

Compiler Optimizations for Energy-Efficiency of Heterogeneous Computing Systems

(ヘテロジニアス計算機システムの電力効率向上のための
コンパイラ最適化手法)

薦田 登志矢

© 2014 - *Toshiya Komoda*
ALL RIGHTS RESERVED.

Abstract

Energy-efficiency has become one of the most important metric in recent computing systems. Power consumption of a large scale computing center is reaching to the physical limitation of power supply systems. Now, we have to scale system performance without increasing power consumption. To this end, heterogeneous systems with accelerators are becoming popular due to its high performance and its high energy efficiency. Peak performance and peak energy-efficiency of such systems are much higher than those of conventional homogeneous systems. However, it is not easy to achieve high energy-efficiency in practical situations because the actual energy efficiency highly depends on the optimizations used in the applications.

To get great performance from accelerators in practical situation, it is important to provide an easy programming environment to use accelerators. This is because the performance improvement is limited to a fraction of program codes which are not accelerated. To encourage programmers to utilize accelerators, programming environments should provide simple and high level abstraction of the underlying heterogeneous systems. However, current programming environments only provide low level and complicated APIs for users. Hence, we have to develop simpler and high performance programming environments in heterogeneous systems with accelerators.

Also, to draw the maximum energy-efficiency from the systems, it is also necessary to reduce unnecessary power consumption. For example, leakage power consumed in processors should be taken into account. Plus, it is not necessary to execute tasks which are not on critical paths with the highest processor frequency. In homogeneous systems with CPUs, such power consumption has been successfully reduced by power gating (PG) and dynamic voltage and frequency

scaling (DVFS). However, it is unclear that how we should maximize the effectiveness of these techniques in heterogeneous systems with accelerators.

To this end, in this dissertation, we propose three new optimization techniques for the accelerator compilers in order to solve these problems. The three techniques can be orthogonally applied to applications running on heterogeneous systems. And each technique independently contributes to improving the energy-efficiency of the system.

First, to ease the programming process in accelerator platforms, we propose a directive based compiler which utilizes multiple accelerators automatically. The compiler provides single monolithic memory space on top of discrete accelerator memories, and programmers can transparently utilize multiple accelerators. The experimental in a machine with 2 GPUs show that the proposed compiler reduce 68% of the code lines in utilizing 2 GPUs while it achieves 71% of the performance compared to hand-written CUDA programs

Second, to reduce leakage power in CPU functional units, we propose a compiler directed sleep control technique. Using a static analysis, the proposed sleep control technique can effectively reduce leakage power in CPU functional units. The experimental result shows that the proposed technique can reduce 23.6% of leakage power compared to a conventional sleep control.

Finally, to avoid inefficient executions due to load imbalance between CPUs and accelerators, we investigate a runtime software technique with which we can utilize both CPUs and accelerators in parallel. In addition to balancing load between CPUs and accelerators, the proposed technique can cooperatively control the device frequencies and the task mapping between CPUs and accelerators to further optimize energy-efficiency of the system. The experimental result shows that the proposed technique achieves up to 14.4% higher performance compared to other power capping techniques where the task mapping is not orchestrated with the device frequency settings.

Through these experiments about the three proposed techniques, we demonstrate that compiler optimization techniques can greatly help us achieve high energy-efficiency in heterogeneous systems with accelerators.

Contents

1	Introduction	1
1.1	Problems for Improving Energy-Efficiency	2
1.2	Contribution	4
1.2.1	Improving Programmability (Chapter 3)	5
1.2.2	Reducing Leakage Power (Chapter 4)	6
1.2.3	Reducing Dynamic Power (Chapter 5)	6
1.3	Organization of the Dissertation	7
2	Background	9
2.1	Heterogeneous System	9
2.1.1	GPU Computing Server	10
2.1.2	Other Examples of Heterogeneous Systems with Accelerators	10
2.2	Programming Model for Accelerators	12
2.2.1	CUDA and OpenCL	12
2.2.2	OpenACC	13
2.3	Power Management Technique	14
2.3.1	Power Gating	15
2.3.2	DVFS	16
2.3.3	Power Capping	17
3	Multi-Device Execution in a Directive Based Compiler	19
3.1	Drawbacks of Current GPU Programming Environments	19

3.2	OpenACC Compiler with Software Distributed Shared Memory	21
3.2.1	Directive Extensions	22
3.2.2	Implementation and Optimization in multi-GPU platforms	25
3.3	Evaluation	31
3.3.1	Performance and Programmability	32
3.3.2	Evaluation in a Supercomputer Node	37
4	Compiler Based Sleep Control in CPU Functional Units	41
4.1	Drawbacks of Conventional Sleep Control Techniques	41
4.2	Sleep Control Based On Precise Analysis of Idle Length	43
4.2.1	Architecture Support for Compiler Based Sleep Control	45
4.2.2	Code Analysis to Detect Fine-Grained Idle Periods	45
4.3	Evaluation	50
5	Cooperative DVFS and Heterogeneous Task Mapping	55
5.1	Drawbacks of Conventional Power Capping Techniques	55
5.2	Model-based DVFS and Task Mapping	57
5.2.1	Model Parameters and Profile Information	58
5.2.2	Empirical Models	60
5.2.3	Parameter Selection	62
5.3	Evaluation	63
5.3.1	Model Verification	64
5.3.2	Performance Under the Power Constraint	65
5.3.3	Sensitivity on Different Input Data	68
5.3.4	Optimizing for Energy-Efficiency	70
6	Discussion	75
6.1	Combination of the Proposed Techniques	76
6.2	Extension to Other Heterogeneous Systems	77

7	Related Work	81
7.1	GPU Programming Environment	81
7.2	Runtime Power Gating	83
7.3	Power Capping	83
8	Conclusion	85
	Acknowledgment	87
	Bibliography	89

List of Figures

1.1	Trends of power and energy-efficiency of supercomputers in Top500 (extracted from Green500 web site [1]).	2
2.1	Heterogeneous computing system.	10
2.2	An example of CPU-GPU heterogeneous systems.	11
2.3	CUDA vector addition.	13
2.4	OpenACC Vector Addition.	14
2.5	Power gated circuit and the power transition during sleep mode.	16
3.1	Comparison of current GPU programming environments.	20
3.2	Overview of the proposed multi-device OpenACC compiler.	22
3.3	Execution steps of parallel loops in the proposed system.	23
3.4	An simple example of the proposed directives.	24
3.5	An overview of the prototype system	26
3.6	A flowchart to determine the placement policy of each array.	29
3.7	Two-level dirty-bit mechanism to keep consistency of the replicated arrays.	30
3.8	Performance and lines of code (LOC). Normalized to the CUDA 2GPU versions.	33
3.9	Performance trends when the size of input data changes. Normalized to the CUDA 2GPU versions with base inputs.	34
3.10	Evaluations in TSUBAME2.0 thin-node (2CPU-3GPU).	39

4.1	Correlation between fraction of fine grained idle periods and WLR (BET=20 cycles).	43
4.2	An overview of the proposed sleep control system.	44
4.3	Architectural support for run-time PG in functional units.	45
4.4	Structures in programs	46
4.5	Normalized leakage power (BET=20cycle, FPALU).	52
4.6	Normalized leakage power (BET=40cycle, FPALU).	52
4.7	Normalized leakage power (BET=20cycle, FPMULT).	53
4.8	Normalized leakage power (BET=40cycle, FPMULT).	53
4.9	Normalized leakage power (BET=20cycle, INTMULT).	54
4.10	Normalized leakage power (BET=40cycle, INTMULT).	54
5.1	Hybrid computation with a CPU and a GPU.	56
5.2	Performance with conventional DVFS control. Normalized to the ideal execution without load imbalance.	57
5.3	An overview of the profile-based power capping.	58
5.4	Parameter space represented as a three-dimensional cube.	59
5.5	Fine-grained synchronization between the CPU and the accelerator.	63
5.6	Errors in the estimated execution time: GPU 614MHz.	66
5.7	Errors in the estimated power (watt): GPU 614MHz.	67
5.8	Performance under the power constraint. Normalized to the ideal performance for each power constraint.	69
5.9	Performance with data 2 times larger than profiled. Normalized to the ideal performance for each power constraint.	71
5.10	Performance for the data 4 times larger than profiled. Normalized to the ideal performance for each power constraint.	72
5.11	Normalized energy-efficiency.	73

List of Tables

3.1	Machine setup for the evaluation.	32
3.2	A: Total device memory usage in single GPU execution, B: # of parallel loops, C: # of kernel executions, D: # of arrays with localaccess directive / # of arrays used in parallel loops.	33
3.3	Number of code Lines.	37
4.1	Wasted leakage energy ratio(<i>WLR</i>) of conventional techniques (% , BET=20cycle).	43
4.2	Idle length prediction in a basic block.	46
4.3	Simulation setup.	51
4.4	Average leakage reduction from <i>timeout</i> sleep control (%).	52
5.1	Model parameters and profile information.	59
5.2	Machine setup for the evaluation.	64
5.3	Description of the benchmarks.	64
5.4	Average Errors of the Model (the percentage of GPU tasks: 50% - 90%).	65
5.5	Selected parameters in the parameter exploration for the highest energy-efficiency.	74

Chapter 1

Introduction

Energy-efficiency, which is given by performance per power, has become one of the most important metric in recent computing systems [17, 44]. The maximum power consumption of a large scale computing center is reaching to tens of Mega Watt, which is the physical limitation of the power supply and the cooling system. In Figure 1.1, pairs of the energy-efficiency (x-axis) and the power consumption (y-axis) of Top500 supercomputers are shown in a log scale scatter plot. Points of exascale systems are plotted on the top right corner of Figure 1.1. We can see that more than ten times improvement in energy-efficiency is required to build exascale computing systems.

To improve energy-efficiency of such computing systems, heterogeneous systems equipped with accelerators are becoming popular. Accelerators are designed to maximize performance and energy-efficiency for specific applications. Among such accelerators, GPUs, which is originally used for image processing applications, have become the center of attention. Previous studies have already shown that GPUs can boost performance and energy-efficiency of many practical applications, such as stencil computations, linear algebra, data mining, monte carlo simulations, graph processing and many others [26]. At the time of 2013, tens of supercomputers in Top500 systems are equipped with GPUs.

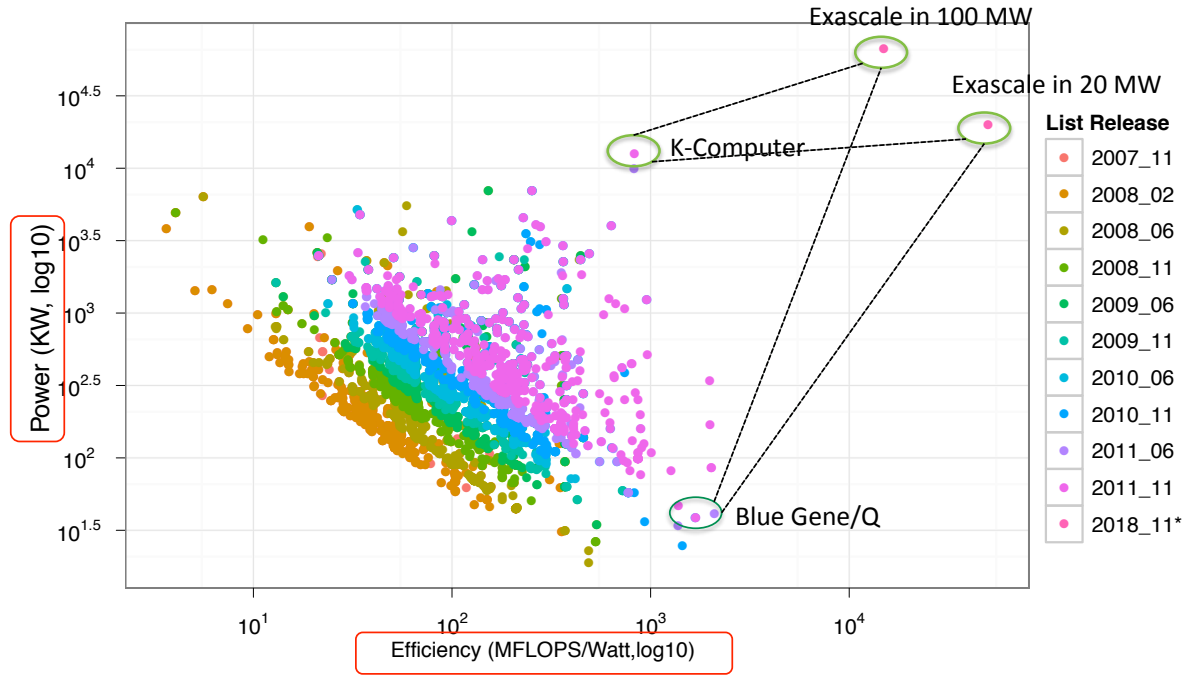


Figure 1.1: Trends of power and energy-efficiency of supercomputers in Top500 (extracted from Green500 web site [1]).

1.1 Problems for Improving Energy-Efficiency

Heterogeneous systems equipped with accelerators have much potential to provide higher performance and higher energy-efficiency than conventional homogeneous systems. However, because the hardware architecture of such systems is very different from conventional systems, conventional compilers cannot always draw the maximum energy efficiency from the systems.

Energy-efficiency of a computing system is defined by the following equations:

$$Eff = \frac{Perf}{Power}. \quad (1.1)$$

Here, Eff is the value of energy-efficiency of the system, $Perf$ is the average performance achieved by the applications running on the system and $Power$ is the average power consumption during the application execution.

In heterogeneous systems, we can offload data parallel tasks in the applications to accelerators in order to improve energy-efficiency. It can significantly improve the performance ($Perf$) with

small increase of the power (*Power*). The actual performance with accelerators can be explained by using Amdahl's law as shown in the following equation:

$$Perf = Perf_{CPUs} * \frac{1}{(1 - P) + \frac{P}{X}}. \quad (1.2)$$

Here, $Perf_{CPUs}$ is performance only with CPUs, P is the percentage of tasks offloaded to accelerators and X is a speedup ratio by utilizing accelerators.

In current accelerator platforms, typical values of the speedup ratio X is large, up to fifteen for data parallel applications [36]. Meanwhile, the equation 1.2 tells us that, even if the value of the speedup ratio X is large, it is important to increase the percentage of the tasks offloaded to accelerators. However, values of the percentage (P in the equation 1.2) tend to be low because the difficulty in optimizing programs for accelerators. In current accelerator programming environments, such as CUDA and OpenCL, low level architecture of accelerators are exposed to programmers. It significantly decreases the programmability in using accelerators and prevents programmers to utilize accelerators in many cases. To increase the percentage of offloaded tasks in practical situations, it is necessary to improve the accelerator's programmability.

Also, to improve the energy-efficiency, it is useful to reduce power consumption in processors if we can do that with small performance degradation. Even in homogeneous systems, more than half of power consumption in a computing node is occupied by processors (56% to 64% [53]). Although power consumption in processors depends on the types and the number of accelerators installed into the system, the percentage of power consumption in processors gets much larger in accelerator platforms. Power gating (PG) and Dynamic voltage and frequency scaling (DVFS) are two of the most effective power optimization techniques to reduce power consumption in processors. They may also be effective when reducing the power consumption of heterogeneous systems. However, they have not been well studied in heterogeneous systems with accelerators yet.

In particular, we are going to tackle the following three problems to improve energy-efficiency of heterogeneous systems.

Improving programmability of accelerators Directive-based programming models have been shown to be effective to ease the programming difficulty in utilizing accelerators [9, 22, 34, 56, 59]. However, existing directive based compilers are limited to single accelerator environments. It is necessary to extend them to multiple-accelerator environments, which is popular in practical situations.

Reducing leakage power of processors In today's high performance processors, leakage power is a non-negligible fraction in total power consumption of active processors. Power gating (PG) is a circuit technique to make circuit blocks asleep to reduce the leakage power. PG has been studied and shown to be effective in various circuit blocks in CPUs and accelerators [25, 30, 6]. However, we found that conventional PG techniques are not effective for CPU functional units. This is because idle periods in CPU functional units are very short and we cannot ignore the energy overhead caused by PG mode transitions. It is necessary to develop a sleep control technique to take the energy overhead into account.

Reducing dynamic power of processors To enable system administrators to optimize power utilization in large scale systems, it is important to provide knobs to control power consumption in each computing node. Dynamic voltage and frequency scaling (DVFS) is an effective technique to realize such power management with minimum performance degradation [17]. Unlike in homogeneous systems, it is not easy to predict the performance and power fluctuation with frequency scaling in heterogeneous systems because task mapping between CPUs and accelerators affects the amount of fluctuation. Hence, to apply DVFS to such systems, it is necessary to develop a technique to guide the settings of DVFS and heterogeneous task mapping cooperatively.

1.2 Contribution

In this dissertation, we propose three new compiler and runtime techniques. The three techniques can be orthogonally applied to applications running on heterogeneous systems. Although the

techniques mainly target at heterogeneous systems equipped with GPUs, which are the most popular accelerator platforms, they should be applicable to heterogeneous systems equipped with other accelerator devices, too. Brief summaries of the proposed techniques are described in the following subsections.

1.2.1 Improving Programmability (Chapter 3)

The industry and a lot of academic researchers have been studying high-level programming models for accelerators to ease the programming complexity [8, 43, 52]. Among those, directive-based accelerator programming models [9, 22, 34, 56, 59] have become the center of attention because of their simplicity and their similarity to OpenMP, which is popular in multiprocessor systems. OpenACC is the first industry standard of such a directive based accelerator programming model, released in 2011 [3]. Previous work reported promising results of the OpenACC programming model [35, 38, 48, 58].

However, the previous work also pointed out some limitations of the current OpenACC compilers. One of the biggest limitations is that the current OpenACC compilers do not automate the utilization of multiple accelerators. In the application development with the low level languages such as CUDA and OpenCL, utilization of multiple GPUs is a popular technique to further improve the performance [57]. For wide acceptance of OpenACC platforms, it is necessary to integrate the multi-device execution into OpenACC compilers.

In order to integrate multi-device execution into an OpenACC compiler, we propose a new compiler design and a new memory management technique among multiple accelerators. The compiler system includes a software distributed shared memory which is customized for multiple accelerator environments. In addition, we also propose a small set of new directives to optimize data movement. The directives allow compilers to optimize the data movement among physically disjointed memories according to the application characteristic. With the proposed compiler, we can simplify the programming process in utilizing multiple accelerators. In terms of number of code lines, it brings us 68% of reduction to the program sizes when compared to programs written in a low level language (CUDA). Meanwhile, it achieves 71% of the performance of hand-tuned

CUDA programs.

1.2.2 Reducing Leakage Power (Chapter 4)

Runtime power gating is a well known circuit technique to reduce leakage power consumption of active processors. It can make idle circuit blocks asleep to reduce the leakage power consumption during applications executions. Since the leakage power occupies a large fraction of power consumption in high performance processors, it is effective to make use of runtime power gating in order to improve the energy-efficiency of the system.

When we apply runtime power gating to circuit blocks in active processors, it is important to control sleep-wakeup mode transitions by considering the energy overhead. To avoid large energy overhead caused by excessive mode transitions, time-based sleep control techniques have been proposed both for CPUs and GPUs [25, 30, 6]. However, in CPU functional units, we found that these conventional techniques can miss a lot of chances to reduce the leakage power consumption. The problem is that time-based sleep control techniques are not effective for fine-grained idle periods, whose length is comparable to break even cycles, where the energy overhead can easily surpass the energy saving gained in sleep mode.

To reduce the leakage power consumed in such fine-grained idle periods, we propose a novel code analysis technique to predict the length of each idle period. With the proposed technique, the compiler can predict lengths of idle periods in cycle level accuracy. Using the prediction, the compiler can guide the sleep/wakeup mode transitions effectively. The experimental result for CPU functional units shows that the compiler based sleep control can reduce 23.6% of the leakage power compared to the conventional time-based technique.

1.2.3 Reducing Dynamic Power (Chapter 5)

Dynamic voltage and frequency scaling (DVFS) is an effective circuit technique to reduce the dynamic power consumption of processors. Because the optimal frequency depends on application characteristic, researchers have investigated techniques to select the optimal processor frequency in various situations [28].

However, it is not easy to select the optimal frequencies of CPUs and accelerators when the application makes use of both CPUs and accelerators in parallel. Such hybrid computation is common in heterogeneous systems because it improves the performance and energy-efficiency [21, 39, 46, 50]. In this case, the optimal frequencies do not only depend on the application characteristics but also depend on task mapping between CPUs and accelerators. It is then necessary to develop techniques to cooperatively select the device frequencies and the task mapping.

To do so, we propose a proactive technique in which the settings of frequencies and task mapping are determined in advance of the execution. The proposed technique includes new empirical models of performance and power of the target heterogeneous system. They can predict performance and power of the system when we change device frequencies, task mapping or both of them. Using such models, the proposed technique enables us to select the optimal setting of device frequencies and task mapping. The experimental result shows that, when we optimize the system performance under the given power budget, the performance with the proposed technique is higher (up to 14.4%) than that with DVFS control techniques which don't consider the task mapping. In particular, the performance of the proposed technique is very close to the performance with ideal parameter settings under the given power budget.

1.3 Organization of the Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 describes the background information about heterogeneous systems with accelerators and fundamental power management techniques. In Chapter 3, we attack the programmability problem in the accelerator platforms and propose a multi-device directive based compiler. In Chapter 4, we propose a compiler technique to reduce the leakage power consumed in CPU functional units. In Chapter 5, we propose a cooperative DVFS and task mapping technique which enables us to reduce the total power consumption of the heterogeneous systems with the minimum performance degradation. Then, limitation and future work of this dissertation is discussed in Chapter 6. Related work is shown

in Chapter 7 and the conclusion is presented in Chapter 8.

Chapter 2

Background

Recently, the amount of data processed in computer systems has been increasing rapidly. To this end, data parallel applications, where the same tasks are executed on individual data elements, has become important. The inherent parallelism in the data parallel applications can grow arbitrarily as the size of input data grows. They can be efficiently executed on parallel computing systems with more than ten thousands of processors. Accelerators, which have specialized architecture for data parallel computing, have emerged as the key computing elements [20]. In this section, we will describe the background information about the hardware organizations of the heterogeneous systems with accelerators, their programming environments, and the fundamental low power technology for computing systems.

2.1 Heterogeneous System

Figure 2.1 shows an illustration of heterogeneous system equipped with accelerators. The system consists of latency processors (LP), accelerators (ACC), and the interconnection which connects these computing devices. Latency processors are usually conventional chip multi-processors, where an operating system is running. Plus, multiple accelerators are installed to the system in order to improve the performance and the energy efficiency. It is common that the accelerator devices have dedicated local memories to fulfill their high memory bandwidth requirement for the accelerators. Hence, the system may not have a single shared memory space and the communication performance between computing devices is often limited and asymmetric.

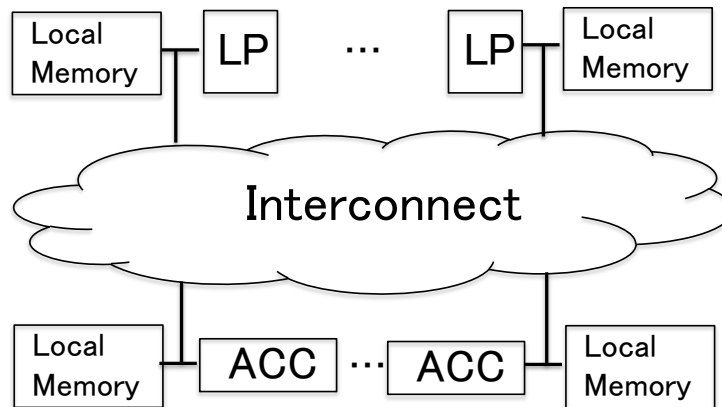


Figure 2.1: Heterogeneous computing system.

