Performance scaling with frequency.

6.4.1 Model 1

Our first model assumes that adding each core to the configuration improves the application's performance proportionate to the core's frequency. However, this improvement is different for various applications, and therefore performance samples are needed.

We assign a performance value to each core (s_i). The performance of a configuration is modeled as:

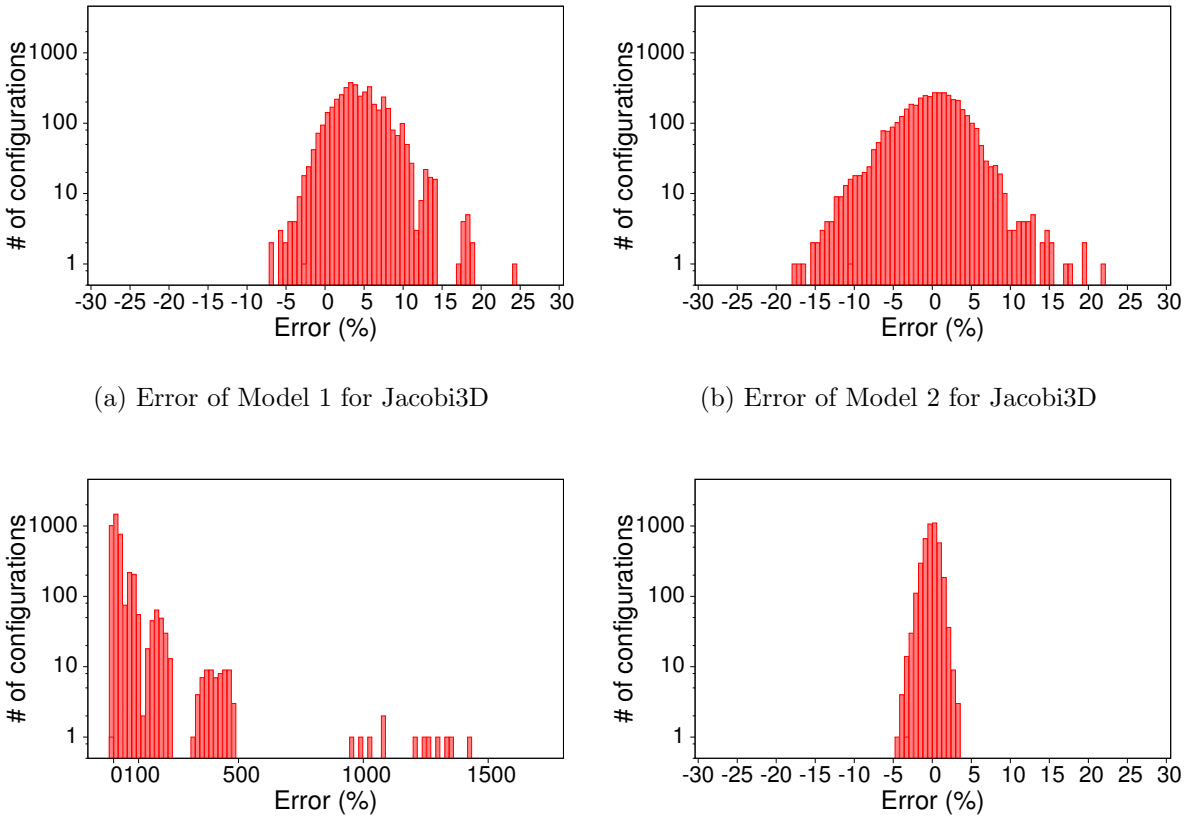
$$S_c = \sum_{i \in c} s_i \tag{6.2}$$

For each core i, s_i is the performance (normalized IPC) of the application when running on that core alone. Therefore, n samples can be used to obtain the s_i values. Note that since s_i values are proportionate to the frequencies of the corresponding cores (see Figure 6.5), one can sample the slowest and fastest cores and build a linear model to predict the other values. Hence, just two samples would suffice for this model as well.

Figures 6.6(a) and 6.7(a) show the error of Model 1 for Jacobi3D and miniMD. Each point corresponds to a configuration. All of the configurations of our 12-core chip (total number of configurations = 4095) are simulated and evaluated exhaustively. Positive values mean that the model overestimated the performance, while negative values illustrate that the model underestimated the performance. As can be seen in the figure, this model is accurate for miniMD but not for Jacobi3D. The reason is that, with more cores running at the same time, memory bandwidth becomes the bottleneck for Jacobi3D and the performance does not improve proportionately with the number of cores. The s_i measurements do not capture this effect since the performance of each core is sampled executing alone. This does not capture resource contention, such as memory bandwidth, that occur in the presence of other active cores. However, for a compute intensive code such as miniMD, this problem is not significant and sampling individual cores separately gives accurate performance predictions. In general, the disadvantage of Model 1 is that it does not consider the bottlenecks that limit the performance when there are multiple cores running at the same time.

6.4.2 Model 2

The second model tries to avoid the drawbacks of the first model by using a random set of samples that potentially have multiple cores running at the same time. Furthermore, it assumes that the application execution time has a memory component (T_{mem}) that does not become faster with more cores.



(a) Error of Model 1 for Jacobi3D

(b) Error of Model 2 for Jacobi3D

(c) Error of Model 3 for Jacobi3D

(d) Error of Model 4 for Jacobi3D

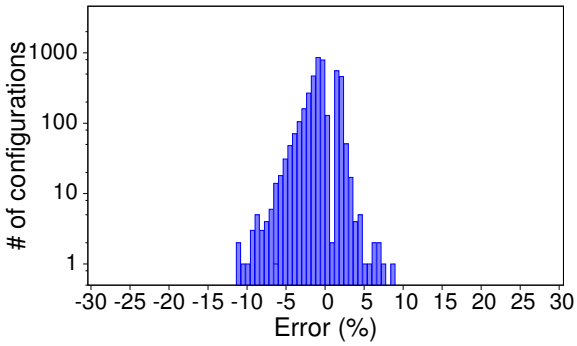
Figure 6.6: Distribution of errors of different models for performance of Jacobi3D. The number of configurations on y-axis is shown in log scale. Model 4 performs very well, with average prediction error of only 0.7% across all the configurations for Jacobi3D.

Model 2 uses the sum of the frequencies as the variable that determines the application's performance for each configuration. It uses n random samples to fit a linear function of the following form:

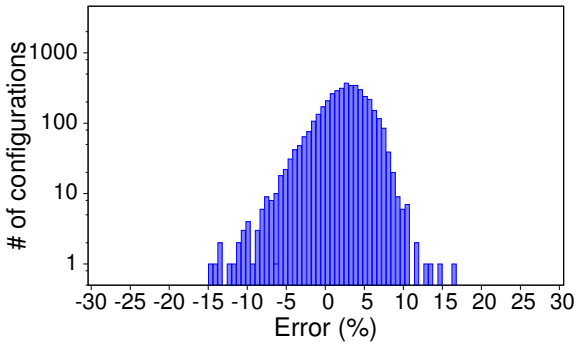
$$F_c = \sum_{i \in c} f_i \tag{6.3}$$

$$t_c = \frac{T_{comp}}{F_c} + T_{mem} \tag{6.4}$$

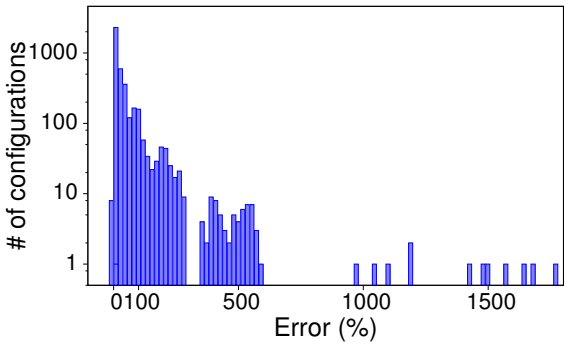
In this equation, t_c is the predicted execution time of configuration c. T_{comp} and T_{mem} are constants that represent the computation and memory time of the application (which are found by function fitting). F_c is the sum of the frequencies of the cores of the configuration.



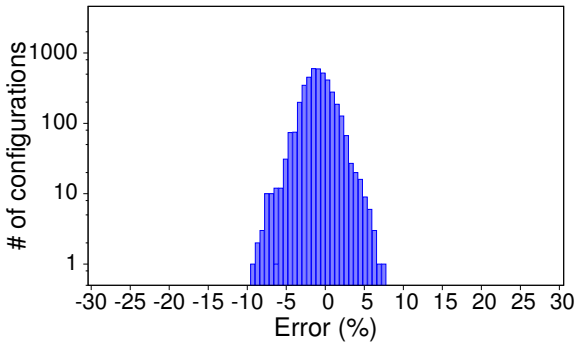
(a) Error of Model 1 for miniMD



(b) Error of Model 2 for miniMD



(c) Error of Model 3 for miniMD



(d) Error of Model 4 for miniMD

Figure 6.7: Distribution of errors of different models for performance of miniMD. The number of configurations on y-axis is shown in log scale. Model 4 performs very well, with average prediction error of only 1.6% across all the configurations for miniMD.

Figures 6.6(b) and 6.7(b) show the error of Model 2 for Jacobi3D and miniMD. The results illustrate that this model is not very accurate in predicting the performance. Similarly, one might fit a linear function of the sum of the frequencies with the following form:

$$S_c = a_1 F_c + a_2 \tag{6.5}$$

However, we verified that this model does not predict well and has similar accuracy to Model 2. These models do not work because they fail to take into account the number of cores that are executing the parallel application.

6.4.3 Model 3

To have more accurate predictions than Model 2, one might consider second degree curve fitting. In this model, the runtime samples n random configurations as before, but fits a second degree curve to predict the performance:

$$S_c = a_1(F_c)^2 + a_2F_c + a_3 \tag{6.6}$$

Figures 6.6(c) and 6.7(c) show the prediction error of Model 3. There are extremely large error values for some configurations, which make the use of this model impractical. This model suffers from *overfitting*; although the curve is very close to the few sample values, it cannot capture the trend and can be very inaccurate for other configurations.