2.1.1 GPU Computing Server

GPUs consist of more than hundreds of simple in-order cores. This means that most of the transistors on GPUs are used for actual computation. The GPU architecture is very different from conventional CPU architecture where most of transistors are used for non computational modules, such as instruction scheduling modules or cache modules. As a result, GPUs can provide much higher performance and energy-efficiency than CPU if the applications have massive parallelism and they are optimized for the GPU architecture [36].

Figure 2.2 shows photos and a module diagram of a commercial GPU server. An Intel CPU is used as a latency processor, and two NVIDIA GPUs are used as accelerators. Main memory can be used only by the CPU. The GPUs have dedicated high throughput memory, which is called GPU memory. A PCI express bus is used as interconnect between the CPU and the GPUs. When the tasks are offloaded to the GPUs, data movement between the main memory and GPU memory must be commanded. Due to the limited performance of the PCI express bus, this data movement often becomes performance bottleneck.

2.1.2 Other Examples of Heterogeneous Systems with Accelerators

Many Core Processors An Intel Xeon Phi coprocessor is one of the most famous examples for many core processors [23]. It features tens of in-order cores with vector units on a single die.

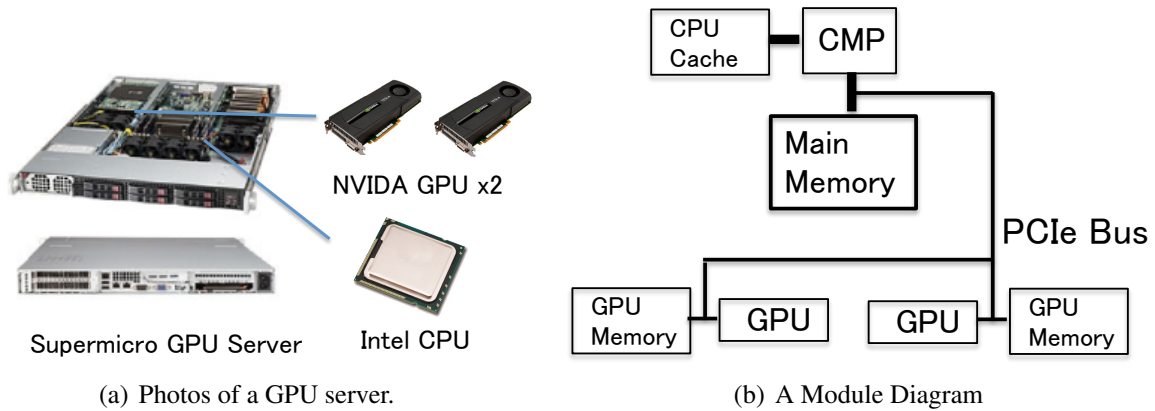


Figure 2.2: An example of CPU-GPU heterogeneous systems.

Similar to GPUs, a Xeon Phi coprocessor is physically mounted in a PCIe slot and has dedicated memory. Communication between the CPU memory and the Xeon Phi memory is also required. Thus, in the system with Xeon Phi coprocessors, the memory organization is quite similar to that in the GPU server.

FPGA Field-programmable gate array (FPGA) is an integrated circuit which can be configured after manufacturing. FPGA can also be an efficient computing device for data parallel applications because the hardware logic in FPGA can be directly customized and optimized for an application. Traditionally, the FPGA configuration is specified by hardware description languages. However, recently, Altera have shown that it is possible to translate an OpenCL programs into FPGA configuration and FPGA can be used as general purpose accelerators [13].

GPU Cluster So far, we focus on heterogeneity inside single computing node. Beyond a single heterogeneous node, many practical computing systems include hundreds of such heterogeneous computing nodes connected with Infiniband interconnect or other high performance inter-node network. For example, more than 50 systems in the Top500 supercomputer systems consist of hundreds of GPU computing nodes. These systems are called GPU clusters [32].

2.2 Programming Model for Accelerators

Because the architectures of accelerators are very different from CPUs, we need special parallel programming environments to utilize the accelerators.

2.2.1 CUDA and OpenCL

CUDA is a parallel computing platform and programming model created by NVIDIA and implemented by the graphics processing units that they produce. In this paper, we use CUDA to indicate the CUDA C/C++, which is the GPU programming interface in C/C++ languages. In CUDA, programmers can express massive parallelism in the application by using the special grammars. The parallel code is compiled into the assembly language used in the GPU. To get maximum performance gain from GPUs, programmers have to optimize the memory access in parallel codes by considering the GPU memory architecture, such as on-chip shared memory or a coalesce access mechanism. Furthermore, programmers have to manually manage and optimize the communication between a CPU memory and GPU memories because the GPUs have physically discrete memories and the communication between CPUs and GPUs tends to be the performance bottleneck.

Figure 2.3 shows an example of a vector addition program written in CUDA. We remove error checking codes for simplicity. The actual parallel tasks are written in the function which is declared with the special qualifier *__global__* (the line 1–10). The function is executed on the GPUs. In the code at the line 12 to 35, data movement between the CPU memory and the GPU memory is manually commanded with the CUDA API functions such as *cudaMalloc* and *cudaMemcpy*.

For accelerators other than NVIDIA GPUs, including AMD GPUs, FPGAs, DSPs and others, OpenCL [4] is defined as a portable and standard programming interface for accelerator programming. The programming model and API of OpenCL is also low level and very similar to those of CUDA.

```
1  __global__ void
2  d_vectorAdd(const float *A, const float *B,
3             float *C, int numElements)
4  {
5      int i = blockDim.x * blockIdx.x + threadIdx.x;
6      if (i < numElements)
7      {
8          C[i] = A[i] + B[i];
9      }
10 }
11
12 void
13 vectorAdd(const float *h_A, const float *h_B,
14           float *h_C, int numElements)
15 {
16     float *d_A = NULL;
17     cudaMalloc((void **)&d_A, size);
18     float *d_B = NULL;
19     cudaMalloc((void **)&d_B, size);
20     float *d_C = NULL;
21     cudaMalloc((void **)&d_C, size);
22
23     cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
24     cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
25
26     int t_per_b = 256;
27     int b_per_g = (numElements + t_per_b - 1) / t_per_b;
28     d_vectorAdd <<<b_per_g, t_per_b>>>(d_A, d_B, d_C, numElements);
29
30     cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
31
32     cudaFree(d_A);
33     cudaFree(d_B);
34     cudaFree(d_C);
35 }
```

Figure 2.3: CUDA vector addition.

2.2.2 OpenACC

CUDA or OpenCL can be accepted for expert programmers while not for common programmers due to its low programmability. For wide acceptance of accelerator computing, more productive accelerator programming environments have been proposed [8, 43, 52]. Especially, the directive-based accelerator programming models have become the center of attention because of their simplicity and their similarity to OpenMP, which is popular in developing parallel applications for multiprocessor systems [9, 22, 34, 56, 59].

OpenACC is the first industry standard of such a directive based accelerator programming model, released in 2011 [3]. The OpenACC API is designed to offload the low-level accelerator programming process to the compiler. It assumes the similar execution model as CUDA. That

```

1 void
2 vectorAdd(const float *h_A, const float *h_B,
3           float *h_C, int numElements)
4 {
5     #pragma acc kernels copyin(h_A[0:numElements], h_B[0:numElements])\
6         copyout(h_C[0:numElements])
7     {
8         #pragma acc loop independent
9         for (i=0; i < numElements; ++i) {
10            h_C[i] += h_A[i] + h_B[i];
11        }
12    }
13 }

```

Figure 2.4: OpenACC Vector Addition.

is the main program runs on the CPU and the data parallel tasks are offloaded to the accelerators. Unlike CUDA or OpenCL, which provide the unique languages to describe parallel tasks for accelerators, the OpenACC API provides OpenMP-like directives to use accelerators. The directives allow programmers to run parallel tasks written in C or Fortran on accelerators only with a few additional lines of code. It can be thought as an accelerator extension of OpenMP.

Figure 2.4 illustrates a vector addition program written in OpenACC. As we can see, the amount of complexity in the program is smaller than that in the program shown in Figure 2.3. At the line 5-6, the *kernels* directive is used to identify the code regions to be offloaded to accelerators. In addition to annotate the parallel code regions, the *copyin* and *copyout* clauses indicate which arrays would be read or written on accelerators. The hint information about array accesses helps the OpenACC compiler to avoid generating unnecessary data movement between CPUs and accelerators. At the line 8, the loops are annotated with the *loop* directives. The loop is the actual candidates to be transformed into the kernel program which is executed on accelerators. Also, like OpenMP, the *reduction* clause can be used for scalar variables (not shown in the Figure 2.4).

2.3 Power Management Technique

Power consumption in processors consists of dynamic power (P_D) and leakage power (P_L). Dynamic power is power consumed in switching activity of CMOS circuits. Given the capacitance

of the circuit C_L , the supply voltage V_{dd} and the frequency of the device, dynamic power is the product of energy per charge-discharge ($C_L V_{dd}^2$) and the number of switching per unit time (αf_{clk}). On the other hand, static power is consumed due to the leakage current ($I_{leakage}$) in CMOS circuits. This kind of power consumption is called leakage power consumption and it exists both in idle and active periods. The power consumption of processors can be modeled with the following mathematical equation [11]:

$$P_{processor} = P_D + P_L = \alpha C_L V_{dd}^2 f_{clk} + I_{leakage} V_{dd}. \quad (2.1)$$

A lot of researchers have investigated both circuit and architectural techniques to reduce both leakage and dynamic power of processors. Here, power gating and DVFS are explained as elementary circuit techniques which enable us to control the power consumption of the processors. Also, power capping is introduced as a key technology in computing center level power management.

2.3.1 Power Gating

Power gating (PG) is one of the most important circuit techniques to reduce leakage power consumption of CMOS circuits [45]. As shown in Figure 2.5 (a), the sleep transistors are inserted between the circuit block and the ground wires to provide the sleep mode in which we can save the leakage current in the circuit block by cutting off the supply voltage.

The energy overhead of the mode transition is not negligible and BET (Break Even Time) must be considered. Power transition during a sleep/wakeup mode transition is shown in Figure 2.5 (b). BET is the time period when the energy saving in the sleep mode balances with the energy overhead caused by the mode transition. The energy overhead of mode transitions, the area slashed as 1 in Figure 2.5 (b), is mainly caused by switching activity in a sleep transistor and charging/discharging for capacitance of the target circuit block. The length of BET depends on the process technology, the structure of circuit blocks, and the temperature. In the case of PG in the CPU functional units, which is the main concern in this dissertation, the length of BET is less than 100 cycles [51]. In order to achieve large leakage power reduction by using runtime PG, we

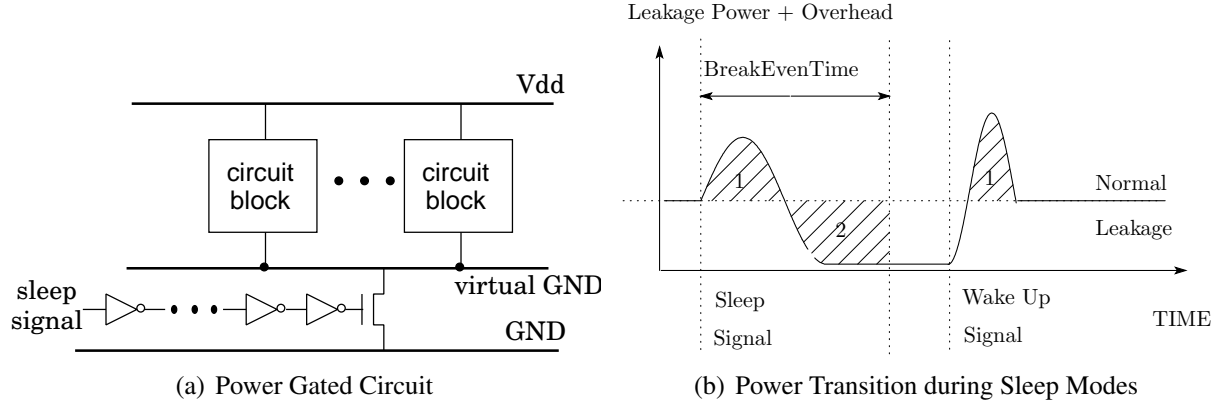


Figure 2.5: Power gated circuit and the power transition during sleep mode.

need sophisticated sleep control technique to avoid unnecessary mode transitions and the energy overhead. Ideally, each functional unit should go into the sleep mode only when the length of the following idle period exceeds the length of BET.

2.3.2 DVFS

DVFS (Dynamic Voltage and Frequency Scaling) is a technique to reduce dynamic power consumption of processors. There is a trade off between the frequency and power consumption of the processor and we can improve the energy-efficiency of the system by tuning the frequency according to applications characteristic.

In the power model shown in equation 2.1, we can see that the power consumption of processors depends on the supply voltage. Thus, we can reduce the power consumption of processors by reducing the supply voltage. However, we have to decrease the clock frequency of the device according to the supply voltage because lower supply voltage results in longer signal propagation time. Given the threshold voltage of the transistors V_{th} , the relationship between the supply voltage and the clock frequency is formalized in the following equation:

$$f_{clk} \propto \frac{(V_{dd} - V_{th})^\alpha}{V_{dd}}. \quad (2.2)$$

α is a parameter which depends on process technology. It is usually from 1 to 2. In DVFS, as shown in equation 2.1 and the equation 2.2, there is a trade-off between the power consumption

of processors and the clock frequency. The amount of performance change caused by the fluctuation of the clock frequency depends both on the processor architecture and the applications characteristics.

2.3.3 Power Capping

Power capping is an elementary technique which is used in the global power management of large scale computer systems. It eliminates the burst of power consumption in components or computing nodes and keeps the power consumption under the given power constraint. Based on power capping, we can optimize the power utilization among servers according to online activity of running applications [17].

At the component level, power capping techniques for CPUs [15] or system memories [28] have been already proposed. On a different level, power capping for a single computing node is important for large scale computing systems [37]. In homogeneous systems with CPUs, several power capping techniques have been proposed, but power capping in heterogeneous systems with accelerators has not been well studied.

Chapter 3

Multi-Device Execution in a Directive Based Compiler

In this chapter, we focus on a technique to utilize multiple accelerators from OpenACC, standard programming API for directive based accelerator programming.

The difficulty lies in the communication management among multiple accelerator memories. In the conventional programming models, such as CUDA or OpenCL, programmers have to manually distribute data among the different memories and have to keep consistency of the data replicated on the multiple memories. It is often the case that the complicated optimization is required to avoid the performance bottleneck at the data movement among the distributed memories. To integrate such optimized communication management into an OpenACC compiler, we propose a compiler and a runtime memory management mechanism which makes single memory space illusion on top of the multiple accelerator memories.

3.1 Drawbacks of Current GPU Programming Environments

Figure 3.1 shows the performance and the number of code lines in several parallel implementations of two GPU friendly applications *KMEANS* and *BFS*. The implementations include OpenMP (*OpenMP*), OpenACC (*ACC*), and CUDA $\{CUDA(1, 2GPU)\}$ with one and two GPUs¹. Exper-

¹Note that we cannot implement a multi-GPU OpenACC program for *KMEANS* and *BFS* because these application requires us to use critical sections in the parallel loops. The current OpenACC API does not provide such functionality.

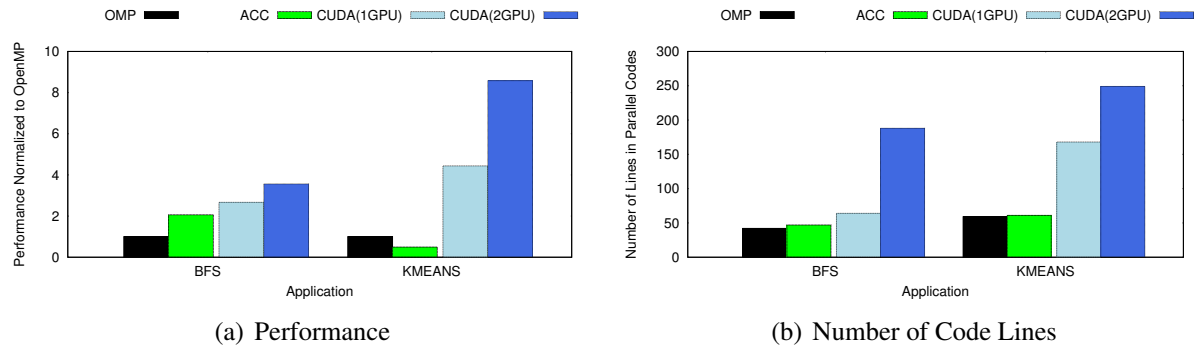


Figure 3.1: Comparison of current GPU programming environments.

imental setup is the same as that used in later evaluations: OpenMP uses one 6-core Intel CPU, OpenACC and CUDA uses one or two 448-core GPUs. In measuring the number of code lines, we only count the code lines which are used in actual parallel processing. The performance is normalized to the performance of OpenMP. Figure 3.1 (a) shows that we can achieve 3-8 times higher performance than that of *OpenMP* by utilizing multiple GPUs in *CUDA(2GPU)*.

However, in Figure 3.1 (b), we can see that we have to develop much larger programs to utilize multiple GPUs in CUDA. The numbers of code lines in *CUDA(2GPU)* is 4-5 times larger than those in the *OpenMP* implementation. The reason is as follows. In *BFS*, parallel tasks generate a lot of irregular write requests to visit the next graph node with chasing memory pointers. These irregular write requests can cause memory accesses to remote GPU memories in the multi-GPU execution. In the CUDA multi-GPU implementation, we have to manage special communication buffers on each GPU to manage these remote memory accesses. This increases the number of code lines of two GPU CUDA *BFS*. In *KMEANS*, complicated reduction operation is used to summarize the results of each parallel tasks. To implement the reduction on GPUs, we have to implement hierarchical parallel reduction algorithm which is optimized for the GPU memory hierarchy. In CUDA, this implementation needs a lot of code lines in the program.

As seen in this case study, no existing platform can provide both high programmability and high performance in utilizing multiple GPUs.

3.2 OpenACC Compiler with Software Distributed Shared Memory

To overcome the drawbacks of the current programming environments, we propose an OpenACC compiler which is integrated with software distributed shared memory. Figure 3.2 shows the concepts of the proposed compiler. To avoid the complicated programming interface for using multiple accelerators, the proposed compiler receives OpenACC programs. Unlike conventional platforms, it is unnecessary to rewrite the program to utilize multiple accelerators. The *compiler* and the *Multi-GPU memory manager* in the proposed system automatically distribute parallel loops and manage data among the physically disjointed memories. However, it is not easy for compilers to automate efficient communication among the memories because this highly depends on the applications characteristics, as explained in the previous case study. To solve the problem, we propose new directives for programmers to express the applications memory access pattern in parallel loops (*Memory access directives* in Figure 3.2). With the directives, the compiler can safely assume which data items are accessed on a certain device. It greatly helps compiler reduce unnecessary communication.

Figure. 3.3 illustrates the overall steps of the multi-device execution in the proposed compiler. Parallel loops are executed on the multiple accelerators with three steps. First, the system maps tasks and data to the multiple accelerators. The iteration space of the parallel loop is divided into tasks. The system also determines the mapping of the tasks to the multiple accelerators. At the same time, all the data potentially read by the tasks mapped to each accelerator are loaded into the corresponding accelerator memory. Here, the memory access information given by the newly introduced directives is used to reduce unnecessary data load. Next, actual computations are executed in parallel on the multiple accelerators. In this step, all write requests on data replicated among multiple accelerator memories and all write requests on data which is not on the local accelerator memory are recorded by the system. This is done by additional codes instrumented in parallel kernel functions. Finally, the system handles necessary inter-accelerator communications, which include handling of writes to the replicated data, and include the handling of

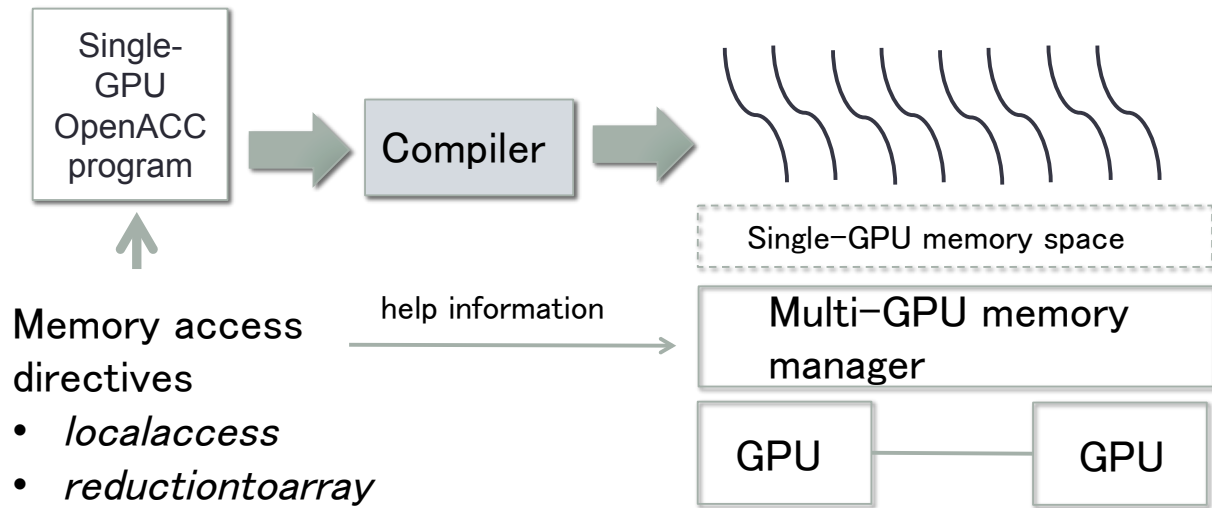


Figure 3.2: Overview of the proposed multi-device OpenACC compiler.

irregular writes whose destinations do not present in the local accelerator memory. In the final step, the memory access information given by the newly introduced directives is also used to update remote data items efficiently. Then, a global barrier operation among accelerators occurs and the next parallel loop will be executed.

3.2.1 Directive Extensions

In order to allow compilers to optimize communication according to the application characteristic, we propose two new directives as an OpenACC extension for multiple accelerator environments. Note the proposed system uses these directives only for performance tuning purpose. The compiler can execute any OpenACC program on multiple accelerators without using these newly introduced directives.

The design of the directives are based on the common memory access patterns which are observed in typical data parallel applications. One directive is *localaccess* directive, which is used to describe the region of data elements read by each parallel task inside the parallel loop. The other is *reductiontoarray* directive, which is used to tell the compiler about complicated

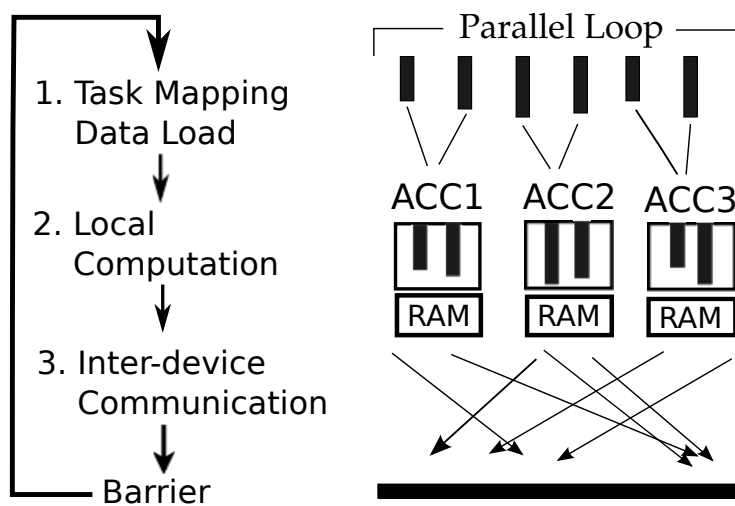


Figure 3.3: Execution steps of parallel loops in the proposed system.

reduction operations to array elements inside the parallel loops. These directives are designed based on typical memory access patterns observed in common data parallel applications.

localaccess [clause] The directive allows programmers to specify the range of indices for a certain array which can be read in i -th iteration of the loop. When the directive is given, the compiler can aggressively optimize the generated code with the assumption that the i -th iteration of the loop does not read any part of the array outside the specified range of indices. The range of indices must be consecutive. Therefore, it is specified by a pair of a lower bound of indices and an upper bound of indices. To express constant stride read accesses, the directive supports *stride(ArrayName[stride:left:right])* clause. This means that the i -th iteration of the loop use the array elements whose indices are from $(stride * i - left)$ to $\{stride * (i + 1) - 1 + right\}$. To support non-uniform stride access, programmers can specify an index array which contains lower bounds of access indices for each iteration of the loop. The *indirect(ArrayName[LowerBounds])* clause can be used for this purpose. We can specify an index array at *LowerBounds* and can specify the actual array to be read in the loops at *ArrayName*. The *LowerBounds* array contains lower bounds of indices used for the *ArrayName* in the i -th iteration of the loop. The upper bound is given by the lower bound on the $i + 1$ -th iteration of the loop like the compressed

```

1  #pragma acc loop
2  #pragma acc localaccess stride(x[1:0:0], b[1:0:0]) \
3      indirect(c[cIndex])
4  for (i=0; i < n; ++i)
5  {
6      for(j=cIndex[i]; j < cIndex[i+1]; ++j){
7          x[i] *= c[j];
8      }
9      #pragma acc reductiontoarray (+:errors[0:e_size])
10     {
11         errors[b[i]] += x[i];
12     }
13 }

```

Figure 3.4: An simple example of the proposed directives.

sparse row format used in the field of sparse linear algebra.

reductiontoarray [clause] We extend the scalar reduction clause for arrays. Unlike the conventional *reduction* clause, the directive is used inside parallel loops to annotate single reduction statement directly. We can specify the name of the destination array and the range of indices in the clause. The access index of the destination array can be dynamically determined. The compiler generates optimized reduction codes which can run efficiently on the multiple accelerator environment.

Example Fig. 3.4 illustrates an example of a C program annotated with the proposed directives. In the code, the read access patterns for the array *x*, the array *b* and the array *c* are passed to the compiler through the *localaccess* directive (line 2). On the other hand, the *errors* array does not have the *localaccess* directive. In this case, the compiler does not aggressively optimize the data movements for the array (detailed in Section 3.2.2). Also, we use the *reductiontoarray* directive to tell the compiler that the statement at line 10 must be treated as the reduction operations whose destinations are the elements in the array *errors*. Note that programmers do not have to consider the existence of multiple accelerators because no task mapping and no data transfer between multiple accelerators are manually commanded.

3.2.2 Implementation and Optimization in multi-GPU platforms

We have implemented a prototype system of the proposed compiler on top of multi-GPU platforms. Although, we have implemented the prototype system for NVIDIA CUDA platforms, the proposed design of the system can be applicable to other accelerator platforms in which we can use an OpenCL programming environment.

3.2.2.1 Module Diagram

Figure 3.5 illustrates a module diagram of the prototype system of the proposed OpenACC compiler. *C to Multi GPU CUDA Translator* (translator) is designed as a source to source translator. The translator generates CUDA kernel codes and host codes from C programs annotated with the OpenACC directives and the proposed extension. In addition to generating CUDA codes, the translator also generates the information which summarizes memory access patterns for arrays. The information is used in the runtime system in order to optimize data movement among distributed memories. *GPU Data Loader* (data loader) manages necessary data movement between the CPU memory and the multiple GPU memories according to the OpenACC semantics. To make sure that the execution is correct, all the data which are potentially read by the kernel running on each GPU must be loaded into the corresponding GPU memory before the kernel execution. *Inter-GPU Communication Manager* (inter-GPU communication manager) is responsible for handling inter-GPU communication. It is called just after the kernels executed on every GPU. It checks write operations done by the kernel on each GPU and updates remote GPU memories.

3.2.2.2 Translator

The translator generates a CUDA kernel and host codes which utilize multiple GPUs. In the current implementation, a parallel loop annotated with an OpenACC *loop* directive is transformed into a CUDA kernel function. The translator replaces the original loop with the call statement for the kernel function. Also, the translator generates the CUDA host code which includes the control codes to initialize the devices, to call the kernel functions, and to control the data movement among distributed memories.

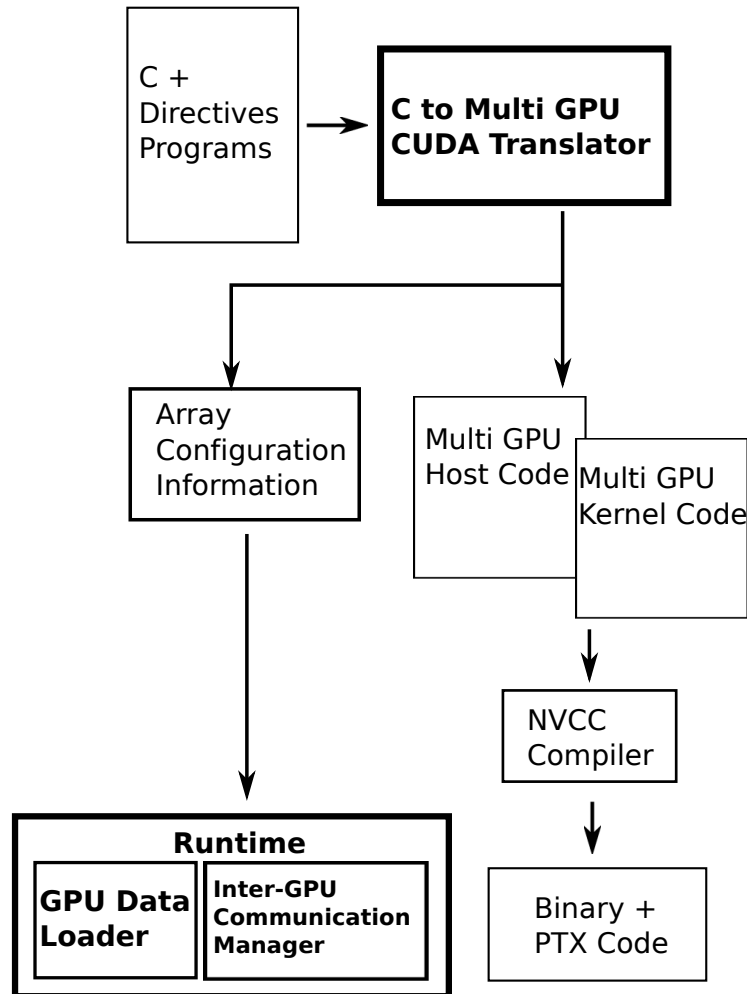


Figure 3.5: An overview of the prototype system

Handling Data Movement The actual operations for data movement among the distributed memories are delegated to the runtime system. The translator just inserts statements to call the runtime functions at the program points where the data movements are required.

Task Mapping and Thread Generation In the current implementation, the tasks in the parallel loop are equally divided among the GPUs. Before the call of the kernel function on each GPU, the host program sets up the number of the thread blocks and the number of the CUDA threads per blocks for the GPU according to the number of the assigned tasks to the GPU.

Organizing Array Accesses on GPUs In the kernel functions, all indices in array accesses must be recalculated by considering data layout of the array on each local GPU memory. To do so, the host program includes the codes to ask the runtime system about the layout of the arrays among the GPU memories and includes codes to pass the information to the arguments of the kernel functions. The translator rewrites indices of the array accesses in the kernel function by using the arguments. In addition, the translator inserts additional codes for the inter-GPU communication manager to identify which parts of the arrays are written by the kernel (detailed in Subsection 3.2.2.4).

Optimizing Kernel Functions The translator applies two GPU-specific optimizations to kernel functions. One is the data layout transformation of two-dimensional arrays for enhancing the coalesced memory accesses. The transformation is applied to device arrays which satisfy the following three conditions: read-only, the access indices are all in affine forms, and the array has the *localaccess* directive. Also, to avoid the performance bottleneck at reduction operations, the translator uses the hierarchical reduction algorithm for the reductions in the kernel functions. At first level, the reduction is done on the shared memory for each thread block. Next, the values are collected among the thread blocks on the same GPU. Finally, the values are transferred and merged between multiple GPUs.

Generating Array Configuration Information The translator generates array configuration information, which is used by the data loader and the inter-GPU communication manager. The information summarizes memory access patterns of arrays. It is generated for every parallel loop and for every device array in the loop. The information contains several attributes of the array, including whether the array is read-only or write-only, the range of the access indices in each loop iteration (if the array has the *localaccess* directive), and the array is the destination of complicated reduction operations.

3.2.2.3 Data Loader

In OpenACC, the compiler manages the data movement between the system memory and the device memory. The data loader is responsible for guaranteeing the semantics of the GPU memory management while it transparently manages multiple GPU memories.

The data loader is called at the entrance and the exit points of the *parallel* regions, the *kernels* regions and the *data* regions. In these regions, the data loader is called at the point where programmers command data movement by inserting OpenACC directives such as the *update* directives. The data loader is also called before every kernel calls to load the necessary data into the GPU memories because the necessary data can be different between different kernel calls.

In order to avoid unnecessary data movement among multiple GPU memories, the data loader makes use of two different policies to load arrays into GPU memories. One policy is replica-based policy. With the policy, all data elements in the array are replicated to all the GPU memories. The other policy is distribution-based policy. In the policy, an array is divided into sub-arrays and only the sub-array actually accessed by each GPU is loaded into the corresponding GPU memory. With the distribution-based policy, the arrays require less amount of data movement and less amount of device memory footprints than arrays with the replica-based policy.

The data loader decides which placement policy should be used for each array according to the user hint information given from the *localaccess* directive. Figure 3.6 is the flow chart used to determine array placement policy. If an array is annotated with the *localaccess* directive at the top of the parallel loop, then, the data loader uses the distribution-based policy. Also, even if the array is not annotated with *localaccess* at the top of the corresponding parallel loops, the data loader uses distribution-based policy as far as the array is annotated with the *localaccess* directive at the top of the post dominant parallel loop. Otherwise, the data loader uses replica-based policy.

Note that the data loader can avoid additional data movement before the kernel calls when the read memory access pattern in the next kernel call is the same to that in the previous kernel call. This is common in iterative algorithms, the same parallel loop is executed many times, as seen in the applications evaluated in the evaluation chapter.

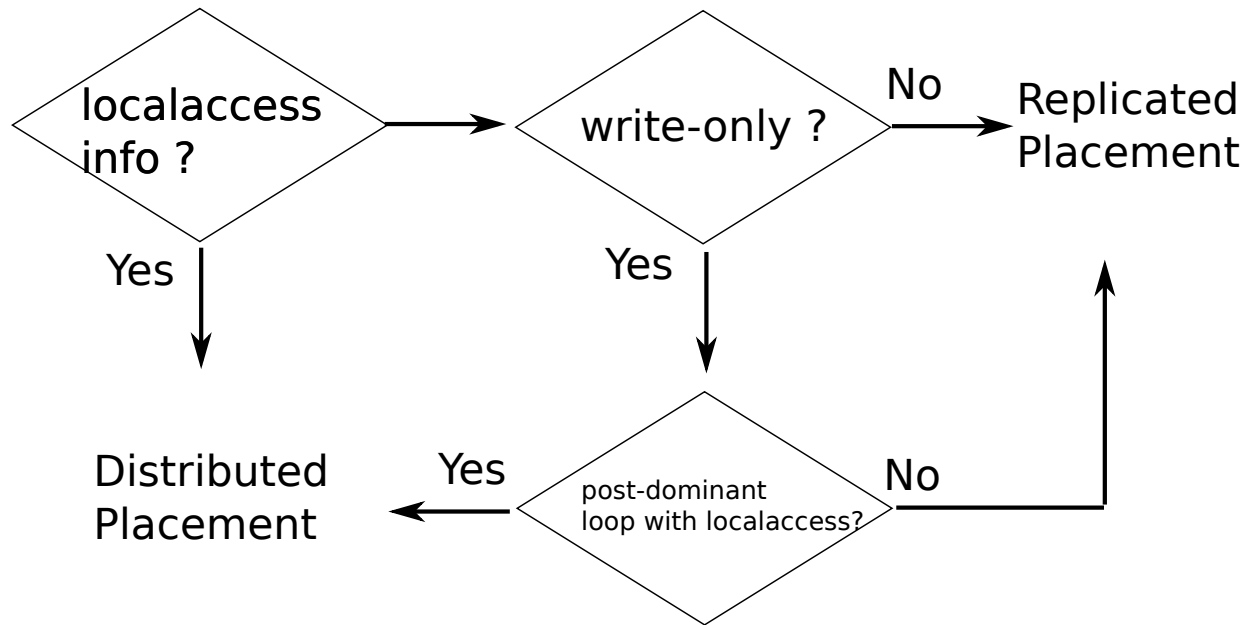


Figure 3.6: A flowchart to determine the placement policy of each array.

3.2.2.4 Inter-GPU Communication Manager

The inter-GPU communication manager is called just after the kernel functions executed on the GPUs. It handles necessary data exchanges between multiple GPU memories, including update operations at the writes to the replicated data and the remote write operations to the data which exist only in the remote GPU memory. To maximize the performance at the communication step, the communication manager directly exchanges the data between the GPU memories and the communications are executed asynchronously.

Replicated Array When write accesses occur in replicated arrays, the inter-GPU communication manager must update other copies on the different GPU memories to keep consistency.

To do this, the manager prepares the dirty bit arrays on the GPU memories for replicated arrays. In order to identify which elements of the array are written in a kernel execution, the translator inserts additional operations to turn on the dirty bits at every writes accesses to the replicated arrays. However, with the single level dirty bits, the manager has to transfer all array data, including clean elements and the dirty bits, to other GPU memories because the manager

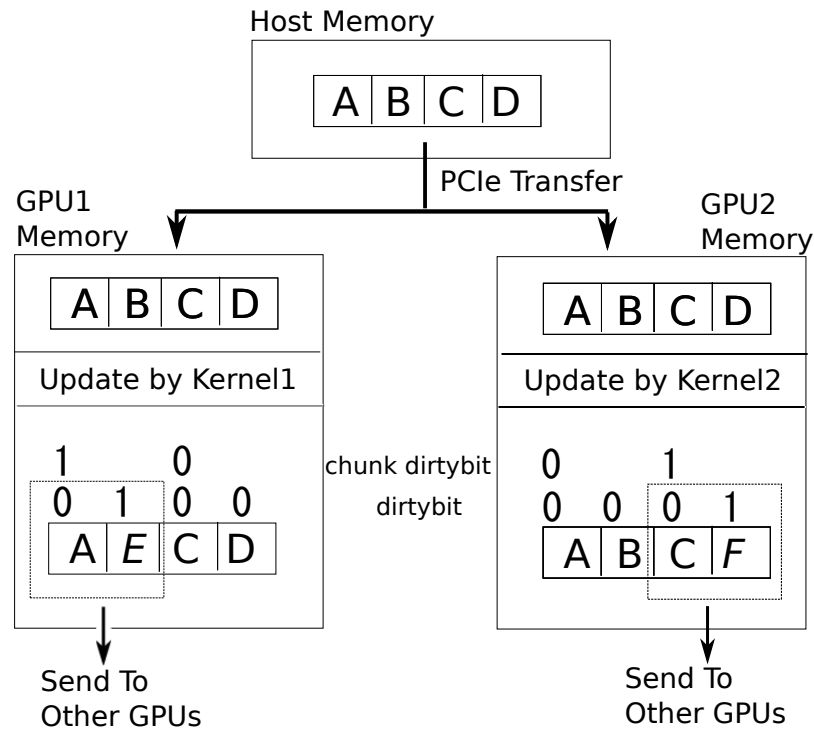


Figure 3.7: Two-level dirty-bit mechanism to keep consistency of the replicated arrays.

cannot check the contents of dirty bits efficiently on the sender GPU. It degrades the performance of the inter-GPU communications.

To solve the problem, we use two-level dirty-bit mechanism. It is illustrated in Fig. 3.7. A dirty bit array is subdivided into chunks whose sizes are constant. Each chunk also maintains single bit, which indicates all the dirty bits in the chunk are clean. The bit is used as the second level dirty bit. The translator add codes to turn on the second level dirty bits in kernel functions. With the second level dirty bits, the inter-GPU communication manager avoids unnecessary data transfers at chunks which have no dirty data. The optimal size of the chunks is dependent both on applications and hardware characteristics. In the evaluation chapter, we experimentally choose 1MB to the chunk size of the second level dirty bit arrays.

Distributed Array In the case of the arrays which are subdivided and distributed among multiple GPUs, we have to handle irregular writes to the data which do not exist in the local GPU memory. To correctly handle writes to data on remote GPU memories, the manager must know

that which write accesses missed on the local GPU memory in the previous kernel execution. To tell the manager about the write misses, the translator insert check codes at every write accesses on distributed arrays to identify the write misses. When the write access causes write miss, a pair of the written data and the destination address are temporarily buffered into the system buffers on the local GPU. After the kernel execution, the inter-GPU communication manager transfers the records of the write misses to the remote GPU memories where the destination exists. Then, the communication manager calls the CUDA kernels to complete the write access on the remote GPUs. If the compiler can statically analyze that the write address is always within the range described by the *localaccess* directive, we can eliminate the check code to avoid the additional performance overhead.

3.3 Evaluation

The translator is implemented by using ROSE compiler infrastructures developed at Lawrence Livermore National Laboratory [5]. It also uses parts of an existing C to CUDA translator [56], which is publicly available. The runtime system is implemented with C++ on top of the CUDA 4.0 platform. We evaluated the proposed compiler system in two different machines. One is the desktop machine equipped with two GPUs. The other is a thin-node of TSUBAME2.0 supercomputer at Tokyo Institute of Technology. Each platform differs by the type and the number of CPUs and GPUs. Also, the performances of communication buses are different. The details of the platforms are shown in the Table 3.1. We use three benchmark applications *BFS*, *MD*, and *KMEANS* selected from rodinia [12] and shoc [14] benchmark suites. They exhibit different inter-GPU communication characteristics. *BFS* is highly memory intensive with a lot of irregular writes. It is one of the most difficult applications to be efficiently executed in multiple GPU environments. On the contrary, *MD* requires no inter-GPU communications. *KMEANS* is in the middle of these two applications. It requires small amount of inter-GPU communications due to the existence of reduction operation whose destination is the array which is used in all GPUs. We summarize the details of the applications in Table 3.2.

We compare the performance of the several versions of the applications. The versions are as follows.

- OpenMP (*OpenMP*): The programs are written with OpenMP. They are compiled by the gcc compiler with the O2 optimization flag. The number of running threads is set to 12 in the Desktop Machine and 24 in the supercomputer node.
- CUDA (*CUDA(1, 2)*): The programs are written in CUDA. They are compiled by the nvcc compiler with the O2 optimization flag. The number in the label denotes the number of GPUs which is used in the implementation.
- Proposal(*Proposal(1, 2)*): The programs are written with the OpenACC API with the proposed extensions (the *localaccess* directive and the *reductiontoarray* directive). The generated CUDA codes are compiled by the nvcc compiler with the O2 optimization flag. The number in the label denotes the number of GPUs which is used in the implementation.

3.3.1 Performance and Programmability

In this subsection, we show the results of performance and the programmability in Desktop Machine in detail.

In Figure 3.8, we plot the pairs of the performance and the number of code lines (LOC) for each platform. The x-axis denotes LOC normalized to that of *CUDA(2)*. The y-axis denotes performance normalized to that of *CUDA(2)*. Each point represents each implementation of the application. We measure the execution time spent in the parallel regions, including the time

Table 3.1: Machine setup for the evaluation.

Desktop Machine	
CPU	Intel Core i7 x 1 (6core, Hyper Threading)
GPUs	Nvidia Tesla C2075 x2
Supercomputer Node	
CPU	Intel Xeon x 2 (12=2x6core, Hyper Threading)
GPUs	Nvidia Tesla M2050 x3

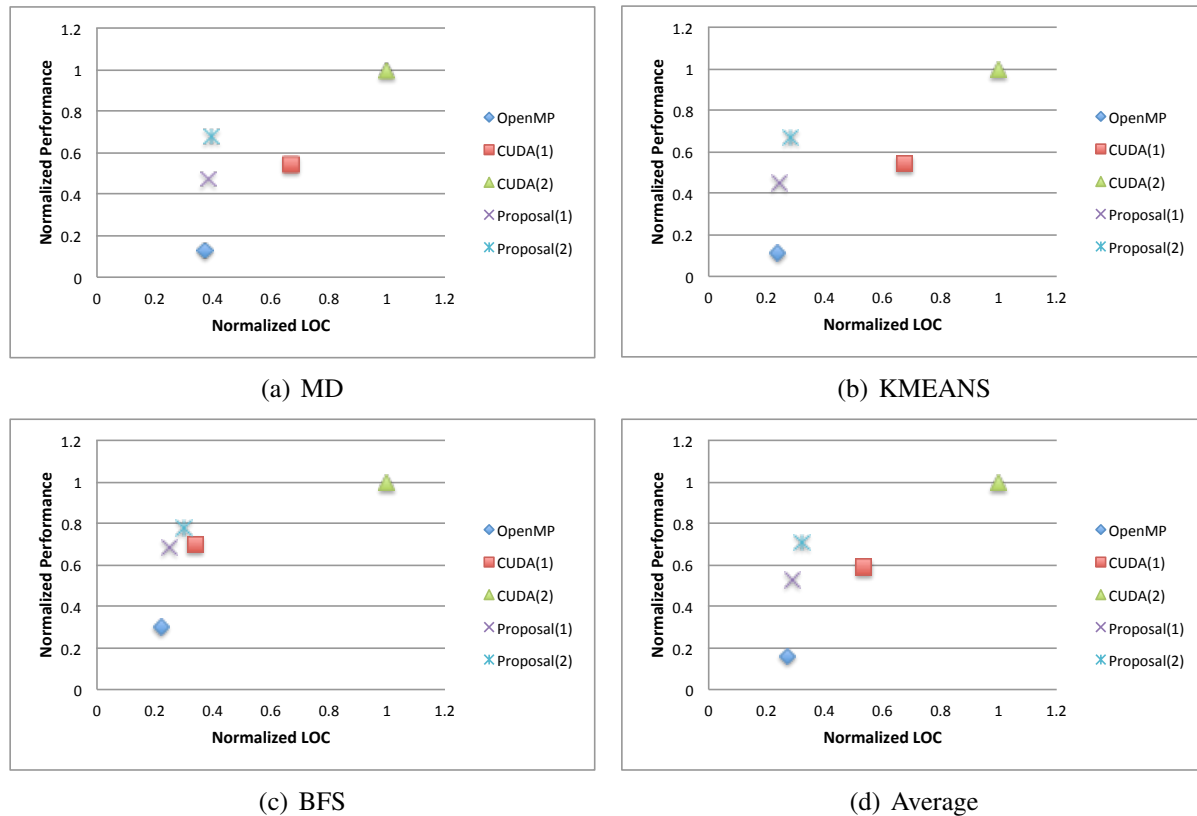


Figure 3.8: Performance and lines of code (LOC). Normalized to the CUDA 2GPU versions.

spent on the CPU-GPU communications and the GPU-GPU communications. In measuring the number of code lines, we only count codes related to actual parallel processing. We exclude I/O related operations to prepare the input data for parallel processing. However, we count codes related to transform data layout of arrays to enhance coalesce memory accesses on GPUs and codes related to prepare index arrays specified in the `localaccess` directive with the `indirect` clause. In the figure, a point is better if it is plotted in upper-left region because it is better for

Table 3.2: A: Total device memory usage in single GPU execution, B: # of parallel loops, C: # of kernel executions, D: # of arrays with `localaccess` directive / # of arrays used in parallel loops.