6.4.4 Model 4

Previous models could not capture various aspects of the configurations simultaneously (such as memory bandwidth limitations and frequency sensitivity). Considering Figures 6.4 and 6.5, we observe that both the frequencies and the number of cores influence the performance. Moreover, just adding the frequencies does not capture the effect of additional cores. For example, two 1 GHz cores are not the same as one 2 GHz core. Therefore, an accurate model needs to consider the frequencies as well as the number of cores in each configuration.

To consider both the frequencies and number of cores, Model 4 fits a linear function for each possible number of cores, i.e. one line for configurations with only one core, one line for configurations with two cores, and so on. Therefore, a 12-core chip will have 12 linear functions. A line for k-core configurations is a linear function of the sum of frequencies of the k cores of each configuration.

In general, each line needs at least two samples but more samples can make it more accurate by using regression tools to fit the best possible line for all the samples. To keep the number of samples needed low, we use only two samples. Hence, $2n$ samples are needed, which is more than other models but still low enough for the runtime system to collect with negligible overhead. As a heuristic, we choose the configuration with the minimum sum of frequencies and the configuration with the maximum sum of frequencies for sampling. Finding the configuration with minimum (or maximum) sum of frequencies is easily done by choosing the needed number of cores from the list of cores sorted by frequencies.

The runtime system builds the model as follows.

$\forall k \in (1..n)$:

$K = \{c | c \text{ has } k \text{ cores}\}$, where c is a configuration

$$c^k_{min} = c \in K | \sum_{i \in c} f_i \text{ is minimum} \tag{6.7}$$

$$c^k_{max} = c \in K | \sum_{i \in c} f_i \text{ is maximum} \tag{6.8}$$

$$F^k_{min} = \sum_{i \in c^k_{min}} f_i \tag{6.9}$$

$$F^k_{max} = \sum_{i \in c^k_{max}} f_i \tag{6.10}$$

$$Y^k_{min} = \text{performance of } c^k_{min} \tag{6.11}$$

$$Y^k_{max} = \text{performance of } c^k_{max} \tag{6.12}$$

$$a^k_1 = \frac{(Y^k_{max} - Y^k_{min})}{(F^k_{max} - F^k_{min})} \tag{6.13}$$

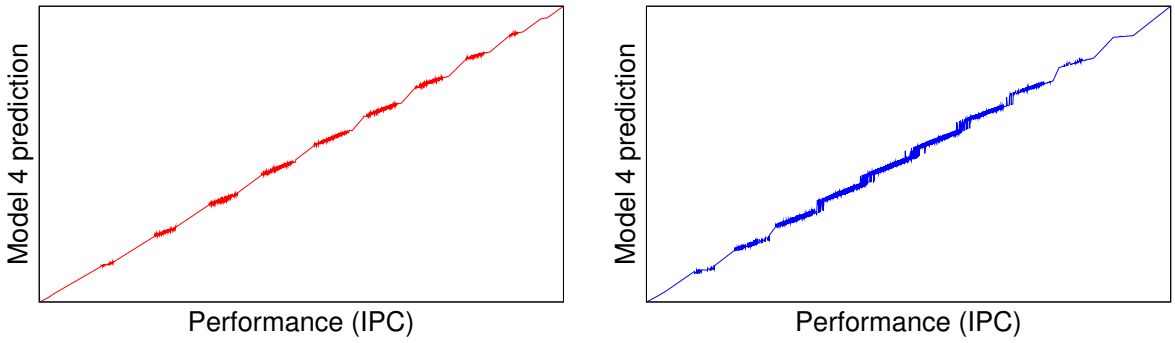
$$a^k_2 = Y^k_{min} - a_1 F^k_{min} \tag{6.14}$$

The performance of any configuration with k cores is then predicted by a simple linear formula:

$$S^k_c = a^k_1 F_c + a^k_2 \tag{6.15}$$

Figures 6.6(d) and 6.7(d) show the accuracy of Model 4 for Jacobi3D and miniMD. Compared to other models, the points are closer to the zero error line, meaning that this model is much more accurate.

For each application, Figure 6.8 illustrates the performance predictions of Model 4 for the chip configurations as a function of their actual performance obtained by simulation. There are some jitters but both of the functions are mostly monotonic. This means that, given two configurations, the model predicts higher performance for the configuration that is actually faster. Therefore, Model 4 is accurate at comparing configurations.



(a) Jacobi3D (b) miniMD

Figure 6.8: Model 4 predictions as a function of actual (simulated) performance.

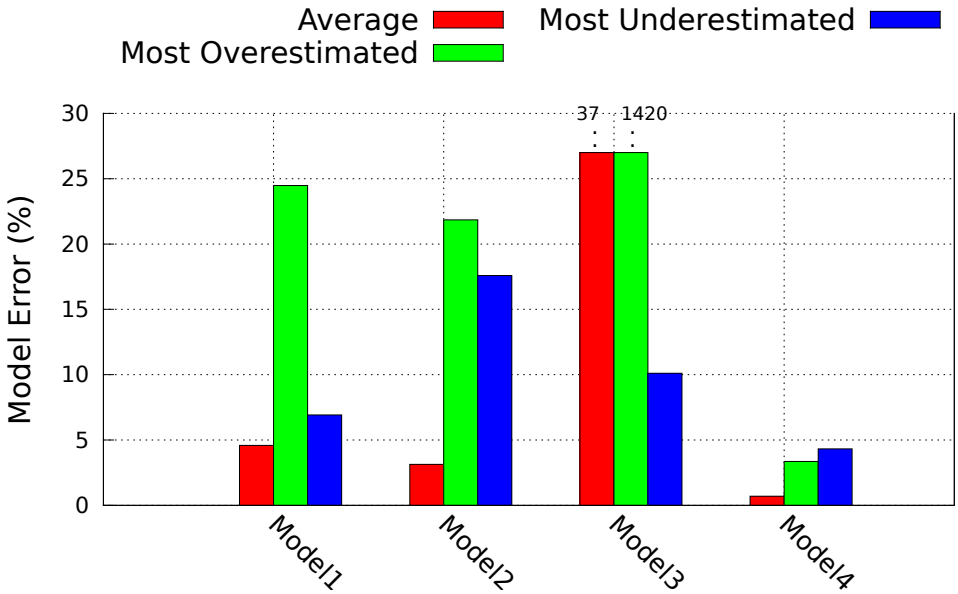


Figure 6.9: Prediction accuracy of different models for Jacobi3D. Numbers on top of the Model 3 bars represent the values that are beyond the plotted range.

6.4.5 Summary of Performance Models

Figures 6.9 and 6.10 compare the accuracy of the discussed models. Model 4 is superior to others in all of the metrics. The average error of Model 4 for miniMD is 1.6%, and it is 0.7% for Jacobi3D. In the worst case, the maximum error of Model 4 is 9.2% for miniMD and 4.3% for Jacobi3D. Thus, we conclude that Model 4 is sufficiently accurate for predicting the performance of various configurations of a heterogeneous chip.

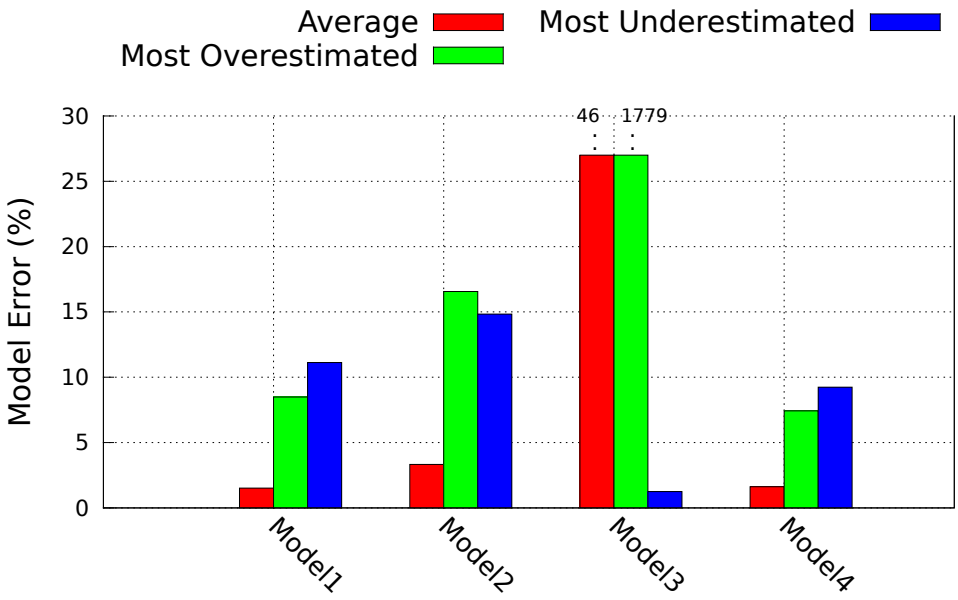


Figure 6.10: Prediction accuracy of different models for MiniMD. Numbers on top of some bars represent the values that are beyond the plotted range.

6.4.6 Modeling Dynamic Power

Model 4 predicts performance accurately but dynamic power also varies with configuration and needs to be predicted. The dynamic power of a processor core can be formulated as follows:

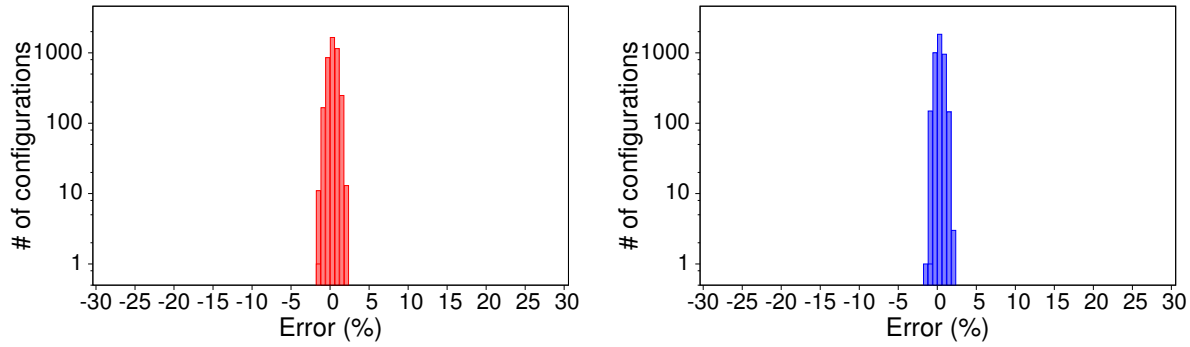
$$D_i = \alpha_i C V^2 f_i \tag{6.16}$$

In this formula, f_i is the frequency of the core, V is its voltage, C is its capacitance, and α_i is the activity of the core. Except the activity level of the core α_i, which varies in different configurations, other parameters are constants. Therefore, α_i needs to be taken into account for accurate dynamic power predictions.