Application	Source	Description	Input	A	B	C	D
BFS	SHOC	Graph Traversal	5M node	444.9MB	1	10	2/3
MD	SHOC	Simulation	73728 Atom	39.8MB	1	1	2/3
KMEANS	Rodinia	Clustering	kdd_cup	69.2MB	2	37	2/5

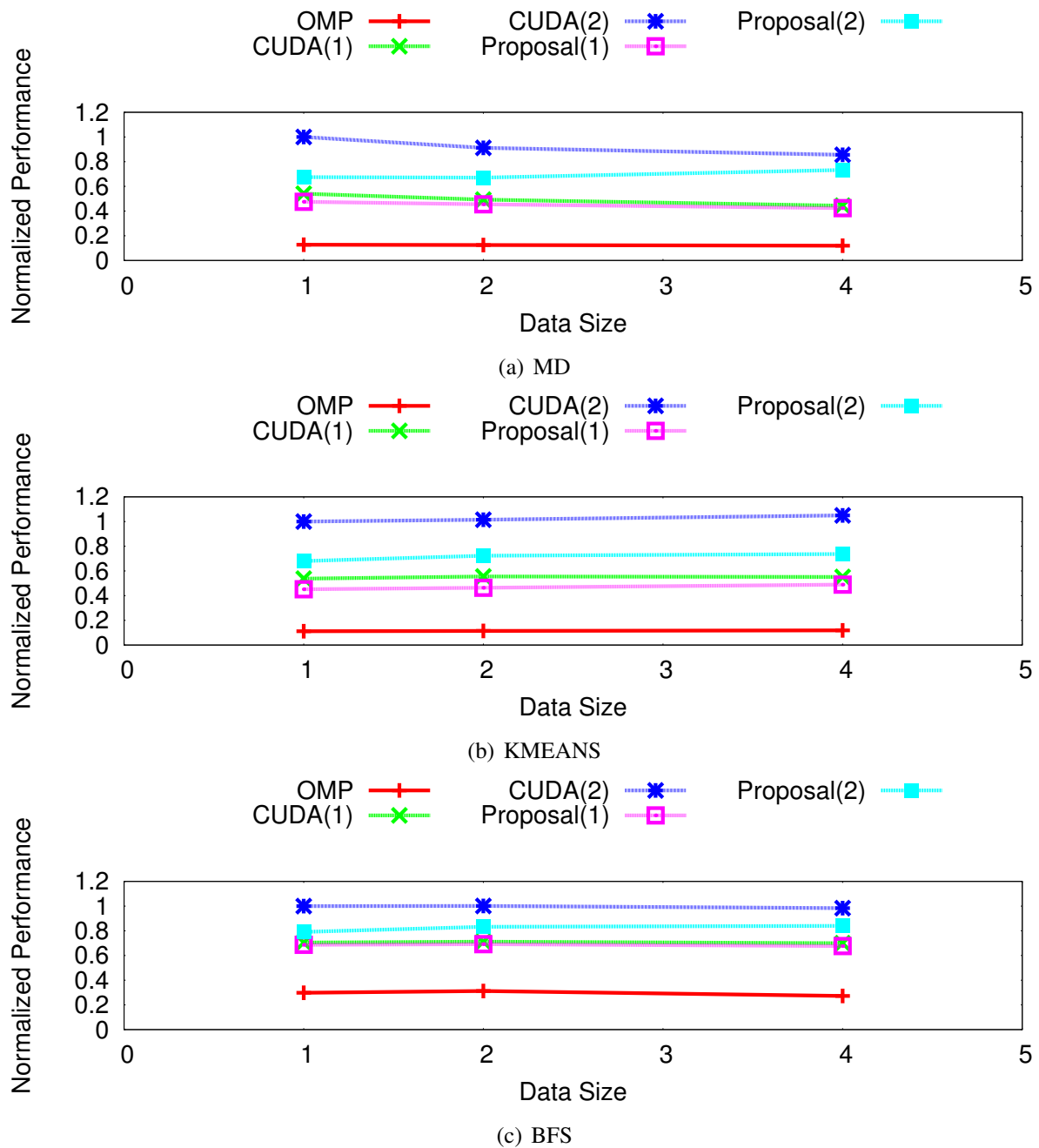


Figure 3.9: Performance trends when the size of input data changes. Normalized to the CUDA 2GPU versions with base inputs.

platforms to have the low LOC (x-axis) and the high performance (y-axis).

First of all, we can see that the points of *Proposal(1, 2)* are at the upper-left of other points, which represents other platforms (*OpenMP* and *CUDA(1, 2)*). This means that we can achieve better performance with smaller programs in the proposed system than in the other platforms. Comparing *Proposal(1)* and *Proposal(2)*, in the proposed system, we can improve the performance by utilizing 2 GPUs with small increase of LOC. On average, the performance with *Proposal(2)* achieves 71% of the performance with *CUDA(2)* while the LOC with *Proposal(2)* is 32% of the LOC with *CUDA(2)* (Figure 3.8 (d)). Also, Figure 3.9 shows the performance when we vary the size of input data. The performance is normalized to the OpenMP versions with the base data sizes, which are show in the Table 3.2. The performance trend does not significantly differ in the different data size.

Next, we look at the detailed performance analysis for each application through comparing *CUDA (2GPU)* versions and the proposed compiler.

MD In MD, there's no inter-GPU communications and the largest data array, which occupies the most of the CPU-GPU communication, are accessed with the constant stride in parallel loops. Hence, with the *localaccess* directives, the compiler can successfully avoid most of the unnecessary data movement as same in the optimized CUDA program. Therefore, in Figure 3.8 (a), *Proposal(2)* achieves almost same performance to *CUDA(2)* at large input data. This is because the runtime overhead becomes relatively smaller when the size of input data gets large and the percentage of computation time grows.

KMEANS Both in CUDA and the proposed compilers, KMEANS achieved the highest speed up by utilizing two GPUs among the evaluated applications. This is because it is possible to apply GPU specific optimizations to the parallel loops in KMEANS, such as data layout transformation to enhance coalesce accesses or parallel reductions on the GPU shared memory. On the other hand, for large data, the difference between performance of *Proposal(2)* and that of *CUDA(2)* is the largest among the applications. This is because that there is some difference in the imple-

mentations. First, in the proposed compiler, the data layout transformation is done on the CPU while it is done on the GPU in the CUDA program. Second, at the end of each iteration of the KMEANS parallel kernels, it is necessary to communicate some small arrays which contain the results of the reduction operations. The proposed compiler uses dirty-bit based coherence mechanism to communicate the small arrays. It incurs non-negligible performance overhead due to the execution time of the runtime memory manager and the communication time of the dirtybit array. The performance overhead is originated from unoptimized implementations in the compiler and the runtime memory manager. Hence, it seems to be reduced if we further optimize the implementations of them.

BFS Although the runtime memory manager has to handle a lot of irregular write requests which incurs inter-GPU communications, these requests still exist in the hand-written CUDA program. Then, the performance difference is not very large in BFS across the input data sets. One interesting observation is that the CUDA program uses the smaller size data type for the inter-GPU communication because the programmer can know the values of the communication data do not exceed a certain constant value. In particular, char type (1byte) is used for communicating data whose original type is integer (4byte). Such kind of application specific optimizations cannot be done by the proposed compiler. It is a fundamental limitation of the automatic GPU memory management. However, in this case, the effect is not very large and the proposed compiler achieves modest performance (79%-84% of the performance of *CUDA(2)*).

Finally, Table 3.3 shows the details of LOC in *OpenMP*, single GPU *OpenACC* (equivalent to *Proposal(1)*), *Proposal(2)*, and *CUDA(1)*. We can see that the numbers of code lines needed to implement the parallel programs for the proposed multi-GPU *OpenACC* compiler are almost same to the numbers of code lines in *OpenMP* and *OpenACC*. In summary, the proposed multi-GPU *OpenACC* compiler can provide comparable programmability to *OpenMP* or conventional single-GPU *OpenACC*, which provide better programmability than *CUDA*.

3.3.2 Evaluation in a Supercomputer Node

To see scalability of the proposed system, we evaluate the proposed system on the machine with more than 2 GPUs. Figure 3.10 (a) shows the performance of the proposed system in a thin node of TSUBAME2.0. We do not evaluate *CUDA(2)* and *CUDA(3)*. The computing node is equipped with 3 GPUs. The y-axis is the performance normalized to the performance of *OpenMP*. Also, we show the breakdown of the execution time in Figure 3.10 (b). To investigate the memory overhead caused by the runtime system, including the data replication and the temporary buffers on GPUs, Figure 3.10 (c) shows the GPU memory usages in the applications with the proposed system. The bars in the figure indicate the amount of memory used to store the user data (*User*) and the amount of memory used in the runtime system (*System*). The values are normalized to the total device memory usage in the single GPU execution, where no system memory or no replicated data exists.

In Figure 3.10 (b), we can see that the time spent on the data transfer between CPUs and GPUs (*CPU-GPU*) are the main reason that prevents us from achieving linear speed up to the number of GPUs (shown in Figure 3.10 (a)). Because the proposed system does not incur large overheads in the data transfer between the CPU and the GPUs, the limitation is originated in the application characteristics. Thus, the data transfer between CPUs and GPUs still prevents the linear speed up even when we manually make use of multiple GPUs with CUDA or OpenCL. To

		OpenMP	OpenACC	Proposal(2GPU)	CUDA(1GPU)
BFS	Actual Processing	40	40	40	42
	Directive	2	7	9	–
	API Call	–	–	–	22
	Other	–	–	7	–
	Total	42	47	56	64
MD	Actual Processing	26	26	26	26
	Directive	1	2	3	–
	API Call	–	–	–	23
	Other	–	–	–	–
	Total	27	28	29	49
KMEANS	Actual Processing	51	54	54	113
	Directive	8	7	16	–
	API Call	–	–	–	34
	Other	–	–	–	21
	Total	59	61	70	168

Table 3.3: Number of code Lines.

solve the problem, we have to rewrite the applications by redesigning the algorithm. However, this is beyond the scope of this work.

In Figure 3.10 (c), we can see that the amount of *User* memory, which includes the memory for replicated data, does not increase significantly even with the multi-GPU executions. If the data loader replicates all the data on every GPU memory, it would increase in proportion to the number of GPUs. However, with the memory access patterns provided through the *localaccess* directives, the proposed system can avoid this situation by making use of the access locality in the arrays. On the other hand, the runtime system consumes some amount of the device memory in order to handle the inter-GPU communications. The amount of device memory used in the runtime system is larger in the applications in which more inter-GPU communications are needed, such as *BFS*. However, the overhead is less than 30% even in *BFS*. Thus, the extra memory overheads in the proposed system do not decrease the benefits of larger available memory with multi-GPU executions.

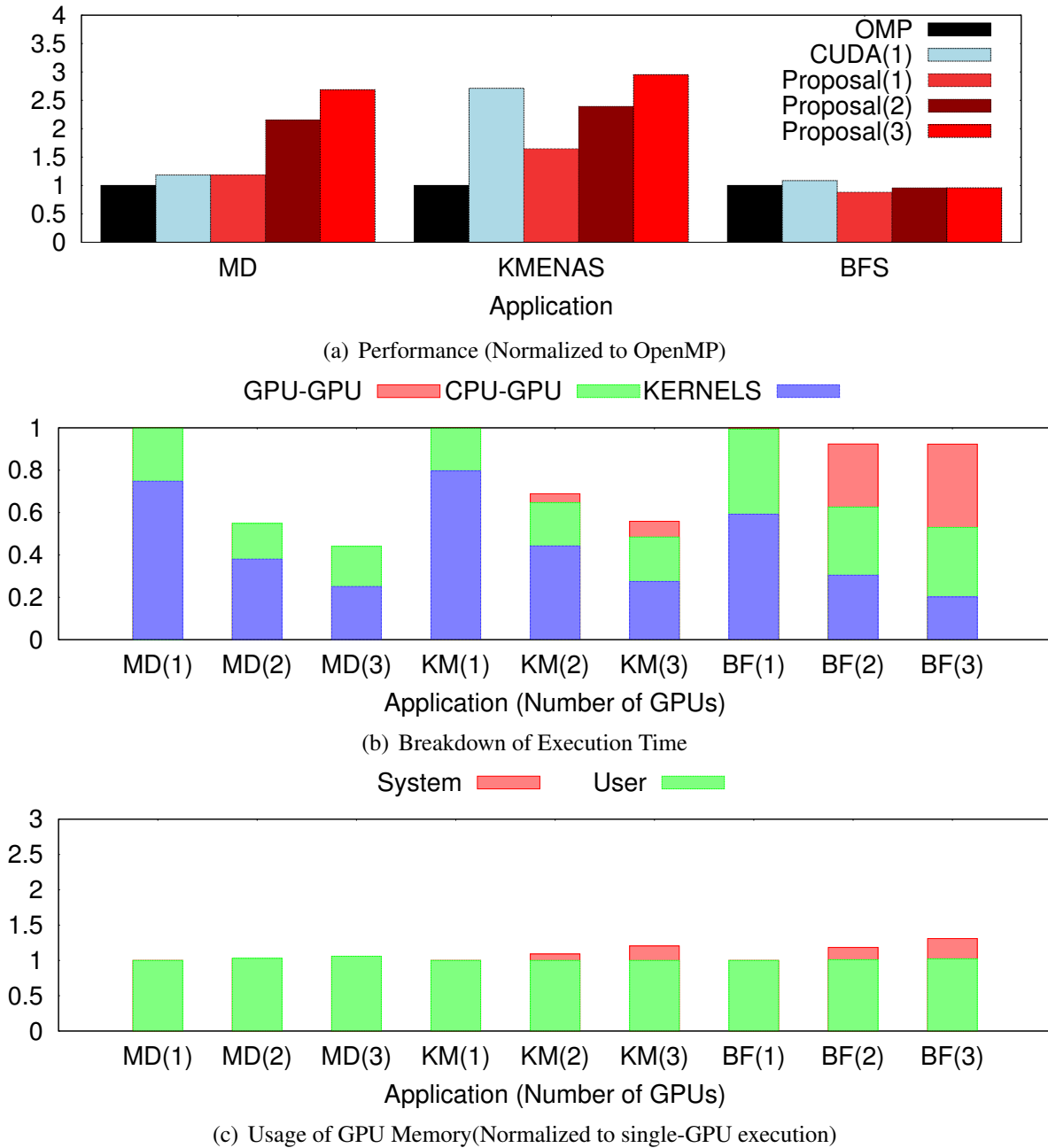


Figure 3.10: Evaluations in TSUBAME2.0 thin-node (2CPU-3GPU).

Chapter 4

Compiler Based Sleep Control in CPU Functional Units

In the case when we offload all data parallel tasks to accelerators, CPUs have less tasks to do. As a result, the utilization of CPUs gets lower than that in conventional homogeneous systems. However, CPUs still have to be active in order to execute tasks which cannot be accelerated by accelerators, such as I/O operations, legacy software libraries, or sequential processing on complex data structures. Such CPUs still consumes large amount of power due to leakage power consumption of CMOS circuits. To eliminate this inefficiency, we have to develop techniques to aggressively reduce leakage power wasted in the CPUs.

4.1 Drawbacks of Conventional Sleep Control Techniques

Conventional sleep control techniques for run-time PG intended to detect idle phases in which the target circuit is rarely used. This results in almost ideal leakage power reduction in the case that the target functional unit is rarely used through the entire application execution. However, if fine-grained idle periods have a large amount of fraction in the total execution time, conventional sleep control techniques fails to reduce the leakage power. And this is often observed in CPU functional units during the execution.

To quantify the amount of the inefficiency of conventional sleep control in CPU functional units, we consider the difference between the ideal leakage power with ideal sleep control(*opt*)

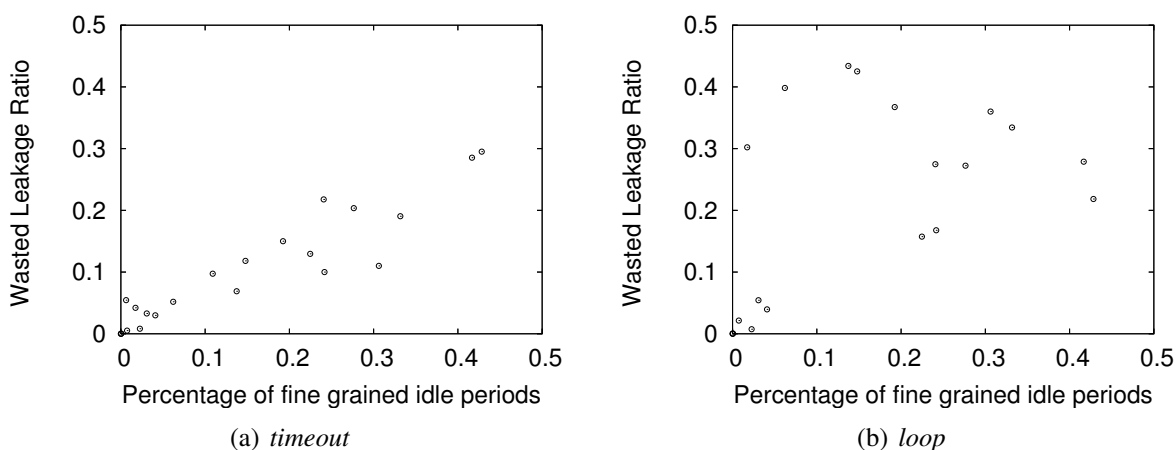
and realistic leakage power (*real*) achieved by a realistic sleep control technique in the following equation: $WLR = 100 * (L_{real} - L_{opt}) / L_{nosleep}$. *WLR* stands for *Wasted Leakage energy Ratio*. L_{opt} , L_{real} , $L_{nosleep}$ denote leakage power consumed with ideal sleep control, with a realistic sleep control, and without sleep respectively. Note that *WLR* is better if the value is lower because it means that the difference from ideal sleep control is smaller.

Table 4.1 gives the evaluated values of *WLR* in conventional sleep control techniques for several applications in SPEC CPU 2000 FP Benchmark Suite. In the table, *timeout* is the sleep control proposed by Hu[25] and *loop* is the sleep control proposed by Roy[49]. The value is obtained via the cycle accurate processor simulator used in the experiments described in the later evaluation in this Chapter. Table 4.1 tells us that a large amount of potential leakage power reduction is left even when runtime PG with conventional sleep controls is used.

The source of this inefficiency is that the fine grain idle periods whose lengths are in the same order to the BET. Figure 4.1 shows the correlation between the fraction of fine grain idle periods in total execution cycles and the *WLR*. In the figures, the y-axis denotes the values of *WLR* showed in the Table 4.1. The x-axis denotes the fraction of fine grain idle periods in total execution cycles. Here, the fraction of fine grain idle periods is defined as the idle periods whose length are shorter than 2x of the length of BET. Figure 4.1 shows there is strong positive correlation between the fraction of fine grain idle periods and the amount of inefficiency in conventional sleep control techniques. In short, conventional sleep control techniques fail to reduce the leakage power consumed during the fine grain idle periods.

In this chapter, we propose a sleep control technique in CPU functional units which can reduce leakage power consumed during such fine grain idle periods. To achieve large amount of leakage power reduction in fine grain idle periods, we developed a sophisticated idle length prediction technique based on compiler.

application	FPALU		FPMULT		INTMULT	
	loop	timeout	loop	timeout	loop	timeout
ampp	1	1	2	1	0	0
apsi	27	22	15	13	28	29
art	4	3	5	2	0	0
equake	27	20	16	10	0	0
mesa	36	11	43	7	40	5
mgrid	8	10	59	10	67	5
swim	22	30	37	15	0	0
wupwise	31	19	43	12	30	4

Table 4.1: Wasted leakage energy ratio(WLR) of conventional techniques (% , $BET=20$ cycle).Figure 4.1: Correlation between fraction of fine grained idle periods and WLR ($BET=20$ cycles).

4.2 Sleep Control Based On Precise Analysis of Idle Length

Our proposal is based on an observation that the fine grain idle periods is mainly originated in instruction sequences and cache misses. The most important factor is instruction sequences, which determine usage pattern of functional units statically. Therefore, our proposed sleep control technique is based on the static compiler analysis to predict the length of each idle period in cycle-level accuracy. Second factor is cache misses which cause the processor pipeline stalls. Because cache misses are dynamically determined, it is not easy to predict them by compilers. Our solution is to combine a compiler-based technique with an existing cache miss based sleep control technique[51], in which hardware detects last level cache misses and turn off functional

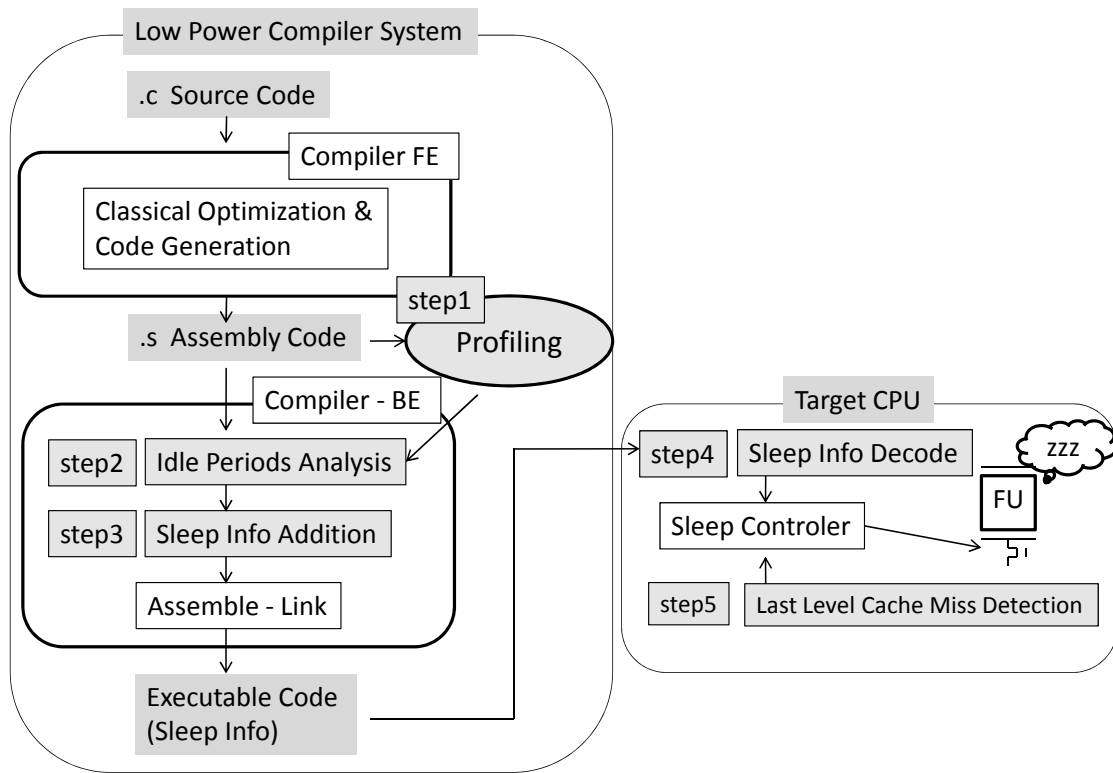


Figure 4.2: An overview of the proposed sleep control system.

units at that moment. The cache miss based sleep control technique requires little hardware to implement. Thus, it does not affect the advantage of static sleep control technique that it requires little changes in hardware.

Figure 4.2 gives an overview of the proposed sleep control technique. In the compiler system, object codes are generated as usual. The object codes are analyzed in order to predict the length of idle periods at link time (Step 2). In this analysis step, the compiler determine predicted lengths of idle periods after every instructions. And it uses the predictions to determine where to inject sleep control information into the final machine code (Step 3). To increase the accuracy of the prediction, we can utilize profiling information about the branch probability of each branch instruction (Step 1). The target CPU dynamically decodes the sleep control information injected by compiler and turn off functional units (Step 4). Also, a simple cache miss detector is added to the target CPU to turn off functional units when the last level cache misses occur(Step 5).

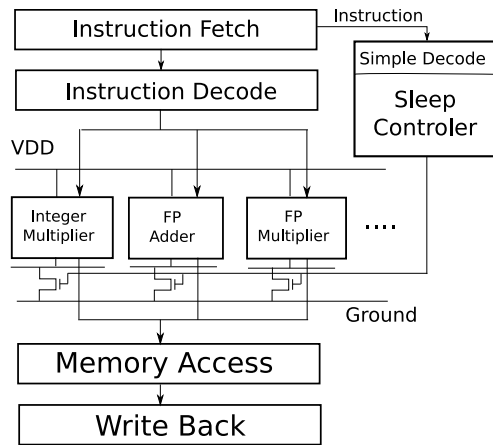


Figure 4.3: Architectural support for run-time PG in functional units.