The dynamic power of a configuration is the sum of the dynamic powers of the cores:

$$D^c = \sum_{i \in c} \alpha_i^c C V^2 f_i = C V^2 \sum_{i \in c} \alpha_i^c f_i \tag{6.17}$$

We strive to adopt our performance model (Model 4) to predict dynamic power due to the following observations. First, dynamic power has similar properties to performance in general. Dynamic power is higher when there are more cores and when the cores have higher frequencies. Second, the activity level of each core is correlated with performance, and hence, correlated with the sum of frequencies. We therefore formulate our dynamic power model as follows:



(a) Jacobi3D (b) miniMD

Figure 6.11: Distribution of errors of Model 4 for power consumption prediction.

$$Z^k_{min} = \quad \text{dynamic power of } c^k_{min} \tag{6.18}$$

$$Z^k_{max} = \quad \text{dynamic power of } c^k_{max} \tag{6.19}$$

$$b^k_1 = \quad \frac{(Z^k_{max} - Z^k_{min})}{(F^k_{max} - F^k_{min})} \tag{6.20}$$

$$b^k_2 = \quad Z^k_{min} - a_1 F^k_{min} \tag{6.21}$$

The dynamic power of a configuration with k cores is then predicted by a simple linear formula:

$$D^k_c = b^k_1 F_c + b^k_2 \tag{6.22}$$

Figure 6.11 illustrates the accuracy of our model for predicting the dynamic power of all of the configurations of a 12-core chip. The errors of the model are all less than 2%, which means the accuracy is very high. Furthermore, since the accuracy of the model is higher for dynamic power than performance, one can conclude that predicting dynamic power is easier than performance.

6.5 Model Driven Scheduling

The runtime system should choose the frequency domains of the chip that execute the application intelligently in order to meet the performance, power, and energy constraints. There are various tradeoffs that need to be considered. For example, choosing a configuration with too many cores for a memory-bound application might not improve the performance much, but it can consume excessive power and energy. In addition, a configuration might be fast but have high power consumption.

The number of configurations can be prohibitively large for the runtime system to try exhaustively. For a chip with n frequency domains, there are $2^n - 1$ configurations since each domain can be turned on or off (with at least one domain on). This exponential growth is due to the heterogeneity of the chips. Otherwise, analogous homogeneous chips with n cores have only n distinct configurations. In the previous section, we developed a model to predict the running application's performance on any configuration of the cores of a heterogeneous chip. In this section, we use the model to solve the scheduling problem in the presence of variation, given the performance and power constraints.

6.5.1 Efficient Configuration Space Exploration

Using our performance and power models, the runtime system can evaluate many configurations quickly (with only a few samples), but exhaustive exploration is not always practical. For our example processor chip with 12 frequency domains, the runtime only needs to evaluate the models for $(2^{12} - 1 = 4095)$ configurations. However, future processors will have more frequency domains and the number of configurations increases exponentially with the number of frequency domains. Therefore, exploring all of the configurations in the runtime can be impractical.

We propose the use of integer linear programming (ILP) by the runtime system for finding the best (or very close to the best) configuration given the performance and power constraints. Using ILP (in many cases) needs linear objective functions and constraints. Our performance and power models are linear for configurations with the same number of cores, but the overall functions are not linear. To solve this issue, we setup separate ILP problems for configurations with different number of cores (36 ILP problems for a 36-core chip). Therefore, to maximize performance given a power budget, an ILP problem for a given number of cores (k) is formulated as follows :

Parameters

x_i : binary variable indicating whether core i is used

$$F = \sum_{i \in \text{all cores}} x_i f_i$$

$\mathcal{P}$: power budget of the chip

p_i^s : static power of core i

Objective function

Maximize performance:

$$S_c^k = a_1^k F + a_2^k \tag{6.23}$$

Constraints

Only configurations with k cores:

$$\sum_{i \in \text{all cores}} x_i = k \tag{6.24}$$

Cap total power according to budget:

$$\sum_{i \in \text{all cores}} x_i p_i^s + b_1^k F + b_2^k \leq \mathcal{P} \tag{6.25}$$

After solving these ILPs, the runtime system needs to compare the results and choose the best one. Note that in our formulation, we considered performance as the main objective metric that needs to be maximized given a power budget. One can similarly minimize power given performance constraints. In this case, the roles of Equations 6.23 and 6.25 are switched, and performance becomes a constraint, while power becomes the objective function.

Note that if one wants to minimize energy without any performance constraints, our ILP framework in this form cannot be used since the objective function will not be linear anymore. Energy minimization without performance constraints is left for future work.

6.5.2 Incorporating DVFS

Previous studies suggest Dynamic Voltage and Frequency Scaling (DVFS) for energy-efficient computing for some cases, such as for memory bound applications. Our framework can incorporate DVFS as well. We only need more binary variables and constraints that indicate at which DVFS level each core should operate. The constraints make sure that the solver does not choose illegitimate conditions, such as a core operating at two DVFS levels simultaneously.

Parameters

x_{ij} : binary variable indicating core i at DVFS level j is used or not

$$F = \sum_{i \in \text{ all cores}} \sum_{j \in \text{ all DVFS levels}} x_{ij} f_{ij}$$

$\mathcal{P}$: power budget of the chip

p_i^s : static power of core i

Objective function

Maximize performance:

$$S_c^k = a_1^k F + a_2^k \tag{6.26}$$

Constraints

Only configurations with k cores:

$$\sum_{i \in \text{all cores}} \sum_{j \in \text{ all DVFS levels}} x_{ij} = k \tag{6.27}$$

Only one DVFS level is selected per core:

$$\sum_{j \in \text{ all DVFS levels}} x_{ij} \leq 1, \forall i \in [1..n] \tag{6.28}$$

Cap total power according to budget:

$$\sum_{i \in \text{all cores}} \sum_{j \in \text{ all DVFS levels}} x_{ij} p_i^s + b_1^k F + b_2^k \leq \mathcal{P} \tag{6.29}$$

Further study and evaluation of DVFS in our framework is left for future work. For the chips we evaluate, the static power is high and therefore, operating more cores at lower frequencies is not very well justified, even for memory-bound applications. It is most often more beneficial to turn as many cores off as possible and operate the rest at full speed. However, DVFS might be beneficial in other cases.

6.5.3 Incorporating Communication Performance

So far we have assumed that on-chip communication performance of different configurations does not differ significantly, but this can be incorporated in our framework as well. Depending on several factors, such as the application's communication pattern, network design, and

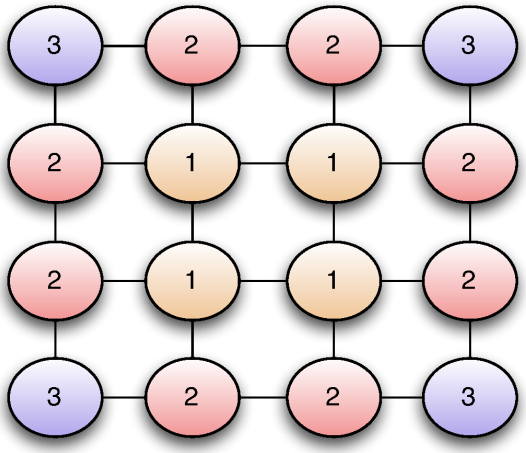


Figure 6.12: Assigning communication scores to different cores on a chip.

the routing algorithm, a communication model needs to be included in the ILP framework.

As an example, we consider a case where the application has a heavy all-to-all communication pattern, the network topology is a 2D mesh, and a minimal adaptive routing algorithm is used. In this case, one heuristic is to use the cores that are farther apart on the chip. This lets the application use more of the network links and have less congestion.

To handle this case in our framework, communication performance needs to be incorporated in the objective function as a linear expression. We assign a communication score (e_i) to each core on the chip. Figure 6.12 shows an example of possible core assignments. The cores closer to the corners and sides are assigned higher scores because they can potentially use more of network's links to send and receive messages. Using this model, we extend the object function of the ILP as follows:

$$S_c^k = \mathcal{E}_1(a_1^k F + a_2^k) + \mathcal{E}_2(\sum_{i \in \text{all cores}} x_i e_i) \tag{6.30}$$

In this equation, $\mathcal{E}_1$ and $\mathcal{E}_2$ are constants that give weights to computation and communication performance depending on the application. They can be tuned offline, or online by the runtime system using measurements.

Further study and evaluation of communication models is beyond the scope of this study and is left for future work.

6.5.4 Adapting to Application Phases

Some HPC applications have multiple phases with different characteristics. Our previous study demonstrates how application phases can be recognized and exploited effectively in the runtime system (Chapter 4). One could use our scheduling framework for different phases separately as well. In this case, the runtime needs to migrate the tasks and turn cores on and off when changing the configurations. The overheads of this reconfiguration and task migration should also be considered. Extensive study of of this feature is left for future work.

6.6 Evaluation

In this section, we evaluate our ILP-based scheduling framework by comparing it against two heuristic based scheduling algorithms. The goal of these algorithms is to maximize the performance of a parallel application under a given power budget for the chip. Evaluation and analysis of cases that consider other aspects such as DVFS and communication constraints is left for future work.

The first heuristic, called the Min heuristic, chooses as many cores as possible with the lowest frequencies as possible such that the selected configuration remains within the given power budget and the performance is maximized. Although this heuristic does not consider static power for core selection, most often the slower cores also have lower static power. A previous study confirms this correlation using silicon measurements [166]. Hence, this heuristic strives to choose as many low power cores as possible. The second heuristic, called the Max heuristic, chooses as many of the highest frequency cores as possible, while staying within the power budget. In contrast to the first one, this heuristic therefore most often chooses the highest power consuming cores and probably fewer cores.

Figures 6.13 and 6.14 compare our ILP framework to the heuristics for miniMD and Jacobi3D applications. The bars represent the average benefits of the different schemes across 100 chips, and are normalized to the Min heuristic. The vertical lines on the ILP and the Max heuristic bars illustrate the maximum and minimum benefit obtained with the corresponding approach across all the chips for the given power budget. MiniMD's power caps are lower since it consumes less power in general compared to Jacobi3D. One can conclude that Min and Max heuristics have similar results, while intelligent ILP scheduling can be considerably better. ILP finds configurations that are up to 1.85 times faster for Jacobi3D and up to 2.5 times faster for miniMD. On average, for all the cases we examined (various power caps for 100 chips), ILP scheduling is 25% faster for miniMD and 16% faster for Jacobi3D as compared to the Min heuristic.

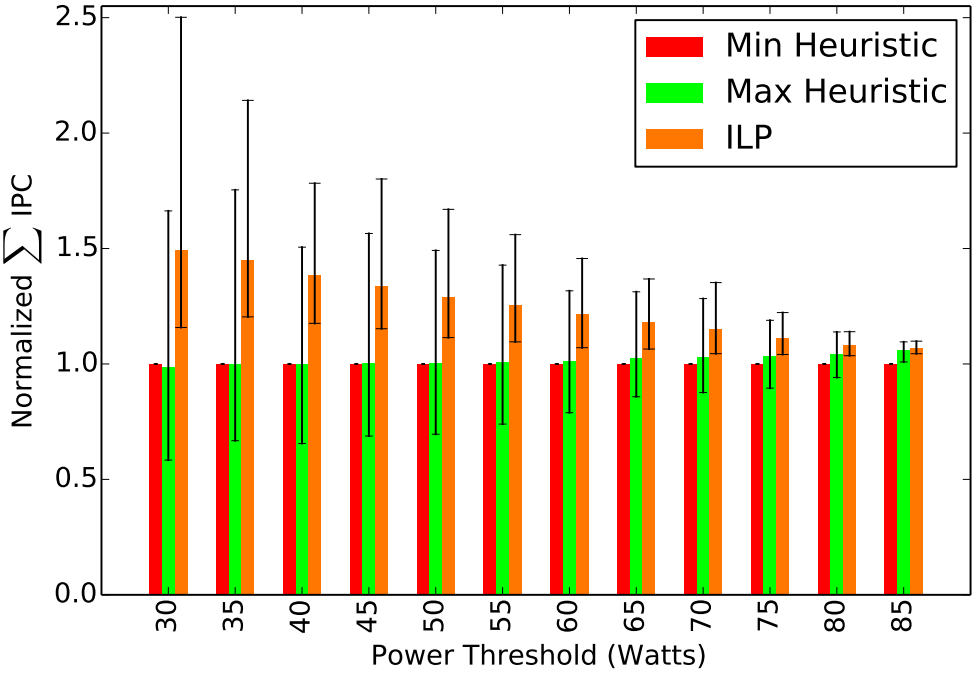


Figure 6.13: Comparison of our ILP-based scheduling approach to simple heuristics for miniMD.

The results indicate that the benefit of intelligent ILP scheduling is more with lower power budgets. This is because with lower power caps (that still allow multiple cores to be selected), there are more choices. On the other hand, if the power budget allows many of the cores to be selected, the different configurations chosen by different schemes have many overlapping cores and are similar. If there is enough power, all cores will be chosen by all of them and there is no other choice for the ILP. One can also see that the benefit of intelligent ILP scheduling can be much higher for compute-bound applications such as miniMD, since they are more sensitive to the frequencies of the chosen cores (as illustrated in Figure 6.5).

Table 6.2 presents an example scheduling case, demonstrating that the ILP is choosing cores intelligently and its choices are different from the heuristics. In this case, the power cap is 40 W and the schemes strive to find the highest performing configuration for a 36-core chip running miniMD. The cores are numbered by their frequencies in increasing order. One can conclude that ILP is choosing the cores intelligently resulting in higher performance within the power budget.

Integer program optimization is an NP-hard problem and can be computationally very expensive in the worst case, but it is very fast for our scheduling framework. Our ILP

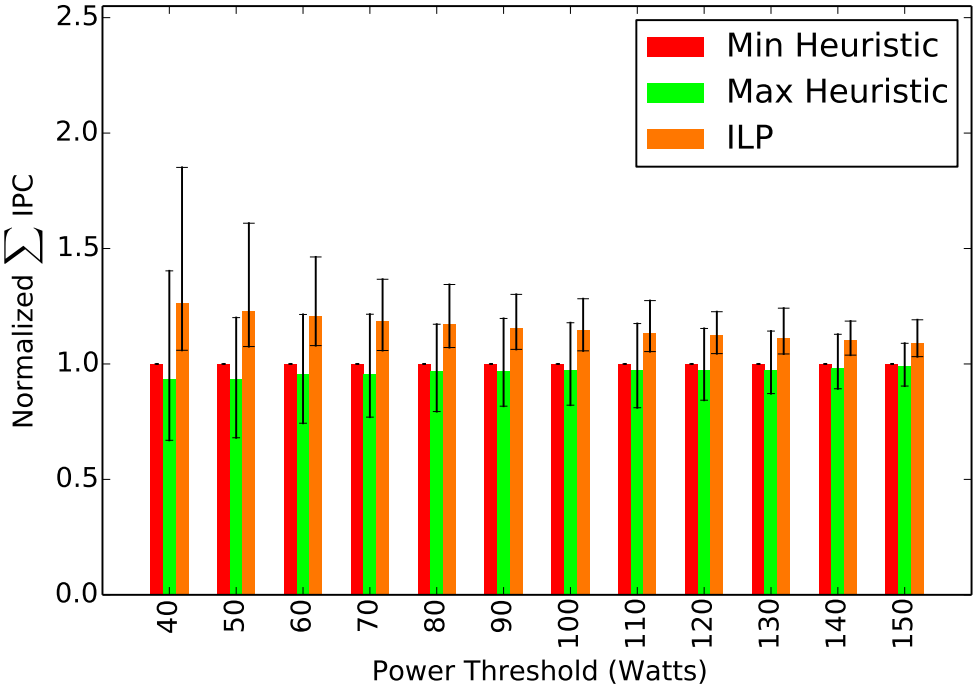


Figure 6.14: Comparison of our ILP-based scheduling approach to simple heuristics for Jacobi3D.

Table 6.2: Example scheduling case comparing various schemes.

Scheme	Selected Cores	Relative Performance
MIN Heuristic	$0, 1, 2, 3, 4, 5, 6$	1
MAX Heuristic	$31, 32, 33, 34, 35$	0.97
ILP	$7, 13, 14, 17, 21, 24, 31$	1.19

solver took only 37 ms on average across all the chips and power budgets we examined. The underlying simplex algorithm performed only 217 iterations on average across all the nodes of the branch-and-bound tree for the corresponding ILP. This is negligible compared to the typical execution time of HPC applications, which can run for up to several days in many cases. We used the state-of-the-art Gurobi [173] optimizer for solving the ILPs.

6.7 Related Work

Process variation has been explored from the manufacturing and circuit perspective [7, 157, 174, 175]. Dighe et al. [166] measured the process variation of Intel TeraFLOPS experimental chips and studied the optimal operating point of different applications. We use the valuable

insights of these studies for our assumptions about the properties of the processor chips.

The proposed Exascale architectures such as Runnemede [3] and Echelon [4] consider hundreds of cores on a chip, which are arranged in many frequency domains (with power gating). Furthermore, those architectures are over-provisioned, and need extensive power management to stay within the power budget [3]. However, previous studies do not provide scheduling algorithms that meet these constraints. A framework like ours seems essential for such architectures, as it provides the necessary runtime component to perform scheduling and power management depending on the application characteristics.

Previous studies have explored scheduling in the presence of process variation for multi-programmed environments [6, 160, 161]. For example, Winter et al. [161] use a Hungarian optimization algorithm to assign different sequential programs to the best matching cores. Karpuzcu et al. [6] use heuristics to map different multithread applications to the many-core chip's cores. However, in HPC environments, almost always a single parallel application runs on the whole chip, and the previous algorithms cannot be used. Since all the threads of the application usually have similar behavior, switching threads among cores is not very useful. In addition, previous studies either ignored the interference of different cores [161] (e.g. memory bandwidth contention), or assumed that load balancing is not possible and the parallel application's speed is determined by the slowest core [6]. To the best of our knowledge, this is the first study to propose a variation-aware scheduling approach for HPC systems.

In their studies on overprovisioned HPC data centers, Sarood et al. [176] and Patki et al. [177], have proposed frameworks that distribute power to nodes such that the performance is maximized under a given power budget for the data center. However, they do not propose how the chosen power budget for each node will be used for achieving maximum performance from the node, which is the focus of this work. The proposed framework in this chapter can be combined with their frameworks for maximizing the performance of future HPC data centers with a fixed power budget.

6.8 Conclusion and Future Work

Process variation causes performance and power heterogeneity among various cores of a many-core chip. We studied various performance and power models for such chips. Based on the models, we developed a novel scheduling framework that uses integer linear programming (ILP) to explore the configuration space efficiently. This enables intelligent HPC runtime systems to enforce different performance and power constraints and schedule work

units effectively. We also illustrated how different constraints, such as communication performance, can be enforced in our framework.