4.2.1 Architecture Support for Compiler Based Sleep Control

Here, we show an overview of the CPU architecture to support the proposed runtime PG control in functional units. Figure 4.3 shows our base processor architecture assumed in this chapter. The processor architecture is based on the prototype CPU reported by Seki et al. [51]. The processor is a simple in-order processor intended for embedded use. We can turn on and off each functional unit in extremely fine grained manner. The mode transition takes only tens of nano seconds. Note that the latency of the sleep/wakeup mode transitions can be hidden with some architectural techniques [51]. Therefore, we assume the performance overhead of the runtime PG in functional units is negligible and we focus on the energy overhead management of the PG in functional units.

4.2.2 Code Analysis to Detect Fine-Grained Idle Periods

To achieve a large amount of leakage power reduction, we must predict the length of each idle period accurately. In this subsection, we show the detail of our static analysis technique which can predict the lengths of fine grain idle periods with cycle-level accuracy.

Table 4.2 shows a simple illustration of the prediction algorithm. In the figure, the first multiply instruction is executed. The second multiply instruction is executed after three instructions. Assuming a single issue in-order processor, the length of idle periods in the multiplier between

Opcode	Latency	Idle Period
mult	3 cycle	-
store	1 cycle	1
load	2 cycle	3
add	1 cycle	4
mult	3cycle	-

Table 4.2: Idle length prediction in a basic block.

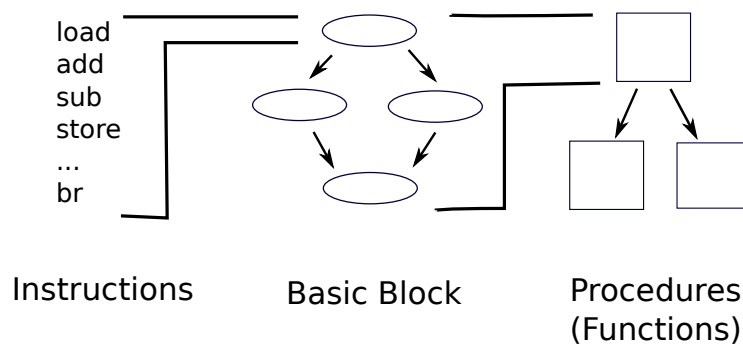


Figure 4.4: Structures in programs

the first and the second multiply instructions is predicted as the sum of execution cycles of intermediate three instructions. The model can be extended with ILP prediction techniques in case that target CPU is a multi-issue processor. Based on the algorithm inside each basic block, we extend the idle length analysis for inter-basicblock and inter-procedural analysis. Figure 4.4 illustrates the structure observed in programs. By considering the structure, the prediction becomes more accurate.

Next, we formalize an intra-procedure analysis techniques to predict the length of idle periods in functional units. The algorithm can be formalized by using data-flow analysis framework[7]. The process collects the information of the utilization of functional units at various points in a program. To gather the data-flow information, we define $idle_{in}[s]$, $idle_{out}[s]$, and $pnuExit_{in}[s]$, $pnuExit_{out}[s]$ for each instruction s . An unusual thing in the data flow analysis is that data flow variables are not boolean variables as usual. To identify the lengths of idle periods, the data flow variables are defined as real number variables. The variables $idle_{in}[s]$, $idle_{out}[s]$ are the average lengths of idle

periods from the points before or after an instruction s . The variables $pnuExit_{in}[s]$, $pnuExit_{out}[s]$ are the average probabilities of reaching the exit point of the procedure without using the target functional unit from the points before or after an instruction s .

In addition to defining data-flow variables, we define length of the execution cycle $len[s]$ and define the probability of not using the target functional unit during execution $pnu[s]$ for each instruction s as follow.

$$\begin{aligned} len[s] &= \begin{cases} 0 & \text{(if } s \text{ use the FU)} \\ ExecLatency[s] & \text{(else)} \end{cases} \\ pnu[s] &= \begin{cases} 0 & \text{(if } s \text{ use the FU)} \\ ProbNotUseFU[s] & \text{(else)} \end{cases} \end{aligned} \quad (4.1)$$

Here, $ExecLatency[s]$, $ProbNotUseFU[s]$ are the length of execution cycles for an instruction s and the probability of not using the target functional unit during execution of an instruction s respectively. These values can be given based on the target architecture except for the case that an instruction s is a function call instruction. We will explain how to handle a function call instruction for inter-procedural analysis later in this subsection. These values are corresponding to transfer functions in a data-flow analysis framework.

Considering the characteristic of $idle_{in}[s]$, $idle_{out}[s]$ and $pnuExit_{in}[s]$, $pnuExit_{out}[s]$, we define data-flow equations in the following equations.

$$\begin{aligned} idle_{in}[s] &= pnu[s] * idle_{out}[s] + len[s] \\ pnuExit_{in}[s] &= pnuExit_{out}[s] * pnu[s] \\ idle_{out}[s] &= (1 - q[s]) * idle_{in}[s_{suc1}] + q[s] * idle_{in}[s_{suc2}] \text{ (if } s \text{ is a branch.)} \\ idle_{out}[s] &= idle_{in}[s_{suc}] \text{ (if } s \text{ is not a branch.)} \\ pnuExit_{out}[s] &= (1 - q[s]) * pnuExit_{in}[s_{suc1}] \\ &\quad + q[s] * pnuExit_{in}[s_{suc2}] \text{ (if } s \text{ is a branch.)} \\ pnuExit_{out}[s] &= pnuExit_{in}[s_{suc}] \text{ (if } s \text{ is not a branch.)} \end{aligned} \quad (4.2)$$

At the branch instructions, $idle_{out}[s_{branch}]$ is given by an average of $idle_{in}[s_{suc1}]$, $idle_{in}[s_{suc2}]$, where s_{suc1} , s_{suc2} are the successors of a instruction s_{branch} , weighted by the branch probability $q[s_{branch}]$. The branch probability $q[s]$ can be simply set to 0.5, or can be obtained by profiling.

The initial values in the iterative solver are given in the following equations:

$$\begin{aligned}
 idle_{in}^0[s] &= pnu[s] * len[s], \\
 pnuExit_{in}^0[s] &= pnu[s], \\
 idle_{out}^0[s] &= 0, \\
 pnuExit_{out}^0[s] &= 0, \\
 idle_{in}^b[s_{exit}] &= 0, \\
 pnuExit_{in}^b[s_{exit}] &= 1.
 \end{aligned} \tag{4.3}$$

Here, the instruction s_{exit} is the exit node of a procedure. Note that although the data-flow variables are not boolean values but real number values, the convergence of iterative algorithm is guaranteed by linear algebra theory if there is no infinite loop. We show a pseudo program of the data-flow algorithm below:

INPUT Control Flow Graph(CFG), the execution cycle(len) and the probability that does not use a target functional unit(pnu) for each instruction

OUTPUT predicted lengths of idle periods in a target functional unit for every instructions

$idle_{in}[s_{exit}] \leftarrow 0, pnuExit_{in}[s_{exit}] \leftarrow 0$

for each node $s \in \text{CFG}$ other than s_{exit} **do**

/* set initial data-flow values according to equations 4.3 */

Set initial values of $idle_{in}[s], pnuExit_{in}[s], idle_{out}[s], pnuExit_{out}[s]$

end for

while changes to any $idle_{out}$ or $pnuExit_{out}$ occur **do**

for each node $s \in \text{CFG}$ other than s_{exit} **do**

/* update data-flow values according to equations (4.2) */

update $idle_{in}[s], pnuExit_{in}[s], idle_{out}[s], pnuExit_{out}[s]$

end for

end while

Practical programs consist of a lot of procedures. In the context of idle periods prediction, it makes it difficult for compilers to detect idle periods across multiple procedures. To overcome this problem, we need to extend the data-flow analysis for inter-procedure analysis.

To extend the analysis for inter-procedural one, the transfer function values for a functional call instruction s_{call} is determined by using the information of callee procedures. Considering that we can treat a callee procedure as a large single instruction, we use an prediction value at callee's entrance point for $ExecLatency[s_{call}]$, $ProbNotUseFU[s_{call}]$. This means that, let $s'_{entrance}$ is the entrance node of the callee procedure, the value of $ExecLatency[s_{call}]$ is set to a value of $idle_{in}[s'_{entrance}]$ and the value of $ProbNotUseFu[s_{call}]$ is set to a value of $idle_{in}[s'_{entrance}]$. This is expressed in following mathematical equations:

$$\begin{aligned} ExecLatency[s_{fcall}] &= idle_{in}[s'_{entrance}], \\ ProbNotUseFU[s_{fcall}] &= pnuExit_{in}[s'_{entrance}]. \end{aligned} \quad (4.4)$$

To use callee's information, the analysis for the callee procedure must have been done before analysis for the caller procedure. We use *Region-Based Analysis* [7] to determine an analysis order of procedures in the same program in order to pass the information correctly between procedures.

The inter-procedural analysis is divided into three steps. First, we construct a call graph and reduce its strongly connected components into single nodes. Second, we determine a first visiting order of procedures through depth-first traversal on the reduced call graph. According to the first visiting order, we determine the data-flow values $idle$, $pnuExit$ at their entrance node $s_{entrance}$. If a visited node have multiple procedures (strongly connected components in a original call graph), we treat multiple procedures as one procedure. Finally, we determine the final data-flow values for every point in every procedure. The final visiting order is the inverse of the first visiting order. In the analysis for each procedure, the values of $idle_{in}^b[s_{exit}]$ and $pnuExit_{in}^b[s_{exit}]$ are set to the average values of $idle$ and $pnuExit$ of the entrance nodes in the procedures which call the procedure.

4.3 Evaluation

In this section, we evaluate the sleep control technique through cycle-accurate processor simulations. Simulations are based on SimpleScalar [10]. We add the functionality of the run-time PG and the cache-miss based sleep mechanism in the simulator. Detailed parameters of the processor model are shown in Table 4.3. In addition to the hardware simulation system, we build an experimental compiler as a back-end system of gcc. It analyzes object codes and insert sleep information into them. We use sleep bit mechanism [51] as a software-hardware interface for communicating sleep information. We turn on the sleep bit at each instruction whose idle length is predicted to be longer than BET. Also, we utilize profile information to determine branch probability $q[s]$ for each branch instruction.

We evaluate the results using 8 benchmarks from SPEC CPU2000 FP Benchmark Suite [24]. These benchmarks are chosen because they exhibit a large amount of fine grain idle periods in functional units. For each program, we skip the first 1 billion instructions and simulate the following 200M instructions. We use reference input set for the simulation. We evaluate leakage power (including energy overhead of mode-transitions) in three functional units, floating point ALU (FPALU), floating point multiplier (FPMULT) and integer multiplier (INTMULT). A key metric in the evaluation is normalized leakage power in functional units. This refers to a ratio of the power of the functional unit with a certain sleep control technique compared to the original leakage power without any runtime PG.

In addition to evaluating the proposed technique, we evaluate two conventional sleep control techniques. In the results, *proposal*, *loop*, and *timeout* refer the proposed technique, the loop based sleep control technique proposed by Roy[49], and time based sleep control technique proposed by Hu[25] respectively. In the time based sleep control, the value of BET is used for the value of sleep threshold. In this evaluation, conventional techniques are combined with a cache miss based sleep control technique because a last level cache miss based sleep always results in leakage power saving in the simulation setup, where memory access latency is longer than BET, thus a cache miss based sleep control technique can be safely combined with conventional sleep

ISA	PISA
Issue	In-Order
Branch Predictor	bimodal(2K Entry)
Num. Functional Unit	
-Int ALU	1
-(load/store)	1
-Integer Multiplier	1
-FP ALU	1
-FP Multiplier	1
L1 I/D cache	32KB 2-way 1-cycle latency
L2 unified cache	1MB 8-way 6-cycle latency
Memory Latency	100 cycle
Break Even Time	20, 40 cycle

Table 4.3: Simulation setup.

control techniques.

Figure 4.5 - Figure 4.10 show the results. Figure 4.5, Figure 4.7, Figure 4.9 are the results in the case that BET is 20-cycle. Figure 4.6, Figure 4.8, Figure 4.10 are the results in the case that BET is 40-cycle. In each graph, each application has three bars according to three sleep control techniques. The rightmost bar is the proposal.

A notable result is that the proposed technique achieves the best leakage power saving in all cases. The result tells us that we can achieve a large amount of leakage power saving in a wide range of applications through the compiler based sleep control. The *proposal* techniques achieve up to 19% more leakage power saving compared to the *timeout* sleep control technique and up to 67% more leakage power saving compared to *loop* sleep control technique. In the applications which use the target functional units most frequently, *apsi*, *mgrid*, *swim*, and *wupwise*, the improvement of leakage power saving by *proposal* is especially high. In these application, the number of total cycles spent in fine grain idle periods is large. In Table 4.4, we show the average leakage reduction from the leakage power in *timeout* sleep control.

BET	FPALU	FPMULT	INTMULT
20 cycles	22.2	24.2	25.1
40 cycles	14.2	21.4	12.7

Table 4.4: Average leakage reduction from *timeout* sleep control (%).

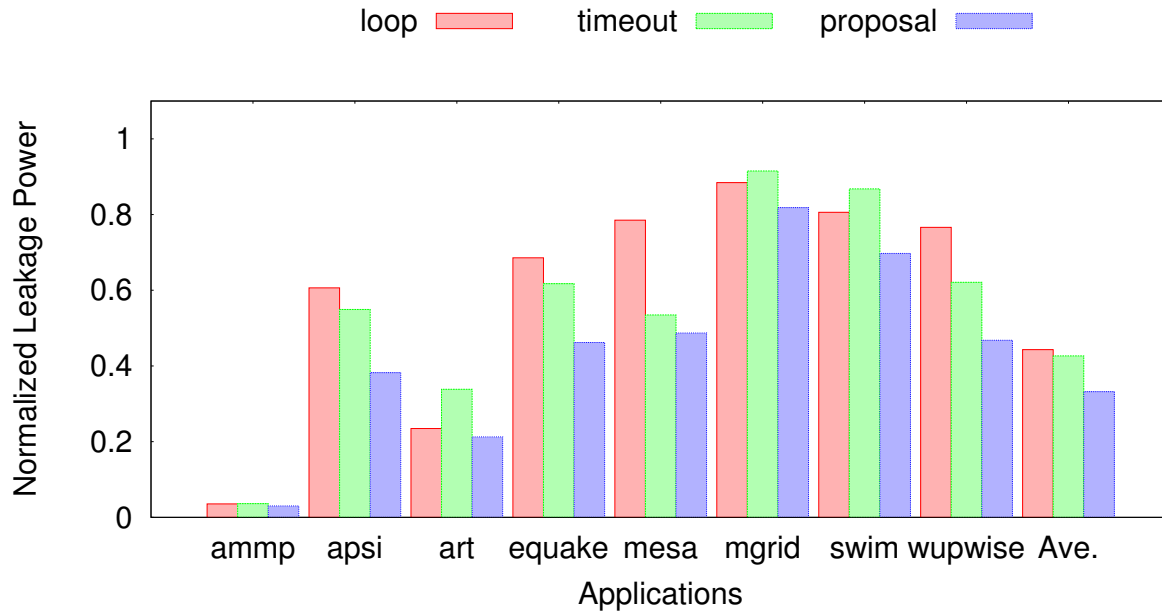


Figure 4.5: Normalized leakage power (BET=20cycle, FPALU).

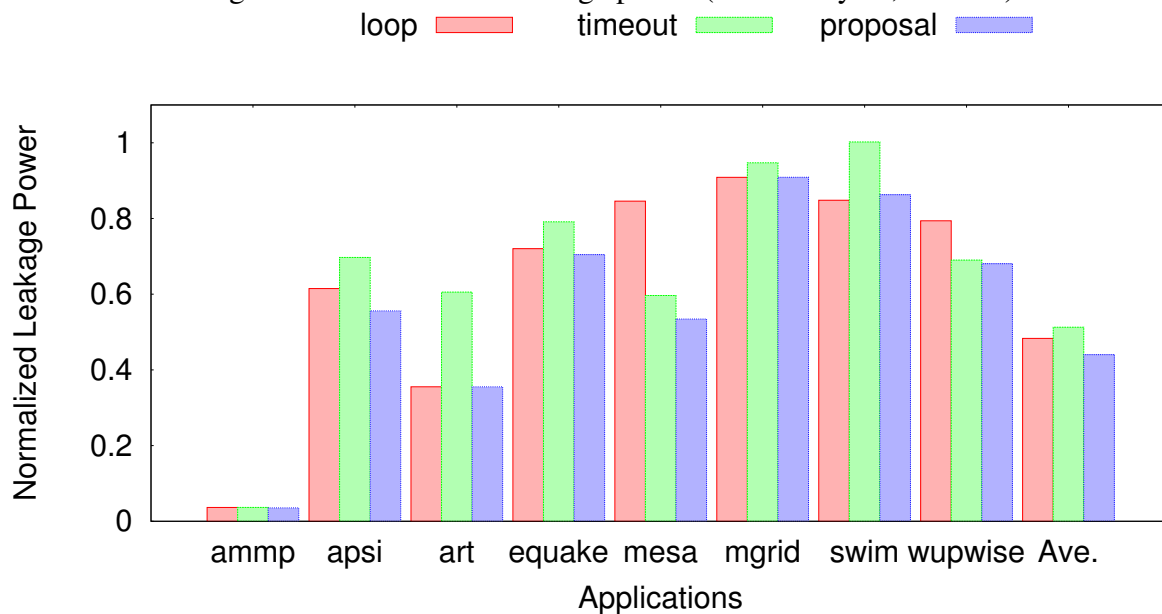


Figure 4.6: Normalized leakage power (BET=40cycle, FPALU).

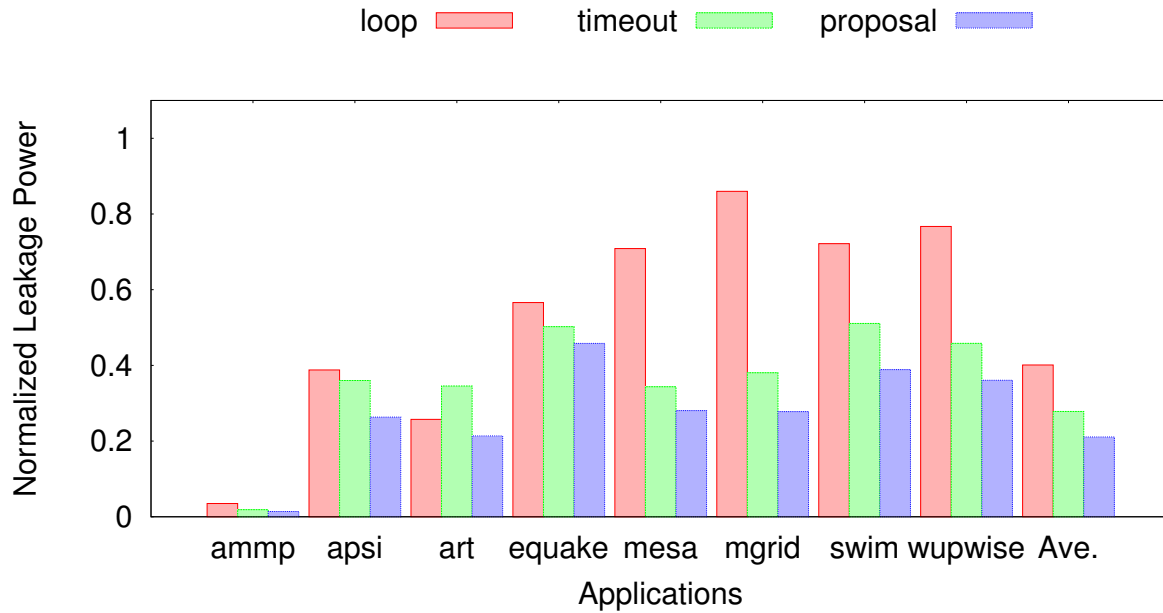


Figure 4.7: Normalized leakage power (BET=20cycle, FPMULT).

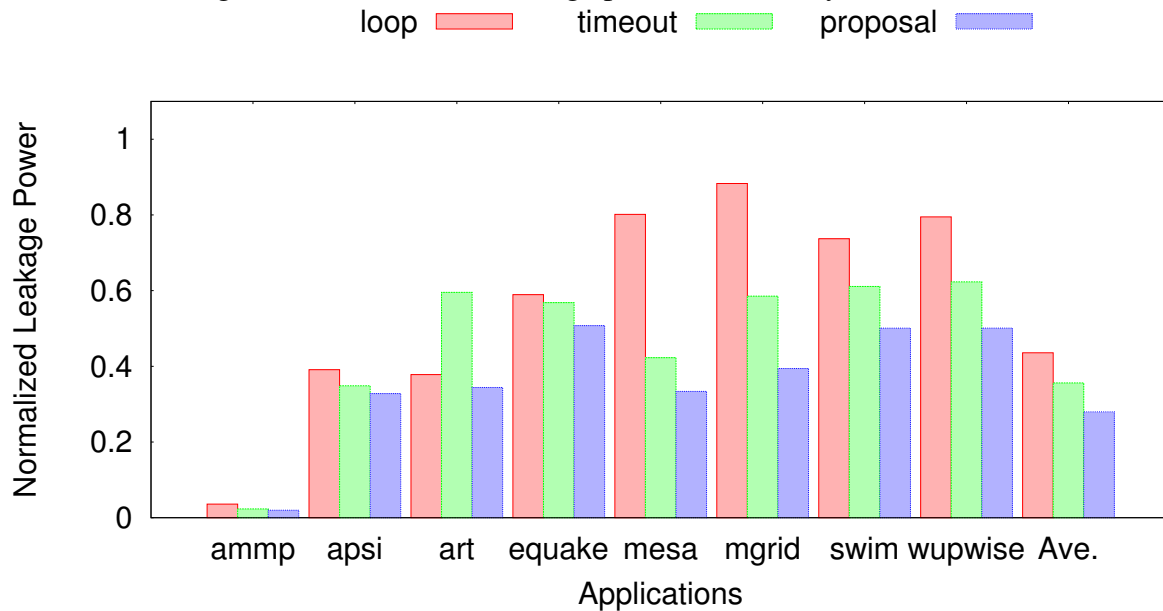


Figure 4.8: Normalized leakage power (BET=40cycle, FPMULT).

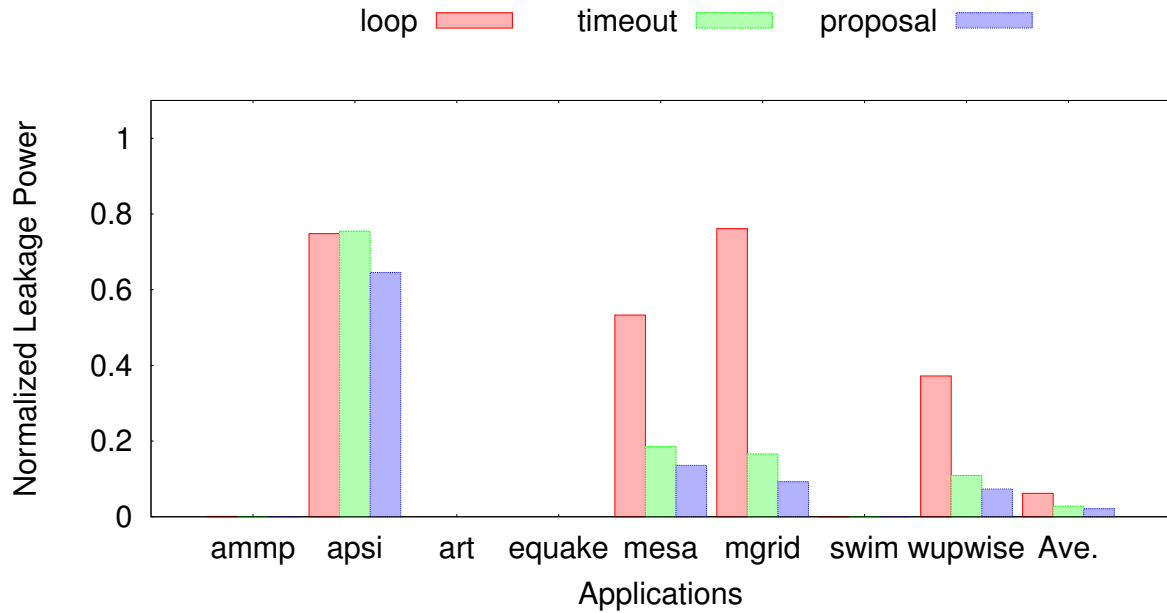


Figure 4.9: Normalized leakage power (BET=20cycle, INTMULT).