There are many future research directions based on our study. More in depth evaluation and analysis of different HPC applications on heterogeneous chips needs to be performed. Furthermore, the use of our framework to find different configurations for different application phases needs to be studied. In addition, we assumed that tasks of the application have the same amount of work but this is not always the case. Variation-aware load balancing of complex applications that have different workloads in different tasks needs further exploration. Moreover, the development and evaluation of new scheduling constraints dependent on the application and system characteristics requires much research.

CHAPTER 7

Concluding Remarks

In this dissertation, we used extensive analysis of common HPC applications and modern architectures to develop novel adaptive runtime system (RTS) methods that improve power and energy efficiency without significant performance overheads. The required hardware support is also very limited. Only the RTS layer has to change, which is inexpensive compared to changing the applications and the hardware.

Our conclusions and contributions can be summarize as follows:

- Many-cores like Intel SCC offer an opportunity to build future machines that consume low power and can run existing codes (e.g. written in Charm++ and MPI) fast.
- Accelerators, such as GPGPUs, are exceptionally powerful for some applications in terms of speed, power and energy efficiency but they require high programming effort.
- Heterogeneous on-chip architectures can be very effective, and a unified programming paradigm such as OpenCL provides a good balance between performance and programmer productivity.
- Mapping of the application kernels to heterogeneous architecture's different devices are important. Using only the GPU or trying to gain maximum utilization of both the CPU and the GPU naively, are not energy efficient (even for highly parallel vision workloads).
- Depending on the application, some other techniques such as software pipelining of kernels and trading accuracy for energy can be exploited for better energy efficiency.
- Common HPC applications usually have a regular structure, which can be exploited for various power and energy efficiency improvements.

- The RTS is able to recognize the applications pattern using formal language theory and predict its future. Therefore, it can adapt the hardware to achieve better power and energy efficiency.

- A fixed cache configuration is not energy efficiency for every application and some adaptation is required. Changing the number of active ways of set-associative caches is an easy way to adapt the cache hierarchy and save significant energy.

- Since some common HPC applications have streaming memory access patterns, switching to a reconfigurable prefetching scheme by the RTS can improve performance and energy efficiency significantly. To avoid the possible drawbacks of prefetching, the RTS can reconfigure prefetching depth and the cache size to use.

- Cache reconfiguration cannot be performed by the hardware using the system metrics easily. The reason is that modern processors are complicated and simple system metrics such as cache misses do not represent performance.

- Many common HPC applications do not fully exploit a significant fraction of the network links, especially on high-radix networks such as Dragonfly. This presents opportunities for power and energy optimization.

- We developed a theoretical model of network usage of applications that can estimate the possible energy savings by turning unused links off.

- We demonstrated that an intelligent RTS can optimize the power consumption of the network by turning off the unused links, with minimal hardware support.

- Process variation causes performance and power heterogeneity among various cores of a many-core chip. The runtime system needs to choose a configuration among many that meets the power and performance constraints. Therefore, variation is a major challenge for power efficiency.

- We studied various performance and power models for many-core chips with process variation. We demonstrated that an accurate model should consider both the number of active cores and the frequency of the active cores.

- Based on the variation-aware performance and power models, we developed a novel scheduling framework that uses integer linear programming (ILP) to explore the search space efficiently.

- Our variation-aware scheduling framework enables intelligent HPC runtime systems to enforce different performance and power constraints and schedule work units effectively. We also illustrated how different constraints such communication performance can be enforced in our framework.

7.1 Future Research Directions

There are many directions for future work based on the insights of this dissertation. Below is a list of example future directions.

- Analysis of various modern HPC architectures with regards to performance, programmability, power efficiency, and energy efficiency tradeoffs is always insightful. The set of HPC applications and benchmarks used should be chosen carefully to draw valid conclusions.

- Heterogeneous architectures have various challenges that need to explored further. Productive programming and resource management are the main issues discussed in this dissertation and need more research. Novel programming languages features are needed to make it possible to have the same or similar code for different devices. The resulting program should have high performance on different devices as well. On the other hand, automatic RTS techniques are needed to map the program on different devices effectively.

- Our results show that the memory hierarchy can be very wasteful for HPC applications. Thus, more research on memory hierarchy design for HPC applications is needed. For example, scratch-pads can alleviate some of the power efficiency challenges but make programming more difficult.

- We demonstrated that networks are over-provisioned and not energy proportional for HPC machines. Therefore, more research is needed to make them more efficient. For example, in addition to links, our techniques can be used to adapt the resources of the routers to consume less power as well.

- We applied our RTS-based adaptive framework to caches and network links, but it can be applied to other system components as well. For example, more opportunities for adaptation in processors that reduce instruction scheduling overheads need to be explored.

- Process variation causes various heterogeneity challenges. More in depth evaluation and analysis of different HPC applications on heterogeneous chips needs to be performed. Furthermore, the use of our scheduling framework in Chapter 6 to find different configurations for different application phases needs to be studied. In addition, we assumed that tasks of the application have the same amount of work but this is not always the case. Variation-aware load balancing of complex applications that have different workloads in different tasks needs further exploration. Moreover, the development and evaluation of new scheduling constraints dependent on the application and system characteristics requires much research.

- Our scheduling framework can find the optimal configuration given a power budget. However, the power budget has to be chosen carefully, in order to utilize the system efficiently. Future research is needed for addressing this problem, especially for overprovisioned HPC data centers.

REFERENCES

[1] H. Esmaeilzadeh, E. Blem, R. St.Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, June 2011, pp. 365–376.

[2] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller et al., "Exascale computing study: Technology challenges in achieving exascale systems," *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, 2008.

[3] N. Carter, A. Agrawal, S. Borkar, R. Cledat, H. David, D. Dunning, J. Fryman, I. Ganev, R. Golliver, R. Knauerhase, R. Lethin, B. Meister, A. Mishra, W. Pinfold, J. Teller, J. Torrellas, N. Vasilache, G. Venkatesh, and J. Xu, "Runnemede: An architecture for ubiquitous high-performance computing," in *High Performance Computer Architecture (HPCA 2013), IEEE 19th International Symposium on*, Feb 2013, pp. 198–209.

[4] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "Gpus and the future of parallel computing," *IEEE Micro*, vol. 31, no. 5, pp. 7–17, 2011.

[5] J. Torrellas, "Extreme-scale computer architecture: Energy efficiency from the ground up," in *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2014.

[6] U. R. Karpuzcu, A. Sinkar, N. S. Kim, and J. Torrellas, "EnergySmart: Toward energy-efficient manycores for near-threshold computing," in *High Performance Computer Architecture (HPCA)*, 2013.

[7] H. Kaul, M. Anders, S. Hsu, A. Agarwal, R. Krishnamurthy, and S. Borkar, "Near-threshold voltage (ntv) design: Opportunities and challenges," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12, 2012.

[8] A. Faraj, S. Kumar, B. Smith, A. Mamidala, J. Gunnels, and P. Heidelberger, "Mpi collective communications on the blue gene/p supercomputer: algorithms and optimizations," in *Proceedings of the 23rd international conference on Supercomputing*, ser. ICS '09, 2009, pp. 489–490.

[9] P. Corbett, D. Feitelson, S. Fineberg, Y. Hsu, B. Nitzberg, J.-P. Prost, M. Snirt, B. Traversat, and P. Wong, "Overview of the MPI-IO Parallel I/O interface," in *Input/Output in Parallel and Distributed Computer Systems*. Springer US, 1996, vol. 362, pp. 127–146.

[10] G. Zheng, A. Bhatele, E. Meneses, and L. V. Kale, "Periodic Hierarchical Load Balancing for Large Supercomputers," *International Journal of High Performance Computing Applications (IJHPCA), 2011*, March 2011.

[11] E. Meneses, "Scalable message-logging techniques for effective fault tolerance in HPC applications," Ph.D. dissertation, Dept. of Computer Science, University of Illinois, 2013.

[12] O. Sarood, P. Miller, E. Totoni, and L. V. Kale, "'Cool' Load Balancing for High Performance Computing Data Centers," in *IEEE Transactions on Computer - SI (Energy Efficient Computing)*, September 2012.

[13] O. Sarood, A. Langer, A. Gupta, and L. V. Kale, "Maximizing throughput of overprovisioned hpc data centers under a strict power budget," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. New York, NY, USA: ACM, 2014.

[14] P. Hammarlund, A. Martinez, A. Bajwa, D. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R. 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," *Micro, IEEE*, vol. 34, no. 2, pp. 6–20, Mar 2014.

[15] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe, "The 48-core SCC processor: The programmer's view," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010, pp. 1–11.

[16] L. V. Kale and G. Zheng, "Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects," in *Advanced Computational Infrastructures for Parallel and Distributed Applications*, M. Parashar, Ed. Wiley-Interscience, 2009, pp. 265–282.

[17] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson, "A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS," in *Solid-State Circuits Conference Digest of Technical Papers*, 2010, pp. 108–109.

[18] R. F. van der Wijngaart, T. G. Mattson, and W. Haas, "Light-weight communications on Intel's single-chip cloud computer processor," *SIGOPS Oper. Syst. Rev.*, vol. 45, pp. 73–83, February 2011.

[19] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kalé, and K. Schulten, "Scalable molecular dynamics with NAMD," *Journal of Computational Chemistry*, vol. 26, no. 16, pp. 1781–1802, 2005.

[20] D. Bailey, E. Barszcz, L. Dagum, and H. Simon, "NAS parallel benchmark results," in *Proc. Supercomputing*, Nov. 1992.

[21] L. V. Kale, G. Zheng, C. W. Lee, and S. Kumar, "Scaling applications to massively parallel machines using projections performance analysis tool," in *Future Generation Computer Systems Special Issue on: Large-Scale System Performance Modeling and Analysis*, vol. 22, no. 3, February 2006, pp. 347–358.

[22] B. Marker, E. Chan, J. Poulson, R. van de Geijn, R. F. Van der Wijngaart, T. G. Mattson, and T. E. Kubaska, "Programming many-core architectures - a case study: Dense matrix computations on the Intel Single-chip Cloud Computer processor," *Concurrency and Computation: Practice and Experience*, 2011.