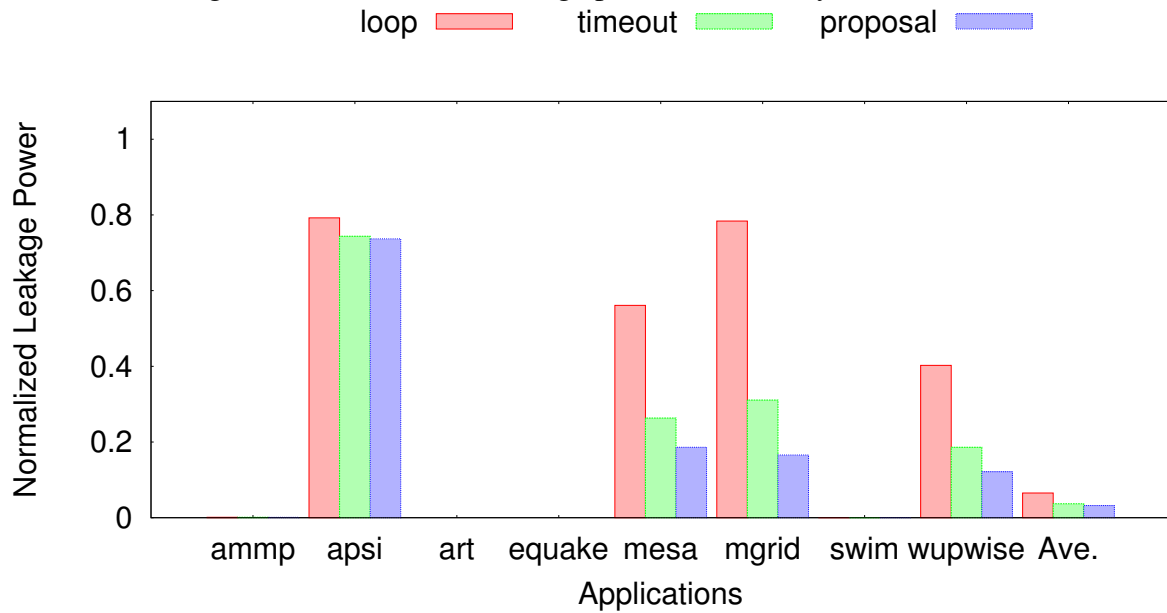


Figure 4.10: Normalized leakage power (BET=40cycle, INTMULT).

Chapter 5

Cooperative DVFS and Heterogeneous Task Mapping

Dynamic voltage and frequency scaling (DVFS) is an effective technique to reduce the dynamic power consumption of processors. Current CPUs and accelerators provide the control knob for softwares to save power consumption. One of the most important problems about the frequency control is that how to minimize the performance degradation when we constrain a system to keep the power under a given power budget. Such power management is called power capping and it is used in large scale computing centers in order to optimize the center-level power utilization [17].

This problem cannot be solved easily in heterogeneous systems because the performance and the power consumption are complicated functions of the task mapping between different computing devices and the frequency settings. To effectively apply DVFS to heterogeneous system with accelerators, it is necessary to develop techniques to guide the settings of the device frequencies and the task mapping cooperatively.

5.1 Drawbacks of Conventional Power Capping Techniques

Hybrid computing is one of the important software optimization techniques in the heterogeneous systems with accelerators [21, 39, 46, 50]. In the hybrid computing, parallel tasks are distributed to both CPUs and accelerators and the tasks are executed in parallel. Figure 5.1 illustrates a simple example of the hybrid computation with a single CPU and a single GPU. In Figure 5.1,

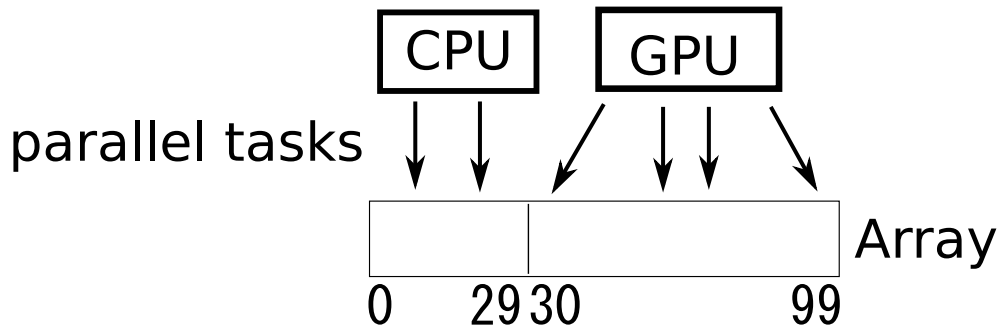


Figure 5.1: Hybrid computation with a CPU and a GPU.

CPU executes 30% items in the array and the rest of 70% items are executed by the GPU. The interesting point is that we can flexibly adjust the percentages of tasks mapped to each device in order to avoid the load imbalance between the computing devices. The optimal task mapping ratios depend on the application characteristics and the hardware organization.

If we directly apply conventional DVFS to such applications in order to set the power budget on the system, the load imbalance between CPUs and accelerators can occur and the performance can be significantly degraded. This is because the amounts of the performance fluctuations by DVFS are different between CPUs and accelerators.

To quantify how large the problem is, in Figure 5.2, we show the performance of such hybrid parallel applications with the conventional DVFS control. The detailed experimental setup is the same as that in the evaluation section. We decrease the frequencies of the CPU and the GPU until the power consumption of the system meets the given power budget. The values of the power budget are shown in the x-axis. The performance is normalized to the performance with the ideal settings of DVFS and task mapping under the given power constraint, in which we can avoid load imbalance caused by DVFS through adjusting task mapping. The result means that the performance degradation can be large if we simply apply DVFS to the CPU and the GPU. This is due to the lack of mechanism to fix the load imbalance caused by applying DVFS.

To this end, to minimize the performance degradation of conventional DVFS in the hybrid parallel applications, it is necessary to develop techniques to guide the settings of the device frequencies and the task mapping cooperatively.

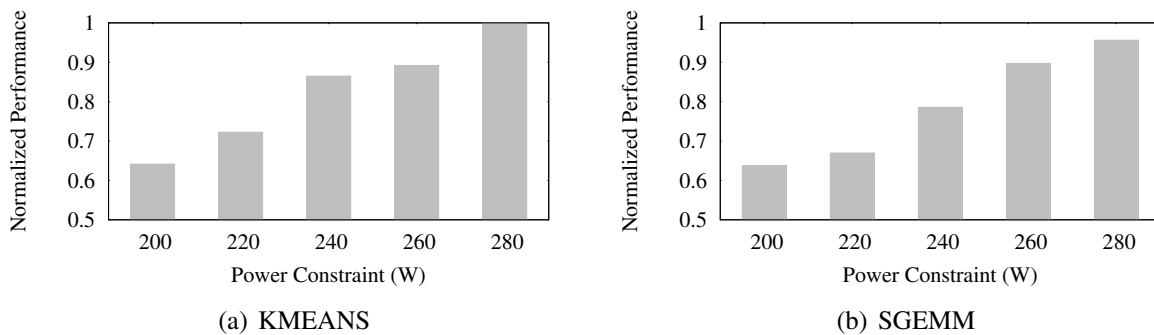


Figure 5.2: Performance with conventional DVFS control. Normalized to the ideal execution without load imbalance.

5.2 Model-based DVFS and Task Mapping

Figure 5.3 shows the overview of the proposed power capping technique. In the proposed technique, we determine the frequencies of CPUs and accelerators, and the task mapping at the beginning of the application execution. The reason why we determine the execution parameters in advance of the execution is that we can avoid performance overhead caused by runtime parameter adjustments. The behavior of a typical data parallel application does not change drastically during the executions because they often use iterative algorithms [50].

To enable us to set the parameters in advance of the application execution, we take profile information at small number of representative parameter points. Based on the profiles, empirical models predict the execution time and the maximum power consumptions for all possible parameter settings. Hence, the models enable us to examine all possible settings of DVFS and task mapping without the exhaustive profiling of the all possible settings of parameters. With the optimal parameters predicted by the model, we execute the application.

It is possible that the system exceeds the power constraint with the pre-determined parameters due to the prediction error of maximum power consumption. In this case, we can adjust the CPU frequency with the existing technique at runtime to prevent the power violation [37]. In Figure 5.3, the *Reactive DVFS controller* is responsible for this.

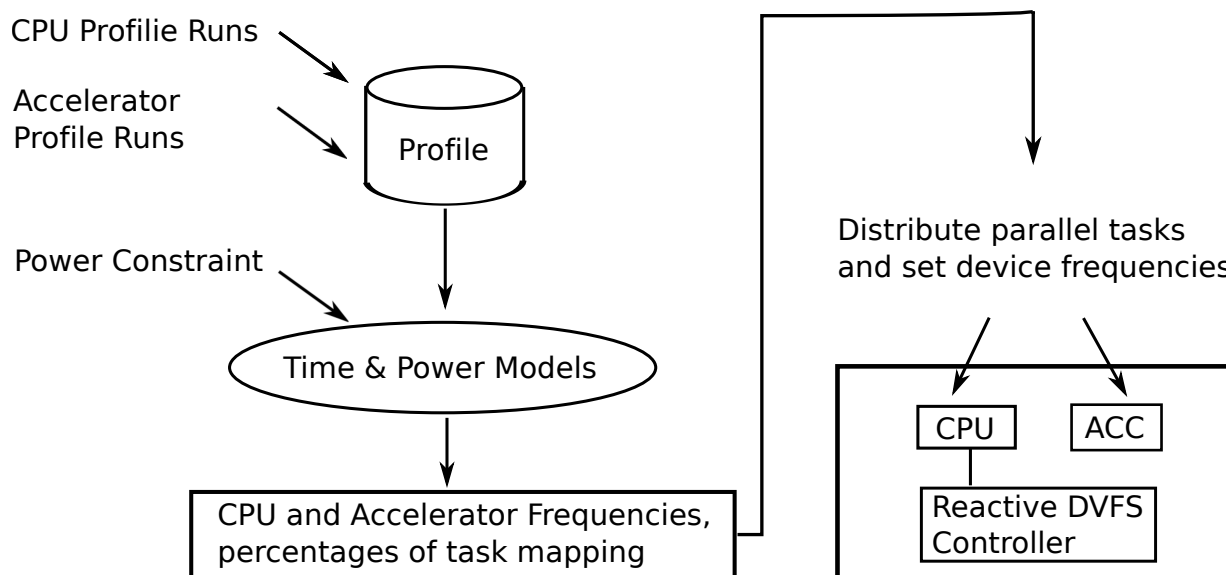


Figure 5.3: An overview of the profile-based power capping.

5.2.1 Model Parameters and Profile Information

The key part of the proposed technique is empirical models of the execution time and the maximum power consumption in the hybrid execution. The models take profiling information about the executions only with the CPU and about the executions only with the accelerator. Using these profiling information, they predict the time and the power of actual CPU-accelerator hybrid executions.

In Table 5.1, we summarize the model parameters and profile information. The control parameters include the frequency of the CPU (f_{cpu}), the frequency of the accelerator (f_{acc}) and the percentages of CPU tasks (r_{cpu}) and accelerator tasks (r_{acc}). The model outputs the normalized execution time of applications (t_{total}) and the maximum power consumption (p_{node}) with the given parameter set: $(f_{cpu}, f_{acc}, r_{cpu}, r_{acc})$. Note that the sum of percentages of CPU tasks and accelerator tasks must be 100: i.e. $r_{cpu} + r_{acc} = 100$. Here, we assume that the system consists of single CPU and single accelerator. However, the proposed method can be easily extended for systems with multiple CPUs and multiple accelerators.

In the profiling runs, we set the percentages of CPU and accelerator tasks to $r_{cpu} = 100, r_{acc} =$

Predicted Values	
t_{total}	Total Execution time
p_{node}	Power Consumption of the Node
Control Parameters	
f_{cpu}	Frequency of the CPU
f_{acc}	Frequency of the accelerator
r_{cpu}	percentage of CPU tasks ($1 - r_{acc}$)
r_{acc}	percentage of accelerator tasks
CPU(accelerator)-only Profile	
$K_i^{c(a)}$	Execution Time of i th Parallel Kernel
$P_{node}^{c(a)}$	Power of the Node
$P_{cpu}^{c(a)}$	Power of the CPU
$P_{acc}^{c(a)}$	Power of the accelerator

Table 5.1: Model parameters and profile information.

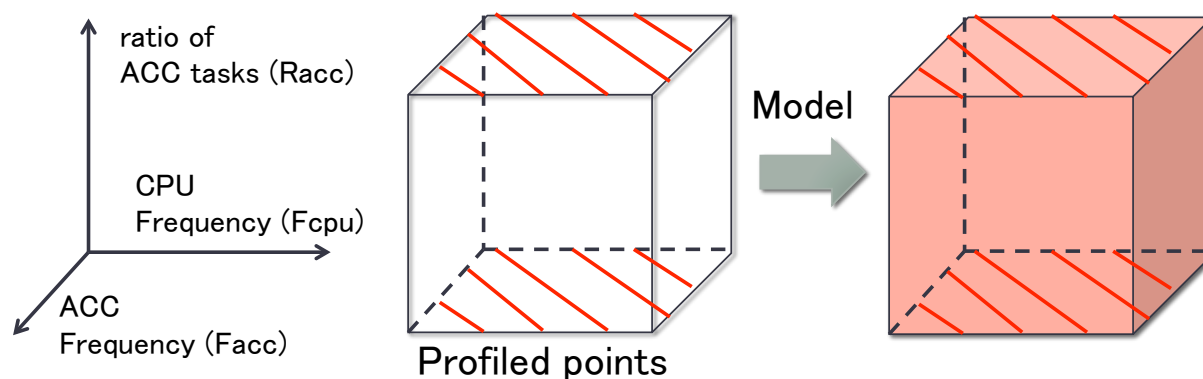


Figure 5.4: Parameter space represented as a three-dimensional cube.

0 (CPU-only profile) and $r_{cpu} = 0, r_{acc} = 100$ (accelerator-only profile). With the settings of task mapping, we profile the execution time and the maximum power consumption for all possible set of device frequencies. Figure 5.4 illustrates the parameter space as a three dimensional cube. The profiled parameter points are at the ceiling and the bottom of the cube. The models predict the time and the power for all the other parameter points inside the cube.

In the profiling runs, we collect the execution times of each parallel kernel function. The execution times are normalized to the execution time of the execution only with the accelerator at the highest frequency. Thus, the model is applicable to executions with different data whose

sizes are different from that of the data used in the profiling runs. The profile information also includes the power consumption of the CPU, the accelerator, and the computing node. We use hardware power counters which is already integrated into commercial CPUs and accelerators to measure the power consumption of CPUs and accelerators.

Scenario for Obtaining the Profile If we can use typical input data and the actual computer system used in the real situation during the testing and tuning phase of the application development, it is not problematic to get the profiling information. For high performance computing applications, it is often the case that we can access the typical input data and the computer system during the application development. If we cannot do that, application users have to prepare the typical input data and have to profile the application when the application is installed in the system. Although some parts of the process can be automated, this scenario adds the complexity of the installing process. Also, in both two scenarios, it is possible that the size of data used in the profiling runs differs from the size of the actual input data. Considering this case, we will evaluate the sensitivity on input data size in the later evaluation.

5.2.2 Empirical Models

The total execution time of the application can be modeled by the sum of the execution time of the parallel kernel functions as the following equation:

$$t_{total} = \sum_i K_i(f_{cpu}, f_{acc}, r_{cpu}, r_{acc}).$$

The execution time of the i -th kernel is represented as $K_i(f_{cpu}, f_{acc}, r_{cpu}, r_{acc})$.

To estimate the execution time of the i -th kernel (K_i) in hybrid computations, in which both the r_{cpu} and the r_{acc} are not zero, we make the following two assumptions: 1. the actual execution time on the CPU or the accelerator is proportional to the amount of mapped tasks, and 2. global barrier is required after the execution of each parallel kernel. The global barrier is illustrated in Figure 5.5. In this situation, the faster device must wait for the other device at the synchronization point. Using the two assumptions and the profile information, we can estimate the execution time

of the i -th kernel with the following equation:

$$K_i = \max \left\{ K_i^c(f_{cpu}, f_{acc}) \times \frac{r_{cpu}}{100}, K_i^a(f_{cpu}, f_{acc}) \times \frac{r_{acc}}{100} \right\}.$$

Note that we treat data transfers between a CPU and an accelerator as a task which is mapped to both of the CPU and the GPU. In this case, $r_{cpu} + r_{acc} \neq 100$. With this assumption, we can use the same model for data transfers.

The total node power $p_{node}(f_{cpu}, f_{acc}, r_{cpu}, r_{acc})$ can be decomposed into three terms as the following equation:

$$p_{node} = p_{idle}(f_{cpu}, f_{acc}) + p_{cpu}(f_{cpu}, f_{acc}, r_{cpu}) + p_{acc}(f_{cpu}, f_{acc}, r_{acc}).$$

The term p_{idle} represents the idle power consumption, which depends on the device frequencies but is independent of the task mapping. It includes static power of the system. The terms p_{cpu} and p_{acc} represent power consumptions in the CPU and the accelerator, which depend on all the control parameters $(f_{cpu}, f_{acc}, r_{cpu}, r_{acc})$.

First, we estimate the values of the idle power in the CPU-only profile and the accelerator-only profile (denoted as p_{idle}^c and p_{idle}^a respectively). We subtract the dynamic power consumption of the device from the total power consumption of the node as the following equation:

$$p_{idle}^{c(a)} = P_{node}^{c(a)}(f_{cpu}, f_{acc}) - \left\{ P_{cpu(acc)}^{c(a)}(f_{cpu}, f_{acc}) - P_{cpu(acc)}^{a(c)}(f_{cpu}, f_{acc}) \right\}.$$

The dynamic power consumption of the device is approximated by the expression $P_{cpu(acc)}^{c(a)} - P_{cpu(acc)}^{a(c)}$. The values of p_{idle}^c and p_{idle}^a include the static power of CPUs, the static power of accelerators, and the power consumption of the all other components (mother board, hard disk etc.). In the power model, the total idle power is linearly interpolated by using these values as the following equation:

$$p_{idle} = p_{idle}^c(f_{cpu}, f_{acc}) \times \frac{r_{cpu}}{100} + p_{idle}^a(f_{cpu}, f_{acc}) \times \frac{r_{acc}}{100}.$$

The CPU and the accelerator are not always busy because fine-grained synchronization is required as illustrated in Figure 5.5. Because the activities of the devices change according to

the task mapping, the values of device power consumption p_{cpu} and p_{acc} also fluctuate. To model this, we assume that the power consumptions are proportional to the fraction of busy time to the total execution time. This relationship is expressed in the following equation:

$$\begin{aligned} p_{cpu} &= \alpha_{cpu} \times \{P_{cpu}^c(f_{cpu}, f_{acc}) - P_{cpu}^a(f_{cpu}, f_{acc})\}, \\ p_{acc} &= \alpha_{acc} \times \{P_{acc}^a(f_{cpu}, f_{acc}) - P_{acc}^c(f_{cpu}, f_{acc})\}. \end{aligned}$$

The term α_{cpu} and the term α_{acc} represent the fraction of busy time in the CPU and that in the accelerator respectively. In the equations, the expression $P_{cpu}^c - P_{cpu}^a$ and the expression $P_{acc}^a - P_{acc}^c$ are used as the approximations of the maximum amount of the dynamic power consumption in the CPU and that in the accelerator respectively.

To estimate the fraction of busy time α_{cpu} and α_{acc} from the profile information, we define the busy time of each device during execution as the sum of the actual execution time of each kernel function. The definition is shown in the following equation:

$$t_{cpu(acc)} = \sum_i K_i^{c(a)}(f_{cpu}, f_{acc}) \times \frac{r_{cpu(acc)}}{100}.$$

The busy times are denoted by t_{cpu} and t_{acc} . Using them, we estimate the fraction of busy time of the CPU α_{cpu} and that of the accelerator α_{acc} as the following equations:

$$\alpha_{cpu(acc)} = \frac{t_{cpu(acc)}(f_{cpu}, f_{acc}, r_{cpu(acc)})}{t_{total}(f_{cpu}, f_{acc}, r_{cpu(acc)})}.$$

5.2.3 Parameter Selection

After taking the profiling information, we examine all possible sets of parameters with the empirical models and build an optimal parameter tables. The table keeps the list of the optimal parameter sets for every possible values of the power constraint. We have to build the tables once for one profiled data. Plus, it does not take much time to build it even if we explore the optimal parameter sets with exhaustive search algorithms because we can quickly estimate the execution

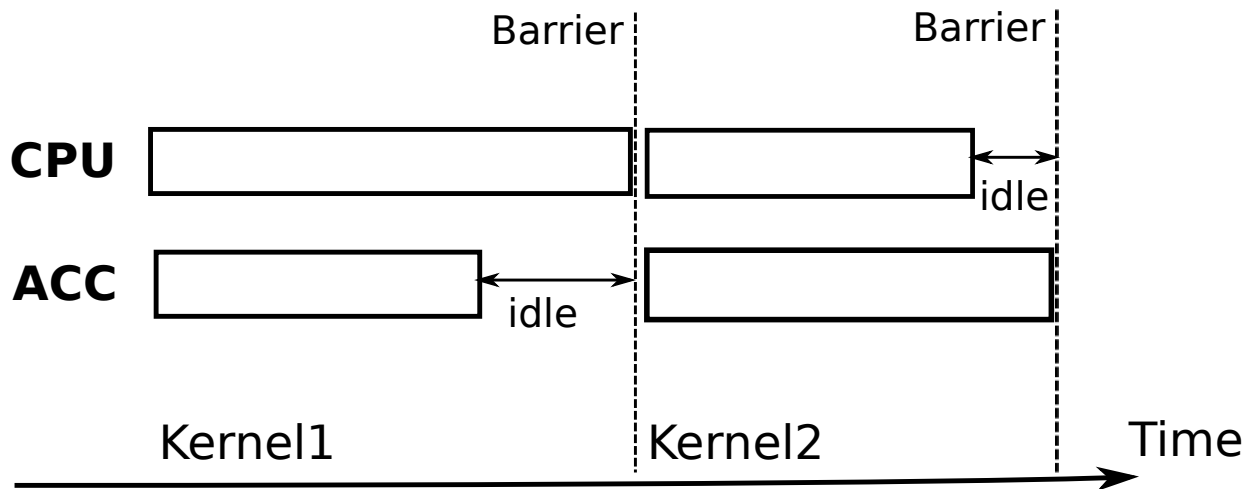


Figure 5.5: Fine-grained synchronization between the CPU and the accelerator.

time and the maximum power consumption at any parameter set with the models. At the beginning of the application execution, the runtime system looks up the table to get the optimal parameter set under the given power constraint.

5.3 Evaluation

In the experiment, we use a computing node with a single multi-core CPU and a single GPU. The details of the platform are shown in Table 5.2. We measure the total power consumption of the machine at every one second with “watts up? .net” [27]. To measure power consumed in the CPU and the GPU in profiling executions, we use RAPL [15] and NVML [2] APIs to read the hardware counters related to the device power consumption. We use Linux “cpufreq-set” command to change the CPU frequencies and use “nvidia-smi” command to change the GPU frequencies.

We evaluate five GPGPU applications selected from rodinia benchmark suite [12] and from BLAS library. The applications are summarized in Table 5.3. Compilers or runtime libraries with the capability of scheduling tasks between CPUs and GPUs have not been publicly available yet. Therefore, we manually implemented a heterogeneous task mapping mechanism for each evaluated application. They are compiled with gcc 4.4 and with nvcc 5.0. We use the “-O3” flag

CPU	Intel Core i7 x1 (6core)
GPU	Nvidia Tesla K20c x1
CPU Frequencies(GHz)	1.2 - 3.2 (0.2 GHz step)
GPU Frequencies(MHz)	614, 640, 666, 705, 754
OS	Linux

Table 5.2: Machine setup for the evaluation.

Application	Description	Input Description
BFS	Graph Traversal	5M node random graph
HOTSPOT	Heat Simulation	(1024, 1024) Grid
KMEANS	Clustering	494020 data items
PF	Model Estimation	50K particles
SGEMM	BLAS Library	(2k, 2k) matrices x100

Table 5.3: Description of the benchmarks.

for the compiler optimization. We enlarge the provided data sets and use them for evaluations.

In the developed hybrid applications, we use the existing parallel codes of rodinia benchmark suite and BLAS library. The parallel codes use OpenMP and SIMD instructions for CPU side computations and they use CUDA for GPU side computations. In the hybrid *SGEMM* implementation, we use Intel MKL for the CPU side computation and NVIDIA CUBLAS for the GPU side computation. In the hybrid executions, the data elements are transferred only when they are really necessary on the GPU at the given task mapping. In the experiment, we assume that we can change the percentages of CPU and GPU tasks with a step of 10%.

5.3.1 Model Verification

In Figure 5.6 and Figure 5.7, we plot the errors in the execution time model and those in the power model. In the figures, the GPU frequency is fixed at 614MHz. The x-axis denotes the percentage of GPU tasks (r_{acc}). We show the result of the percentage of GPU tasks from 50% to 100% because the optimal percentages of GPU tasks are in this range for all the evaluated applications. The y-axis denotes the CPU frequency (f_{cpu}). The z-axis denotes the errors in the proposed model. The execution time is normalized to the execution time of the execution only with the GPU at the highest frequency.

Application	Ave. Time(%)	Ave. Power(W)
BFS	9.18	4.95
HOTSPOT	5.97	3.65
KMEANS	10.41	5.10
PF	14.07	3.49
SGEMM	10.67	1.49

Table 5.4: Average Errors of the Model (the percentage of GPU tasks: 50% - 90%).

We can see that the predicted values are closely fitted to the actual measurement. The average errors for 50%-90% GPU tasks are shown in Table 5.4. The average error of the execution time is up to 14%. The average errors of the maximum power consumption are up to 5.1 Watt.

5.3.2 Performance Under the Power Constraint

Figure 5.8 shows the normalized performance with the proposed power capping techniques. The x-axis denotes the power constraint. The y-axis denotes the performance normalized to the performance of the execution with ideal settings of DVFS and task mapping under the given power constraint. We compare power capping techniques with fixed task mappings to the proposed one. They are: *cpu-dvfs*, the percentage of GPU tasks is fixed at 0%, *gpu-dvfs*, the percentage of GPU tasks is fixed at 100%, *dvfs*, the percentage of GPU tasks is fixed at the optimal percentage without power constraint.

In the region with strict power constraint (200W-240W), the CPU and the GPU must run with lower frequency levels. Because the range of frequency scaling in the CPU is larger than that of the GPU, the performance degradation caused by DVFS tends to be larger in the CPU. This causes load imbalance between the CPU and the GPU in *dvfs*, which directly results in large performance degradation. On the other hand, the performance of *gpu-dvfs* is relatively constant throughout the various power constraints. However, *gpu-dvfs* cannot achieve the optimal performance in any cases. Plus, *gpu-dvfs* fails to run under the most strict power constraint (200W) in *BFS*, *HOTSPOT* and *PF*.

With the proposed technique, we can migrate parallel tasks between the CPU and the GPU with considering the DVFS effects on the performance of the CPU and the GPU. Thus, we can

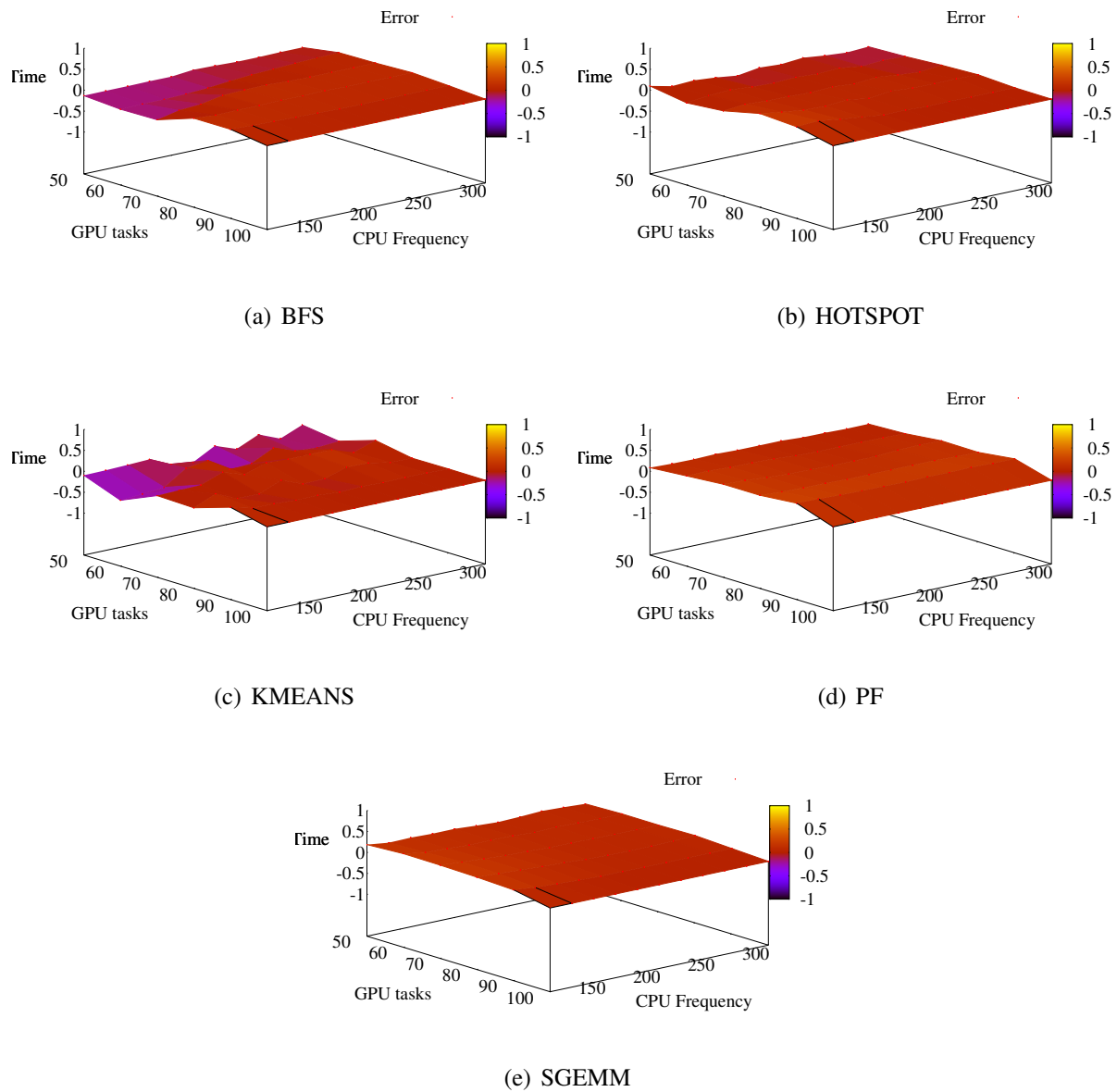
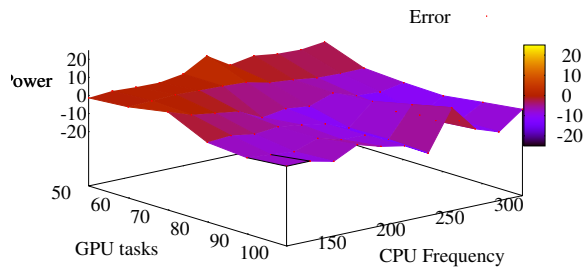
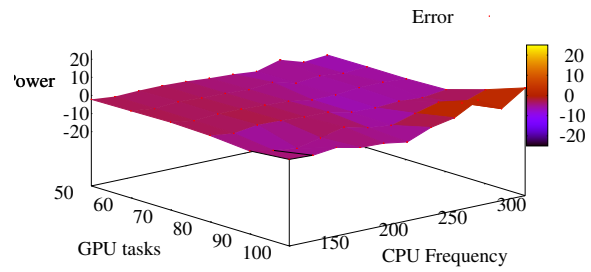


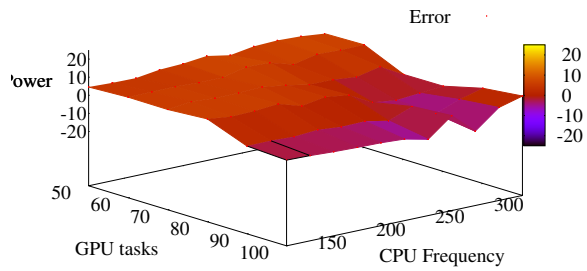
Figure 5.6: Errors in the estimated execution time: GPU 614MHz.



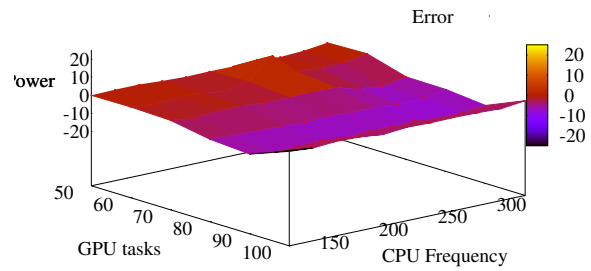
(a) BFS



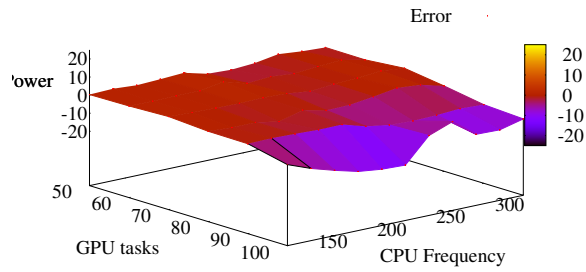
(b) HOTSPOT



(c) KMEANS



(d) PF



(e) SGEMM

Figure 5.7: Errors in the estimated power (watt): GPU 614MHz.

eliminate inefficient execution caused by load imbalance. On average across five applications, the proposed power capping technique achieves more than 97% of performance compared to the ideal performance under all the power constraint. Comparing it to other techniques, it achieves 14.4% higher performance than that of *dvfs* at 200W and it achieves 13.0% higher performance than that of *gpu-dvfs* at 280W.

5.3.3 Sensitivity on Different Input Data

Since the proposed models use the normalized execution times to select the device frequencies and the percentage of CPU and GPU tasks, we can use the same profiles for the execution with data whose size is different. It's worth investigating the sensitivity of the proposed technique on the change in input data size, as the actual input data size is not always the same to that used in the profile runs. In Figure 5.9 and Figure 5.10, we apply the profile information of the input data listed in Table 5.3 for the executions with 2 times larger data (Figure 5.9) and with 4 times larger data (Figure 5.10).

Although we can see small performance degradation for the data 4 times larger than profiled, the proposed technique works well in most cases. This result demonstrates that the proposed technique can be used even in the situation where the input data sizes are not exactly the same to the sizes of profiled data.

However, in some cases, the proposed technique fails to meet the given power budgets. The cases are in *HOSTSPOT* under 200, 220, and 240 watts and in *SGEMM* under 200 watts. The selected parameters by the proposed technique in these cases are ($f_{cpu}=1.2\text{GHz}$, $f_{gpu}=614$, $r_{gpu}=90\%$) for *HOTSPOT* under 200 watts, ($f_{cpu}=1.2\text{GHz}$, $f_{gpu}=666$, $r_{gpu}=100\%$) for *HOTSPOT* under 220 watts, ($f_{cpu}=1.2\text{GHz}$, $f_{gpu}=758$, $r_{gpu}=100\%$) for *HOTSPOT* under 240 watts, and ($f_{cpu}=1.2\text{GHz}$, $f_{gpu}=705$, $r_{gpu}=100\%$) for *SGEMM* under 200 watts. On the other hand, the ideal parameters are ($f_{cpu}=1.6\text{GHz}$, $f_{gpu}=705$, $r_{gpu}=80\%$) for *HOTSPOT* under 200 watts, ($f_{cpu}=1.2\text{GHz}$, $f_{gpu}=614$, $r_{gpu}=100\%$) for *HOTSPOT* under 220 watts, ($f_{cpu}=1.4\text{GHz}$, $f_{gpu}=705$, $r_{gpu}=100\%$) for *HOTSPOT* under 240 watts, and ($f_{cpu}=1.2\text{GHz}$, $f_{gpu}=666$, $r_{gpu}=100\%$) for *SGEMM* under 200 watts. In short, in these problematic cases except for *HOTSPOT* under 200 watts, the proposed

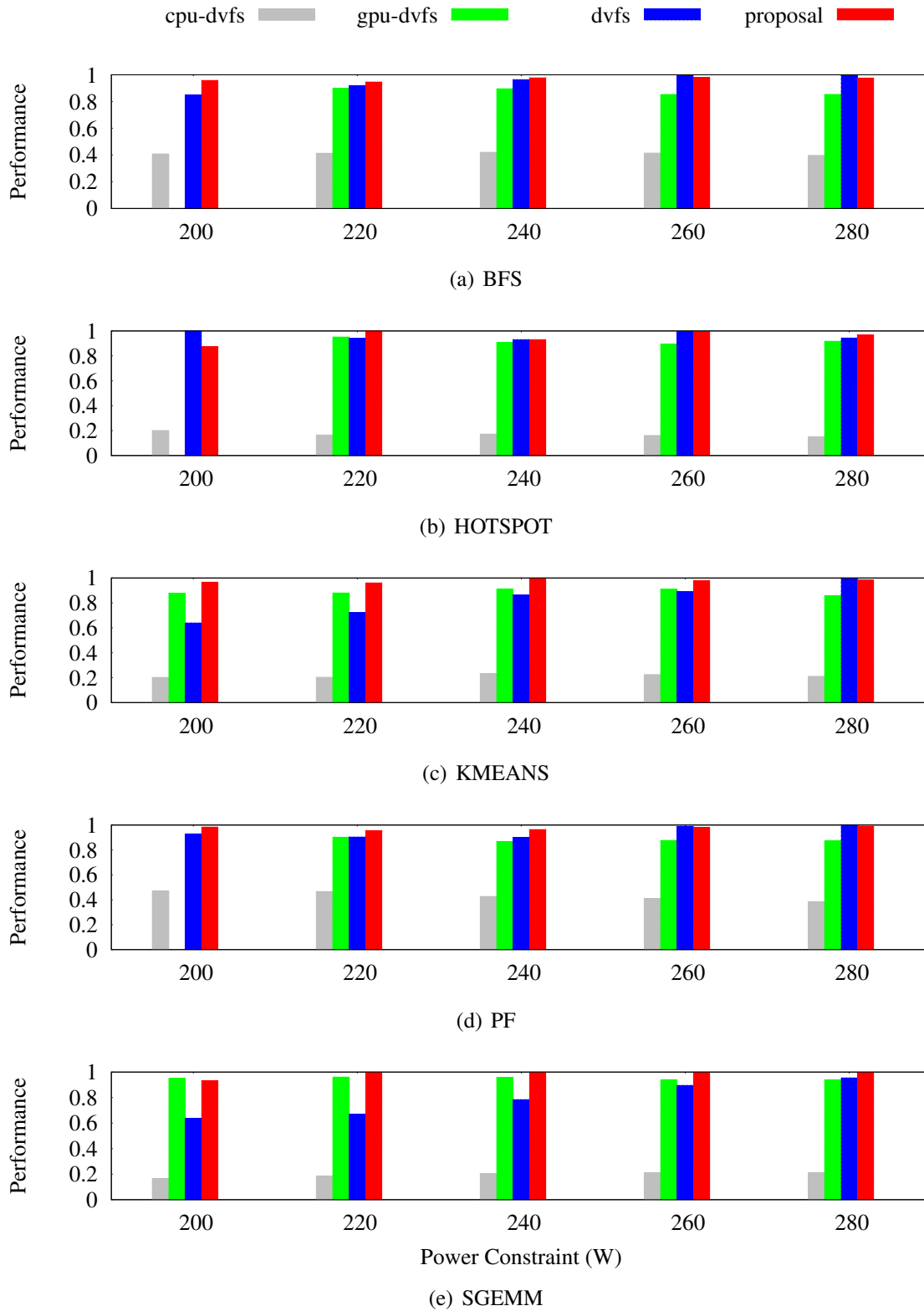


Figure 5.8: Performance under the power constraint. Normalized to the ideal performance for each power constraint.

technique select higher GPU frequencies than the ideal GPU frequencies. The proposed system includes a reactive DVFS controller only on CPUs but does not include a reactive DVFS controller on GPUs. Hence, it could not lower the GPU frequencies to meet the budget when the power violations are detected by the power meter. This problem seems to be easily fixed by adding a reactive DVFS controller on GPUs. Also, the case in *HOTSPOT* under 200 watts is an exceptional case in which we have to migrate tasks from GPUs to CPUs to lower the utilization of GPUs to meet the power budget. This means that, in this case, we have to intentionally cause load imbalance between the CPU and the GPU to lower the power consumption. Hence, adding a reactive DVFS controller on GPUs cannot be the solution in such case. However, such case is rare.

On the other hand, the reason why the proposed technique select the higher GPU frequencies in the problematic cases is that the power models underestimate the power of GPUs with larger data. It seems that the power consumption of the GPU gets larger with the large data. Currently, the proposed models do not include any mechanism to consider such effects of different input data. To predict the power consumption across different data, several previous researches have tried to predict the performance and the power consumption of GPUs when the input data changes [33, 54]. In the previous researches, they use training input data and machine learning algorithms to precisely predict the performance and the power for unknown input data. We may improve the precision of the proposed models by utilizing these prediction techniques.

5.3.4 Optimizing for Energy-Efficiency

The proposed models can be used not only to maximize the performance under the power budget but also to explore the parameter set which achieves the highest energy-efficiency. Here, we use performance per watt as the metric for evaluating the energy-efficiency.

Figure 5.11 shows the performance per watt with the parameters selected by using the proposed models. The baseline in Figure 5.11 is the performance per watt when we choose the parameter sets which can achieve the highest performance (*MaxPerf*). We also show the performance per watt with the ideal parameter sets (*Ideal*). We can see that the proposed models can

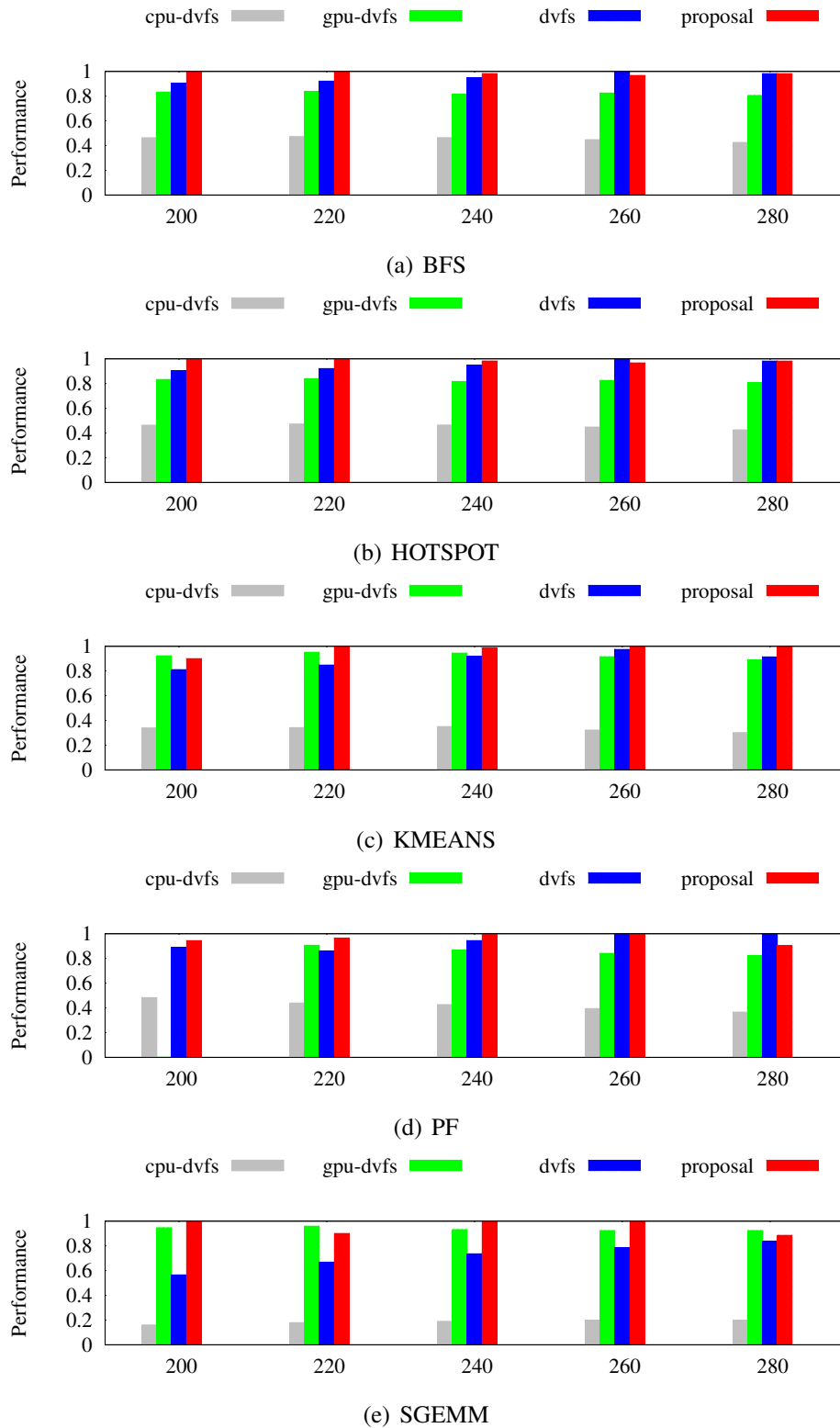


Figure 5.9: Performance with data 2 times larger than profiled. Normalized to the ideal performance for each power constraint.