[23] R. David, P. Bogdan, R. Marculescu, and U. Ogras, "Dynamic power management of voltage-frequency island partitioned networks-on-chip using Intel Single-chip Cloud Computer," in *International Symposium on Networks-on-Chip*, 2011, pp. 257–258.

[24] P. Salihundam, S. Jain, T. Jacob, S. Kumar, V. Erraguntla, Y. Hoskote, S. Vangal, G. Ruhl, and N. Borkar, "A 2 Tb/s 6×4 mesh network for a single-chip cloud computer with DVFS in 45 nm CMOS," *IEEE Journal of Solid-State Circuits*, vol. 46, no. 4, pp. 757 –766, April 2011.

[25] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl, "Evaluation and improvements of programming models for the Intel SCC many-core processor," in *International Conference on High Performance Computing and Simulation (HPCS)*, 2011, pp. 525–532.

[26] I. Ureña, M. Riepen, and M. Konow, "RCKMPI–Lightweight MPI implementation for Intels Single-chip Cloud Computer (SCC)," in *Recent Advances in the Message Passing Interface: 18th European MPI Users Group Meeting*. Springer-Verlag New York Inc, 2011, p. 208.

[27] C. Clauss, S. Lankes, and T. Bemmerl, "Performance tuning of SCC-MPICH by means of the proposed MPI-3.0 tool interface," *Recent Advances in the Message Passing Interface*, pp. 318–320, 2011.

[28] H. Esmaeilzadeh, T. Cao, Y. Xi, S. M. Blackburn, and K. S. McKinley, "Looking back on the language and hardware revolutions: Measured power, performance, and scaling," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011, pp. 319–332.

[29] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan, "Heterogeneous chip multiprocessors," *Computer*, vol. 38, no. 11, Nov. 2005.

[30] S. Damaraju, V. George, S. Jahagirdar, T. Khondker, R. Milstrey, S. Sarkar, S. Siers, I. Stolero, and A. Subbiah, "A 22nm ia multi-cpu and gpu system-on-chip," vol. 55, 2012, pp. 56–57.

[31] D. Foley, P. Bansal, D. Cherepacha, R. Wasmuth, A. Gunasekar, S. Gutta, and A. Naini, "A low-power integrated x86-64 and graphics processor for mobile computing devices," *IEEE Journal of Solid-State Circuits*, vol. 47, no. 1, pp. 220–231, 2012.

[32] NVIDIA, "Bringing High-End Graphics to Handheld Devices," 2011. [Online]. Available: http://www.nvidia.com

[33] M. Dikmen, D. Hoiem, and T. S. Huang, "A data driven method for feature transformation," in *Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2012, pp. 3314–3321.

[34] T. Mudge, "Power: A first-class architectural design constraint," *Computer*, vol. 34, no. 4, pp. 52–58, Apr. 2001.

[35] C. Beleznai, D. Schreiber, and M. Rauter, "Pedestrian detection using gpu-accelerated multiple cue computation," in *Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2011, pp. 58–65.

[36] L. Zhang and R. Nevatia, "Efficient scan-window based object detection using gpgpu," in *Computer Vision and Pattern Recognition Workshops. CVPRW '08*, 2008, pp. 1–7.

[37] M. Jones and P. Viola, "Fast multi-view face detection," *Mitsubishi Electric Research Lab TR-20003-96*, vol. 3, 2003.

[38] P. F. Felzenszwalb, R. B. Girshick, D. McAllester, and D. Ramanan, "Object detection with discriminatively trained part based models," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 32, no. 9, pp. 1627–1645, 2010.

[39] S. Maji and J. Malik, "Object detection using a max-margin hough transform," in *Computer Vision and Pattern Recognition*. IEEE, 2009, pp. 1038–1045.

[40] T. Ma and L. J. Latecki, "From partial shape matching through local deformation to robust global shape similarity for object detection," in *Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2011, pp. 1441–1448.

[41] C. J. Burges, "A tutorial on support vector machines for pattern recognition," *Data mining and knowledge discovery*, vol. 2, no. 2, pp. 121–167, 1998.

[42] M. Dikmen, H. Ning, D. J. Lin, L. Cao, V. Le, S.-F. Tsai, K.-H. Lin, Z. Li, J. Yang, T. S. Huang et al., "Surveillance event detection," in *TrecVID Video Evaluation Workshop*, 2008.

[43] M. Yang, S. Ji, W. Xu, J. Wang, F. Lv, K. Yu, Y. Gong, M. Dikmen, D. J. Lin, and T. S. Huang, "Detecting human actions in surveillance videos," in *TrecVID Video Evaluation Workshop*, 2009.

[44] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *Micro, IEEE*, vol. 28, no. 2, pp. 39–55, March-April 2008.

[45] Y. Zhang, Y. Hu, B. Li, and L. Peng, "Performance and power analysis of ati gpu: A statistical approach," in *Networking, Architecture and Storage (NAS)*, July 2011, pp. 149–158.

[46] "Intel SK for OpenCL Applications 2013," http://software.intel.com/en-us/vcsource/tools/opencl-sdk, 2013.

[47] "OpenCL Optimization Guide," http://software.intel.com/sites/products/documentation/ioclsdk/2013/OG/index.htm, 2013.

[48] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "Rapl: Memory power estimation and capping," in *Proc. of ISPLED*, 2010.

[49] E. Rotem, A. Naveh, D. Rajwan, A. Ananthakrishnan, and E. Weissmann, "Power-management architecture of the intel microarchitecture code-named sandy bridge," *Micro, IEEE*, vol. 32, no. 2, pp. 20–27, March-April 2012.

[50] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu," in *Proc. of ISCA*, 2010, pp. 451–460.

[51] P. Thoman, K. Kofler, H. Studt, J. Thomson, and T. Fahringer, "Automatic opencl device characterization: guiding optimized kernel design," in *Euro-Par 2011*, 2011, pp. 438–452.

[52] "Avoiding AVX-SSE Transition Penalties," http://software.intel.com/en-us/articles/intel-avx-state-transitions-migrating-sse-code-to-avx, 2011.

[53] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[54] J. Reinders, *Intel threading building blocks*, 1st ed., 2007.

[55] M. H. Halstead, *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.

[56] E. J. Weyuker, "Evaluating software complexity measures," *Software Engineering, IEEE Transactions on*, vol. 14, no. 9, pp. 1357–1365, 1988.

[57] C. H. González and B. B. Fraguela, "A generic algorithm template for divide-and-conquer in multicore systems," in *High Performance Computing and Communications (HPCC)*. IEEE, 2010, pp. 79–88.

[58] G. Ren, P. Wu, and D. Padua, "Optimizing data permutations for simd devices," in *Proc. of PLDI*, 2006, pp. 118–131.

[59] S. Maleki, Y. Gao, M. J. Garzarán, T. Wong, and D. A. Padua, "An evaluation of vectorizing compilers," in *Proc. of PACT*, 2011, pp. 372–382.

[60] J. Corbal, R. Espasa, and M. Valero, "On the efficiency of reductions in μ-simd media extensions," in *Proc. of PACT*, pp. 83–94.

[61] D. Talla, L. John, and D. Burger, "Bottlenecks in multimedia processing with simd style extensions and architectural enhancements," *Computers, IEEE Transactions on*, vol. 52, no. 8, pp. 1015–1031, Aug. 2003.

[62] A. J. C. Bik, X. Tian, and M. B. Girkar, "Multimedia vectorization of floating-point min/max reductions," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 9, pp. 997–1007, 2006.

[63] V. Prisacariu and I. Reid, "fasthog-a real-time gpu implementation of hog," *University of Oxford Technical Report*, vol. 2310, no. 09, 2009.

[64] R. Dreslinski, M. Wieckowski, D. Blaauw, D. Sylvester, and T. Mudge, "Near-threshold computing: Reclaiming moore's law through energy efficient integrated circuits," *Proceedings of the IEEE*, vol. 98, no. 2, pp. 253–266, 2010.

[65] C.-K. Luk, S. Hong, and H. Kim, "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *Proc. of MICRO*, 2009.

[66] C. Liu, J. Li, W. Huang, J. Rubio, E. Speight, and X. Lin, "Power-efficient time-sensitive mapping in heterogeneous systems," in *Proc. of PACT*, 2012.

[67] V. J. Jiménez, L. Vilanova, I. Gelado, M. Gil, G. Fursin, and N. Navarro, "Predictive runtime code scheduling for heterogeneous architectures," in *Proc. of HiPEAC*, 2009.

[68] K. Ma, X. Li, W. Chen, C. Zhang, and X. Wang, "Greengpu: A holistic approach to energy efficiency in gpu-cpu heterogeneous architectures," in *Proc. of ICPP*, 2012.

[69] T. Cao, S. M. Blackburn, T. Gao, and K. S. McKinley, "The yin and yang of power and performance for asymmetric hardware and managed software," in *Proc. of ISCA*, 2012.

[70] J. Planas, R. M. Badia, E. Ayguade, and J. Labarta, "Self-adaptive ompss tasks in heterogeneous environments," in *Proc. of IPDPS*, 2013.

[71] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier, "Starpu: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurr. Comput. : Pract. Exper.*, vol. 23, no. 2, pp. 187–198, Feb. 2011.

[72] M. Doerksen, P. Thulasiraman, and R. Thulasiram, "Optimizing option pricing algorithms and profiling power consumption on vliw apu architecture," 2012, pp. 71–78.

[73] A. Rattanatranurak, S. Kittitornkun, and S. Tongsima, "Optimizing and multithreading snphap on a multi-core apu with opencl," 2012, pp. 174–179.

[74] K. Spafford, J. Meredith, S. Lee, D. Li, P. Roth, and J. Vetter, "The tradeoffs of fused memory hierarchies in heterogeneous computing architectures," 2012, pp. 103–112.

[75] M. Daga, A. Aji, and W.-C. Feng, "On the efficacy of a fused cpu+gpu processor (or apu) for parallel computing," 2011, pp. 141–149.

[76] Y. Allusse, P. Horain, A. Agarwal, and C. Saipriyadarshan, "Gpucv: A gpu-accelerated framework for image processing and computer vision," in *Advances in Visual Computing*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, vol. 5359, pp. 430–439.

[77] P. Babenko and M. Shah, "Mingpu: A minimum gpu library for computer vision," *Journal of Real-Time Image Processing*, vol. 3, no. 4, pp. 255–268, 2008.

[78] J. Fung and S. Mann, "Using graphics devices in reverse: Gpu-based image processing and computer vision," 2008, pp. 9–12.

[79] P. Mistry, C. Gregg, N. Rubin, D. Kaeli, and K. Hazelwood, "Analyzing program flow within a many-kernel opencl application," in *Proc. of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-4, 2011.

[80] Y. Sharrab and N. Sarhan, "Accuracy and power consumption tradeoffs in video rate adaptation for computer vision applications," in *Proc of Multimedia and Expo (ICME)*, 2012, pp. 410–415.

[81] J. Bergman, "Energy efficient graphics: Making the rendring process power aware," Ph.D. dissertation, Uppsala University, 2010.

[82] C. Zhang and Z. Zhang, "A survey of recent advances in face detection," Tech. Rep. MSR-TR-2010-66, 2010.

[83] S. G. Kong, J. Heo, B. R. Abidi, J. Paik, and M. A. Abidi, "Recent advances in visual and infrared face recognition a review," *Computer Vision and Image Understanding*, vol. 97, no. 1, pp. 103–135, 2005.

[84] P. Dollar, C. Wojek, B. Schiele, and P. Perona, "Pedestrian detection: An evaluation of the state of the art," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 34, no. 4, pp. 743–761, 2012.

[85] V. Zyuban, J. Friedrich, C. J. Gonzalez, R. Rao, M. D. Brown, M. Ziegler, H. Jacobson, S. Islam, S. Chu, P. Kartschoke, G. Fiorenza, M. Boersma, and J. Culp, "Power optimization methodology for the IBM POWER7 microprocessor," *IBM Journal of Research and Development*, vol. 55, no. 3, 2011.

[86] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving performance via mini-applications," Sandia National Laboratories, Tech. Rep. SAND2009-5574, 2009.

[87] M. Collaboration, "MIMD Lattice Computation (MILC) Collaboration Home Page," http://www.physics.indiana.edu/~sg/milc.html.

[88] M. M. Tikir, L. Carrington, E. Strohmaier, and A. Snavely, "A genetic algorithms approach to modeling the performance of memory-bound computations," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC'07)*, 2007.

[89] R. Cheveresan, M. Ramsay, C. Feucht, and I. Sharapov, "Characteristics of workloads used in high performance and technical computing," in *Proceedings of the 21st Annual International Conference on Supercomputing (ICS'07)*, 2007.

[90] J. Weinberg, M. O. McCracken, E. Strohmaier, and A. Snavely, "Quantifying locality in the memory access patterns of HPC applications," in *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing (SC'05)*, 2005.

[91] R. Murphy and P. Kogge, "On the memory access patterns of supercomputer applications: Benchmark selection and its implications," *Computers, IEEE Transactions on*, vol. 56, no. 7, pp. 937–945, 2007.

[92] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, "Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures," in *ACM/IEEE International Symposium on Microarchitecture (MICRO 33)*, 2000.

[93] A. Dhodapkar and J. Smith, "Managing multi-configuration hardware via dynamic working set analysis," in *International Symposium on Computer Architecture (ISCA)*, 2002.

[94] A. Gordon-Ross, J. Lau, and B. Calder, "Phase-based cache reconfiguration for a highly-configurable two-level cache hierarchy," in *Proceedings of the 18th ACM Great Lakes Symposium on VLSI (GLSVLSI '08)*, 2008.

[95] P. Ranganathan, S. Adve, and N. Jouppi, "Reconfigurable caches and their application to media processing," in *International Symposium on Computer Architecture (ISCA)*, 2000.

[96] S. Mittal, Y. Cao, and Z. Zhang, "Master: A multicore cache energy-saving technique using dynamic cache reconfiguration," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. PP, no. 99, 2013.

[97] S. Srikantaiah, E. Kultursay, T. Zhang, M. Kandemir, M. Irwin, and Y. Xie, "Morphcache: A reconfigurable adaptive multi-level cache hierarchy," in *High Performance Computer Architecture (HPCA)*, 2011.

[98] W. Zang and A. Gordon-Ross, "A survey on cache tuning from a power/energy perspective," *ACM Comput. Surv.*, vol. 45, no. 3, pp. 32:1–32:49, July 2013.

[99] H. Hoffmann, J. Holt, G. Kurian, E. Lau, M. Maggio, J. Miller, S. Neuman, M. Sinangil, Y. Sinangil, A. Agarwal, A. Chandrakasan, and S. Devadas, "Self-aware computing in the Angstrom processor," in *Design Automation Conference (DAC)*, 2012.

[100] J. Hu, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, "Analyzing data reuse for cache reconfiguration," *ACM Trans. Embed. Comput. Syst.*, vol. 4, no. 4, pp. 851–876, Nov. 2005.

[101] S. Tavarageri and P. Sadayappan, "A compiler analysis to determine useful cache size for energy efficiency," in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW)*, 2013.

[102] R. Cheveresan, M. Ramsay, C. Feucht, and I. Sharapov, "Characteristics of workloads used in high performance and technical computing," in *Proceedings of the 21st Annual International Conference on Supercomputing (ICS'07)*, 2007.

[103] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC'08)*, 2008.

[104] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. B. Fraguela, M. J. Garzarán, D. Padua, and C. Von Praun, "Programming for parallelism and locality with hierarchically tiled arrays," in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2006, pp. 48–57.

[105] B. Behzad, H. V. T. Luu, J. Huchette, S. Byna, Prabhat, R. Aydt, Q. Koziol, and M. Snir, "Taming parallel I/O complexity with auto-tuning," in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC'13)*, 2013.

[106] I. Dooley, "Intelligent runtime tuning of parallel applications with control points," Ph.D. dissertation, Dept. of Computer Science, University of Illinois, 2010, http://charm.cs.uiuc.edu/papers/DooleyPhDThesis10.shtml.

[107] "Intel Product Information," 2014. [Online]. Available: http://ark.intel.com/

[108] "IBM Product Information," 2014. [Online]. Available: http://www.ibm.com/

[109] M.-T. Chang, P. Rosenfeld, S.-L. Lu, and B. Jacob, "Technology comparison for large last-level caches (L3Cs): Low-leakage SRAM, low write-energy STT-RAM, and refresh-optimized eDRAM," in *High Performance Computer Architecture (HPCA)*, 2013.

[110] D. H. Albonesi, "Selective cache ways: On-demand cache resource allocation," in *Proceedings of the 32Nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 32)*, 1999.

[111] I. Corp., "Intel 64 and IA-32 architectures software developer manual," 2013.

[112] S. Palacharla and R. E. Kessler, "Evaluating stream buffers as a secondary cache replacement," in *ACM SIGARCH Computer Architecture News*, vol. 22, no. 2. IEEE Computer Society Press, 1994, pp. 24–33.

[113] X. Liu and J. Mellor-Crummey, "A data-centric profiler for parallel programs," in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC'13)*, 2013.

[114] E. M. Gold, "Language identification in the limit," *Information and control*, vol. 10, no. 5, pp. 447–474, 1967.

[115] C. De la Higuera, *Grammatical inference: learning automata and grammars*. Cambridge University Press, 2010.

[116] H. Fernau, "Learning tree languages from text," in *Computational Learning Theory*. Springer, 2002, pp. 153–168.

[117] E. Vidal, F. Thollard, C. De La Higuera, F. Casacuberta, and R. C. Carrasco, "Probabilistic finite-state machines," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 27, no. 7, pp. 1013–1025, 2005.

[118] S. R. Eddy, "Hidden markov models," *Current opinion in structural biology*, vol. 6, no. 3, pp. 361–365, 1996.

[119] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, and L. Kale, "Parallel Programming with Migratable Objects: Charm++ in Practice," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. New York, NY, USA: ACM, 2014.

[120] L. Kale, A. Arya, N. Jain, A. Langer, J. Lifflander, H. Menon, X. Ni, Y. Sun, E. Totoni, R. Venkataraman, and L. Wesolowski, "Migratable Objects + Active Messages + Adaptive Runtime = Productivity + Performance A Submission to 2012 HPC Class II Challenge," Parallel Programming Laboratory, Tech. Rep. 12-47, November 2012.

[121] J. Renau et al., "SESC: SuperESCalar simulator," 2005.

[122] "CACTI: an integrated cache and memory access time, cycle time, area, leakage, and dynamic power model," 2013. [Online]. Available: http://http://www.hpl.hp.com/research/cacti/

[123] P. Cicotti, L. Carrington, and A. Chien, "Toward application-specific memory reconfiguration for energy efficiency," in *Workshop on Energy Efficient Supercomputing (E2SC'13)*, 2013.

[124] E. Totoni, B. Behzad, S. Ghike, and J. Torrellas, "Comparing the power and performance of intel's scc to state-of-the-art cpus and gpus," in *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on*, 2012, pp. 78–87.

[125] B. Arimilli, R. Arimilli, V. Chung, S. Clark, W. Denzel, B. Drerup, T. Hoefler, J. Joyner, J. Lewis, J. Li, N. Ni, and R. Rajamony, "The PERCS High-Performance Interconnect," in *2010 IEEE 18th Annual Symposium on High Performance Interconnects (HOTI)*, August 2010, pp. 75–82.

[126] A. Bhatele, N. Jain, W. D. Gropp, and L. V. Kale, "Avoiding hot-spots on two-level direct networks," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 76:1–76:11.

[127] G. Faanes, A. Bataineh, D. Roweth, T. Court, E. Froese, B. Alverson, T. Johnson, J. Kopnick, M. Higgins, and J. Reinhard, "Cray cascade: a scalable HPC system based on a Dragonfly network," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389136 pp. 103:1–103:9.

[128] Y. Ajima, Y. Takagi, T. Inoue, S. Hiramoto, and T. Shimizu, "The Tofu interconnect," in *High Performance Interconnects (HOTI), 2011 IEEE 19th Annual Symposium on*, aug. 2011, pp. 87–94.

[129] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick, "Exascale computing study: Technology challenges in achieving exascale systems," 2008.

[130] L. Shang, L.-S. Peh, and N. Jha, "Dynamic voltage scaling with links for power optimization of interconnection networks," in *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*, feb. 2003, pp. 91–102.

[131] V. Soteriou and L.-S. Peh, "Exploring the design space of self-regulating power-aware on/off interconnection networks," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 18, no. 3, pp. 393–408, march 2007.

[132] J. Laros, K. Pedretti, S. Kelly, W. Shu, and C. Vaughan, "Energy based performance tuning for large scale high performance computing systems," in *Proceedings of 20th High Performance Computing Symposium*, ser. HPC, 2012.

[133] P. Kogge, "Architectural challenges at the exascale frontier (invited talk)," *Simulating the Future: Using One Million Cores and Beyond*, 2008.

[134] P. Mahadevan, P. Sharma, S. Banerjee, and P. Ranganathan, "Energy aware network operations," in *INFOCOM Workshops 2009, IEEE*, april 2009, pp. 1–6.

[135] D. Abts, M. R. Marty, P. M. Wells, P. Klausler, and H. Liu, "Energy proportional datacenter networks," in *Proceedings of the 37th annual international symposium on computer architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1815961.1816004 pp. 338–347.

[136] L. V. Kale, A. Bhatele, E. J. Bohm, and J. C. Phillips, "NAnoscale Molecular Dynamics (NAMD)," in *Encyclopedia of Parallel Computing*, D. Padua, Ed. Springer Verlag, 2011.

[137] C. Bernard, T. Burch, T. A. DeGrand, C. DeTar, S. Gottlieb, U. M. Heller, J. E. Hetrick, K. Orginos, B. Sugar, and D. Toussaint, "Scaling tests of the improved Kogut-Susskind quark action," *Physical Review D*, no. 61, 2000.

[138] A. Jain and X. Yang, "Modeling the effects of two different land cover change data sets on the carbon stocks of plants and soils in concert with CO2 and climate change," *Global Biogeochem. Cycles*, vol. 19, no. 2, pp. 1–20, 2005.

[139] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown, "ElasticTree: saving energy in data center networks," in *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, 2010.

[140] M. Alonso, S. Coll, J.-M. Martínez, V. Santonja, P. López, and J. Duato, "Dynamic power saving in fat-tree interconnection networks using on/off links," in *Proceedings of the 20th international conference on Parallel and distributed processing*, ser. IPDPS'06. Washington, DC, USA: IEEE Computer Society, 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=1898699.1898826 pp. 299–299.

[141] J. Li, W. Huang, C. Lefurgy, L. Zhang, W. Denzel, R. Treumann, and K. Wang, "Power shifting in thrifty interconnection network," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, feb. 2011, pp. 156 –167.

[142] V. Soteriou, N. Eisley, and L.-S. Peh, "Software-directed power-aware interconnection networks," *ACM Trans. Archit. Code Optim.*, vol. 4, no. 1, Mar. 2007.

[143] A. Becker, "Compiler support for productive message-driven parallel programming," Ph.D. dissertation, Dept. of Computer Science, University of Illinois, 2012, http://charm.cs.uiuc.edu/media/12-44/.

[144] G. Hendry, "Decreasing network power with on-off links informed by scientific applications." in *HPPAC'13*, 2013.

[145] S. Conner, S. Akioka, M. Irwin, and P. Raghavan, "Link shutdown opportunities during collective communications in 3-D torus nets," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, march 2007, pp. 1 –8.

[146] "Top500 supercomputing sites," http://top500.org, 2013.

[147] J. Kim, W. J. Dally, S. Scott, and D. Abts, "Technology-driven, highly-scalable dragonfly topology," *SIGARCH Comput. Archit. News*, vol. 36, pp. 77–88, June 2008.

[148] E. Totoni, A. Bhatele, E. Bohm, N. Jain, C. Mendes, R. Mokos, G. Zheng, and L. Kale, "Simulation-based performance analysis and tuning for a two-level directly connected system," in *Proceedings of the 17th IEEE International Conference on Parallel and Distributed Systems*, December 2011.

[149] G. Zheng, G. Kakulapati, and L. V. Kalé, "Bigsim: A parallel simulator for performance prediction of extremely large parallel machines," in *18th International Parallel and Distributed Processing Symposium (IPDPS)*, Santa Fe, New Mexico, April 2004, p. 78.

[150] S. Kumar, Y. Sun, and L. V. Kale, "Acceleration of an Asynchronous Message Driven Programming Paradigm on IBM Blue Gene/Q," in *Proceedings of 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Boston, USA, May 2013.

[151] A. Bhatele, E. Bohm, and L. V. Kale, "Optimizing communication for charm++ applications by reducing network contention," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 211–222, 2011.

[152] A. Langer, J. Lifflander, P. Miller, K.-C. Pan, L. V. Kale, and P. Ricker, "A Scalable Mesh Restructuring Algorithm for Distributed-Memory Adaptive Mesh Refinement," in *Proceedings of 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2012.

[153] A. Bhatelé and L. V. Kalé, "Quantifying Network Contention on Large Parallel Machines," *Parallel Processing Letters (Special Issue on Large-Scale Parallel Processing)*, vol. 19, no. 4, pp. 553–572, 2009.

[154] D. Kerbyson, K. Barker, A. Vishnu, and A. Hoisie, "Comparing the performance of blue gene/q with leading cray xe6 and infiniband systems," in *Parallel and Distributed Systems (ICPADS), IEEE 18th International Conference on*, 2012, pp. 556–563.

[155] D. Chen, N. Eisley, P. Heidelberger, R. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. Satterfield, B. Steinmacher-Burow, and J. Parker, "The ibm blue gene/q interconnection network and message unit," in *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, 2011, pp. 1–10.

[156] Megan Gilge, "Blue Gene/Q Application Development," http://www.redbooks.ibm.com/abstracts/sg247948.html, 2013.

[157] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De, "Parameter variations and impact on circuits and microarchitecture," in *Proceedings of the 40th Annual Design Automation Conference (DAC '03)*, 2003.

[158] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT press, 1999, vol. 1.

[159] B. Lewis and D. J. Berg, *Multithreaded Programming with Pthreads*. Prentice-Hall, Inc., 1998.

[160] R. Teodorescu and J. Torrellas, "Variation-aware application scheduling and power management for chip multiprocessors," in *Computer Architecture, 2008. ISCA '08. 35th International Symposium on*, 2008.

[161] J. A. Winter, D. H. Albonesi, and C. A. Shoemaker, "Scalable thread scheduling and global power management for heterogeneous many-core architectures," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '10, 2010.

[162] J. L. Gustafson, "Reevaluating amdahl's law," *Commun. ACM*, vol. 31, no. 5, pp. 532–533, May 1988. [Online]. Available: http://doi.acm.org/10.1145/42411.42415

[163] A. B. Downey, "A parallel workload model and its implications for processor allocation," *Cluster Computing*, vol. 1, no. 1, pp. 133–145, 1998.

[164] A. Lastovetsky and R. Reddy, "On performance analysis of heterogeneous parallel algorithms," *Parallel Computing*, vol. 30, no. 11, pp. 1195 – 1216, 2004.

[165] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling heterogeneous multi-cores through performance impact estimation (pie)," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA '12, 2012, pp. 213–224.

[166] S. Dighe, S. Vangal, P. Aseron, S. Kumar, T. Jacob, K. Bowman, J. Howard, J. Tschanz, V. Erraguntla, N. Borkar, V. De, and S. Borkar, "Within-die variation-aware dynamic-voltage-frequency-scaling with optimal core allocation and thread hopping for the 80-core teraflops processor," *Solid-State Circuits, IEEE Journal of*, vol. 46, no. 1, pp. 184–193, Jan 2011.

[167] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11, 2011.

[168] U. R. Karpuzcu, K. B. Kolluru, N. S. Kim, and J. Torrellas, "VARIUS-NTV: A Microarchitectural Model to Capture the Increased Sensitivity of Manycores to Process Variations at Near-Threshold Voltages," in *International Conference on Dependable Systems and Networks*, June 2012.

[169] S. Li, J.-H. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009.

[170] H. Menon and L. Kalé, "A distributed dynamic load balancer for iterative applications," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 15.

[171] J. Kang and S. Park, "Algorithms for the variable sized bin packing problem," *European Journal of Operational Research*, vol. 147, no. 2, pp. 365 – 372, 2003.

[172] Y. Sun, G. Zheng, C. M. E. J. Bohm, T. Jones, L. V. Kalé, and J. C. Phillips, "Optimizing fine-grained communication in a biomolecular simulation application on Cray XK6," in *Proceedings of the 2012 ACM/IEEE conference on Supercomputing*, Salt Lake City, Utah, November 2012.

[173] "Gurobi Optimization Inc. Software, 2012," http://www.gurobi.com/.

[174] K. Kuhn, M. Giles, D. Becher, P. Kolar, A. Kornfeld, R. Kotlyar, S. Ma, A. Maheshwari, and S. Mudanai, "Process technology variation," *Electron Devices, IEEE Transactions on*, vol. 58, no. 8, pp. 2197–2208, Aug 2011.

[175] S. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *Micro, IEEE*, vol. 25, no. 6, pp. 10–16, Nov 2005.

[176] O. Sarood, A. Langer, A. Gupta, and L. V. Kale, "Maximizing Throughput of a Data Center Under a Strict Power Budget," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. New York, NY, USA: ACM, 2014.

[177] T. Patki, D. K. Lowenthal, B. Rountree, M. Schulz, and B. R. de Supinski, "Exploring hardware overprovisioning in power-constrained, high performance computing," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13, 2013.