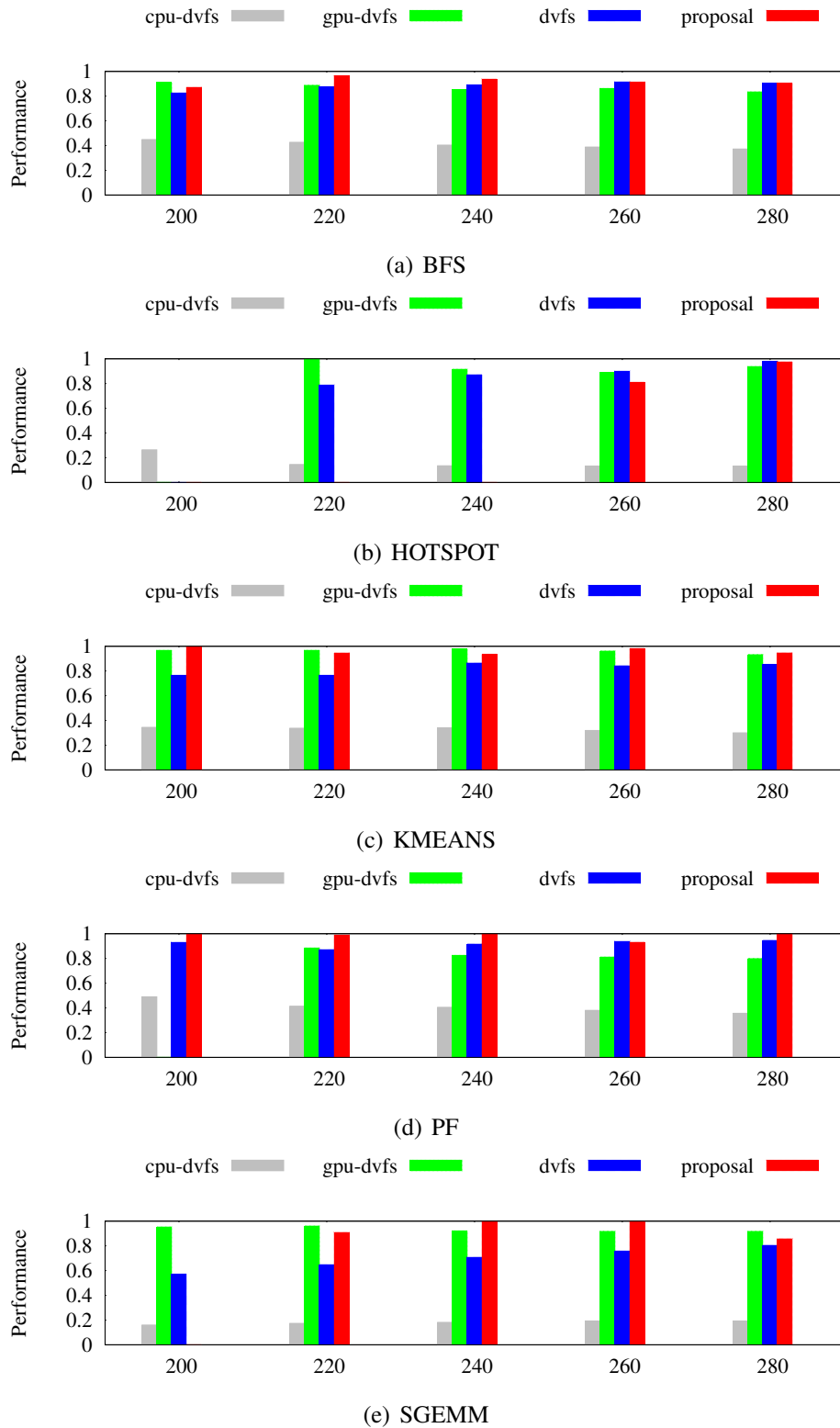


Figure 5.10: Performance for the data 4 times larger than profiled. Normalized to the ideal performance for each power constraint.

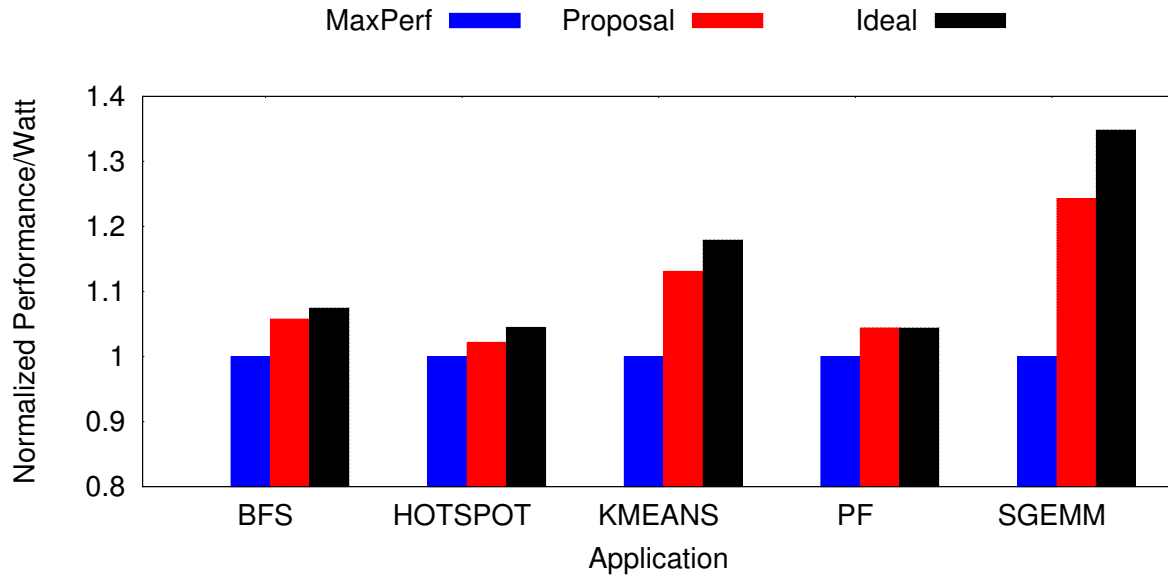


Figure 5.11: Normalized energy-efficiency.

successfully improve the energy efficiency of the computing node.

Also, we show the selected parameter sets in Table 5.5. In *HOTSPOT*, *KMEANS*, *SGEMM*, 100% GPU executions result in the most energy efficient execution. On the other hand, in *BFS*, *PF*, it is not the case because the GPU speedup ratios in these applications are not as high as the previous three applications. Comparing the parameter settings of the *MaxPerf* and the settings of the *Proposal*, a certain amount of tasks are migrated from the CPU to the GPU in *Proposal* and it results in the higher energy efficiency.

Application	CPU Freq	GPU Freq	GPU Task Ratio
BFS(maxperf)	320	758	60
BFS(proposal)	120	705	90
BFS(ideal)	180	758	80
HOTSPOT(maxperf)	280	758	90
HOTSPOT(proposal)	200	705	90
HOTSPOT(ideal)	120	758	100
KMEANS(maxperf)	300	758	80
KMEANS(proposal)	140	705	90
KMEANS(ideal)	120	705	100
PF(maxperf)	320	758	70
PF(proposal)	200	758	80
PF(ideal)	200	758	80
SGEMM(maxperf)	320	758	80
SGEMM(proposal)	160	758	90
SGEMM(ideal)	120	705	100

Table 5.5: Selected parameters in the parameter exploration for the highest energy-efficiency.

Chapter 6

Discussion

Here, we summarize the three techniques proposed in Chapter 3– 5 and discuss cross cutting issues and their extensibility.

Multi-Device OpenACC Compiler The compiler runs an OpenACC program among multiple GPUs. It reduces the programming complexity in utilizing multiple GPUs while it can provide comparable performance to the low level programming environment, such as CUDA. The compiler system includes the novel memory manager to handle inter-GPU communications efficiently. In addition to proposing the basic design, we propose a small set of new directives as extensions to the OpenACC API. They allow programmers to express detailed memory access patterns in parallel loops. The information given by the directives enables the system to optimize the data movement among distributed GPU memories. With the proposed compiler, numbers of code lines to utilize multiple GPUs are significantly reduced. Experiments on a machine with 2 GPUs show that the proposed compiler reduce 68% of the code size in utilizing 2 GPUs while it also achieves 71% of the performance compared to hand-written CUDA programs

Compiler Based Sleep Control in CPU Functional Units We propose a compiler based sleep control technique in order to maximize the leakage reduction of runtime PG in CPU functional units. In CPU functional units, conventional time based sleep control techniques fail to effectively reduce the leakage power consumption because lengths of the idle periods in CPU functional units are too short. To this end, we develop a static analysis technique to predict lengths

of such short idle periods with cycle level accuracy. With the static analysis, the compiler inserts additional sleep control information into the executable codes and this information controls the sleep in CPU functional units. The simulation result shows that the proposed technique can reduce 23.6% of the leakage reduction compared to the time-based sleep control.

Cooperative DVFS and Heterogeneous Task Mapping The proposed technique maximize the performance of CPU-accelerator hybrid applications under a given power budget through coordinating DVFS and the heterogeneous task mapping. To do so, it is necessary to precisely predict changes in both performance and power when the device frequencies and task mapping are different. We proposed empirical models of performance and power to solve the problem. With a small number of profiles, the models enable us to select the optimal set of parameters. The experimental result shows that the proposed technique achieves up to 14.4% higher performance compared to other power capping techniques where the task mapping is fixed. Also, the performance achieved by the proposed technique is almost the same as the ideal performance under the given power budget.

6.1 Combination of the Proposed Techniques

As we mentioned in Chapter 1, the proposed techniques can be orthogonally applied to heterogeneous systems. Here, we discuss the reason.

In the compiler based sleep control technique, proposed in Chapter 3, all the details are hidden in the compiler back-end and in the hardware mechanism. The other two techniques are relatively high level optimization techniques when we compared them to the sleep control technique. They can use the low level functionality without any modification. In this case, it is necessary to use the proposed processor and the compiler when we take profile information in the techniques proposed in Chapter 5. Then, the power model can take the effects of the sleep control technique into account.

Also, the OpenACC compiler, proposed in Chapter 3, can be easily modified to use the technique proposed in Chapter 5 because the design of software distributed shared memory can be

applicable to machines with multiple OpenCL devices. Such integration of the two techniques can be done by modifying the translator to generate OpenCL programs instead of CUDA programs. OpenCL programs can be run either on CPUs and GPUs or other accelerators. Therefore, the generated programs can be run on CPUs and accelerators in parallel. Then, at the task mapping step in the OpenACC compiler, we can use the empirical models in Chapter 5 to determine the optimal task mapping and device frequencies. In this case, one remaining problem is that we have to extend the empirical models to the systems equipped with multiple CPUs and multiple GPUs. Our future work includes extending and evaluating the performance and power models in the system with multiple CPUs and GPUs.

6.2 Extension to Other Heterogeneous Systems

Heterogeneous systems with accelerators are getting more and more popular and various kinds of such systems are emerging. Several examples are explained in Chapter 2. In this section, we discuss the extensibility of the proposed techniques to other heterogeneous systems, especially the extensibility of the techniques described in Chapter 3 and Chapter 5. In particular, we examine two directions to extend the proposed techniques. One direction is to extend them for different types of accelerators other than NVIDIA GPUs. The other direction is to extend them for a computing cluster, which consists of multiple computing nodes.

Many Core Processor and Other Accelerators Using accelerators is getting more and more popular. Now various types of accelerators are becoming available. The internal processor architecture of these accelerators are different because each accelerator has its target application domain and its usage scenario. However, from the software perspective, the execution model of these accelerators are very similar :i.e. main control program runs on host CPUs and the parallel tasks are offloaded to accelerators.

Hence, the industry have specified OpenCL as a standard programming APIs for accelerators [4]. Now many vendors support OpenCL in their accelerator devices because OpenCL enables programmers to port their applications easily between other types of accelerators. Such

kind of accelerators includes most of the practical accelerators, Intel Xeon Phi many core processors, FPGAs, and the GPUs from vendors other than NVIDIA :e.g. AMD and ARM. Assuming that the accelerators supports OpenCL, the proposed techniques are extensible to the system with the accelerators. The OpenACC compiler proposed in Chapter 3 can be modified to generate OpenCL programs instead of CUDA programs. Meanwhile, the cooperative DVFS and heterogeneous task mapping technique proposed in Chapter 5 can be also implemented by using OpenCL API.

In terms of the effectiveness, the communication mechanism used in the multi-accelerator OpenACC compiler is considered to be also effective in systems with accelerators other than NVIDIA GPUs. This is because most of the accelerators have dedicated local memories and they are connected to host CPUs on PCI express bus like NVIDIA GPUs. Hence, the communication optimizations used in the OpenACC compiler are also effective in these platforms. Also, the cooperative DVFS and task mapping technique makes use of profile information of the installed accelerators. It can take characteristic of the accelerator into account. Thus, the technique is considered to be also effective to the system with accelerators other than the GPU evaluated in Chapter 5.

GPU Cluster As explained in Chapter 2, it is common that large scale computing system consists of a lot of computing nodes equipped with GPUs and single application uses multiple computing nodes. It is not obvious how to use the techniques in Chapter 3 and Chapter 5 to the applications running across multiple GPU computing nodes.

In the simplest way, the multi-device OpenACC compiler can be combined with MPI. In this case, programmers use the proposed compiler to make use of multiple GPUs inside the node and use MPI to make use of multiple nodes. The situation is similar to hybrid parallel implementation where programmers use both OpenMP and MPI. Such OpenACC plus MPI style programming have already been investigated in previous work [38]. Meanwhile, it is worth investigating the possibility of extending the proposed compiler to make use of multiple accelerators across multiple nodes. Previous work has proposed a directive based compiler which can make use of

multiple GPUs across multiple nodes, though the compiler do not have a GPU kernel code generator and programmers still have to write parallel GPU kernels by hand [16]. It may be possible to extend our compiler system for multiple nodes with the previous technique. This is our future work.

Next, we want to consider how to extend the power capping techniques proposed in Chapter 5 for multiple computing nodes. One easy way to do so is to give a different power budget for each computing node with the proposed technique according to the amount of tasks in the node. In this scheme, busy nodes get assigned more power budget while other nodes which are not busy receive less power budget. Such kind of power management is called power shifting [37] and is said to be effective to improve the energy-efficiency of the computing center. However, the power shifting strategy miss the chance to further optimize the energy-efficiency by re-mapping the parallel tasks across multiple nodes. It is also our future work to orchestrate inter-node task mapping and DVFS settings in these computing nodes.

Chapter 7

Related Work

7.1 GPU Programming Environment

Previous studies evaluate various aspects of the current OpenACC compilers [38] [48] [58]. Wienke et al. [58] evaluate and compare the performance of the realistic GPU applications written in OpenACC and OpenCL. They reported that OpenACC can achieve good performance gains with modest programming cost compared to OpenCL. Levesque et al. [38] investigate the hybrid implementation with MPI and OpenACC in the GPU cluster system. They use OpenACC for the intra-node application development and use MPI to make use of the multiple nodes in the clusters. The result shows that OpenACC can successfully replace the OpenMP and achieve higher performance. In addition to these case studies, Reyes et al. [48] have developed an open source OpenACC compiler. They have illustrated basic design to implement an OpenACC compiler system. Lee et al. [35] compared the current OpenACC compiler to the previous directive based GPU compilers in performance, functionality, and programmability. They listed limitations in the current directive based GPU compilers. The scalability issues in multiple GPU environments is one of the limitations which are listed by them.

Kim et al. [31] have developed the OpenCL system which can make use of multiple GPUs from single GPU OpenCL programs. The concept is similar to ours. That is to build an single GPU memory abstraction on top of the multiple GPUs. We integrate the concept with an OpenACC compiler and propose a small set of directives to enhance communication optimizations.

The integration is essential because we can design compiler and runtime system in a holistic way in order to make use of the application characteristics.

Tipparaju et al. [55] have developed Global Array on multiple GPU environment. Global Array is based on the GAS programming model and it can provide an abstraction of single memory space on top of the distributed memory machine. However, they do not integrate it with directive-based GPU compilers. Thus, they fail to provide advanced communication optimization among CPU and GPUs memories. Several researches proposed programming interface based on data-flow programming model for multiple GPU environments [8, 16]. OmpSs has been presented as a directive based programming model to execute single GPU CUDA programs on GPU clusters [16]. OmpSs provides directives to express data dependency between parallel tasks. The directives allow programmers to express the detailed memory access pattern, so they are similar to the *localaccess* directives proposed in this paper. However, the OmpSs system does not include compilers to generate programs running on the GPUs. Hence the programmers must manually write the CUDA codes and divide the applications into parallel tasks. Also, their inter-GPU memory manager relies on inefficient software cache mechanisms. StarPU [8] provides another high level programming interface for the multi-GPU environments. In the programming model, programs must be rewritten with a data-flow based API, so we cannot incrementally port the existing programs to the multi-GPU environments.

A lot of researchers have proposed compilers or runtime systems to schedule data parallel tasks among CPUs and GPUs in order to maximize the performance of GPGPU applications [19, 21, 39, 46, 50]. Luk et al. [39] have proposed a dynamic compiler for CPU-GPU heterogeneous systems. The compiler maps parallel tasks among CPUs and GPUs based on an empirical performance model. Scogland et al. [50] have integrated heterogeneous task scheduling into OpenMP compilers with accelerator extensions. Grewe et al. [21] have proposed a static technique based on machine learning in order to determine the optimal percentages of CPU and GPU tasks. On the other hand, several domain specific platforms for GPUs adopt the capability to schedule parallel tasks among CPUs and GPUs [19, 46]. Since these techniques do not consider power consumption of CPU-GPU hybrid executions, we cannot directly apply these

techniques to systems under power budgets.

7.2 Runtime Power Gating

In prior works for overcoming the leakage power problem in processors, leakage power reduction techniques in cache structures are investigated[30][18]. In addition to leakage power reduction in cache structures, researchers aim to apply runtime PG to functional units in CPUs [25] and in GPUs [6]. In these researches, they use timeout based sleep control techniques to avoid excessive sleep-wakeup mode transitions. Also, some researchers have proposed compiler based sleep control techniques [47, 60, 49]. But the accuracy of these analysis is not good enough to reduce leakage power consumption in short idle periods whose lengths are in the same order to BET.

These previous works target the leakage power consumed in relatively coarse grain idle periods originated in application phase in which the target functional units are rarely used. Based on these prior sleep control techniques in functional units, recent research also investigates various aspects of dynamic power gating in functional units. Kannan et al. [29] propose a sleep control technique to adapt the leakage power fluctuation caused by the process variation and temperature variation. Prior to our work, Lungu et al [40] point out that fine grain idle periods can cause the leakage power increase in the existing sleep control techniques and propose a guard mechanism to prevent such a power increase. While Lungu et al. treat fine grain idle periods as bad behavior of running applications, we treat the fine grain idle periods as the chance for reducing the leakage power in functional units. We propose software-hardware hybrid sleep control techniques which can effectively reduce the leakage power consumed in fine grain idle periods, which can not be captured by the existing techniques.

7.3 Power Capping

Power capping techniques have been proposed for various layers of computing systems [17, 37]. Fan et al [17] evaluates the effectiveness of cluster level power capping with commercial work-

loads in data centers. David [15] have proposed a power capping technique for memory modules, and Kai et al [41] have proposed a power capping technique for many-core CPUs running mixed groups of multi-threaded applications. However, no previous work has investigated a power capping for CPU-GPU heterogeneous systems.

GreenGPU [42] is a system which is related to the proposed technique in Chapter 5. The system can apply both DVFS and task mapping between CPUs and GPUs in order to maximize the energy-efficiency. However, they use DVFS and task mapping individually, so their method cannot set optimal parameters of DVFS and task mapping. Plus, their technique cannot be used as a power capping technique because they do not provide any mechanism to keep a power constraint.

Chapter 8

Conclusion

Energy-efficiency has become one of the most important metric in recent computing systems. To this end, heterogeneous systems with GPUs are promising platforms because they have higher peak performance and energy-efficiency than those of conventional homogeneous systems.

However, achieving high energy efficiency in practical situations is not easy because the energy efficiency of the heterogeneous systems highly depends on the optimizations used in the softwares running on them. The hardware architecture of the CPU-GPU heterogeneous systems are very different from conventional systems only with CPUs and we have to reconsider the optimization techniques in compilers and runtime software to get maximum benefit from the heterogeneous system architecture.

To attack these problems, in this dissertation, we proposed three new compiler and runtime optimization techniques. One is for improving the programmability of the GPU platform, and the other two are for reducing dynamic and static power consumption of the system.

First, to improve the programmability of GPU platforms, we present an OpenACC compiler system with the capability to execute single GPU OpenACC programs on multiple GPUs. By orchestrating the compiler and the runtime system, we could build an efficient multi-GPU directive-based compiler on top of the current platforms. In order to enable the advanced communication optimizations in the proposed system, we also propose a small set of directives as an extension to the current OpenACC standards. The directives allow programmers to express the memory access patterns in the parallel loops with a few lines of additional codes. We im-

plemented and evaluated the prototype system. The compiler successfully reduces most of the programming complexity in the multi-GPU programming by automating the data management among multiple GPU memories. And the performance of the compiler is comparable to the hand optimized CUDA programs, in which we have spent much more time to optimize the programs for the GPU architecture.

Second, to reduce static power wasted in CPUs, we propose a sleep control technique to reduce leakage power consumed in CPU functional units during its execution. To maximize the effect of the runtime power gating, it is necessary to predict the length of each fine grain idle period accurately because excessive sleep/wakeup mode transitions must be avoided to suppress the energy overhead caused by power gating. We propose a compiler technique to analyze program codes and predict the length of the idle periods with cycle-level accuracy. Simulation results show that, the proposed technique can achieve higher leakage power reduction, 23.6% on average, than the conventional time-based sleep control.

Finally, to reduce and manage dynamic power consumption, we propose an efficient power capping technique for heterogeneous systems. In the proposed technique, the frequency of the CPU, the frequency of the accelerator, and the task mapping between the CPU and the accelerator are cooperatively determined in order to avoid both the power violation and the load imbalance between the CPU and the GPU. To guide the parameter settings, we build empirical models to predict the execution time and the power consumption of a heterogeneous system. The experimental result shows that the proposed power capping technique can achieve up to 14.4% higher performance compared to conventional power capping techniques. This is close to the performance with the ideal parameter settings.

Acknowledgment

First of all, I would like to express my sincere to my advisor, Professor Hiroshi Nakamura, for his advice, encouragement, and support during my studies. He always thoughtfully listens to my ideas and provides great insights about this research.

I would like to thank Professor Masaaki Kondo for giving me a lot of insights on processor architecture and compiler technology. They help me doing this research a lot.

I would like to thank Professor Shinobu Miwa for providing great discussions about this research.

I would like to thank Takashi Nakada for his advice and enhancement for this research.

I would like to thank Professor Shinji Hara for providing thoughtful comments on how to summarize this dissertation.

I would like to thank Professor Masatoshi Ishikawa for suggesting the interesting discussion about the applicability of the proposed techniques.

I would like to thank Professor Kengo Nakajima for providing the thoughtful comments on detailed analysis on the evaluation results.

I would like to thank Professor Koji Inoue and Professor Naoya Maruyama for teaching me the direction of this research. This dissertation is never completed without their thoughtful advices.

I wish to acknowledge numerous people who studied and discussed with me. In particular, I would like to thank Yuan He, Shinya Takamaeda and Takaaki Miyajima. They give me good motivation for this research.

Bibliography

- [1] Green500 resources. <http://www.green500.org/resources/>.
- [2] Nvidia management library (nvm). <https://developer.nvidia.com/nvidia-management-library-nvm>.
- [3] Openacc directives for accelerators. <http://www.openacc-standard.org/>.
- [4] Opencl specification. <http://www.khronos.org/opencl/>.
- [5] Rose compiler infrastructures. <http://rosecompiler.org/>.
- [6] Mohammad Abdel-Majeed, Daniel Wong, and Murali Annavaram. Warped gates: Gating aware scheduling and power gating for gpgpus. In *MICRO '13: Proceedings of the 46th annual IEEE/ACM international symposium on Microarchitecture*, Davis, Carifornia, USA, 2013.
- [7] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.
- [8] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198, February 2011.
- [9] Francois Bodin and Stephane Bihan. Heterogeneous multicore parallel programming for graphics processing units. *Scientific Programming.*, 17(4):325–336, December 2009.

- [10] Doug Burger and Todd M. Austin. The simplescalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, 1997.
- [11] A.P. Chandrakasan, S. Sheng, and R.W. Brodersen. Low-power cmos digital design. *Solid-State Circuits, IEEE Journal of*, 27(4):473–484, 1992.
- [12] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE International Symposium on Workload Characterization, 2009. IISWC 2009.*, pages 44–54, Oct.
- [13] D. Chen and D. Singh. Invited paper: Using opencl to evaluate the efficiency of cpus, gpus and fpgas for information filtering. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 5–12, 2012.
- [14] Anthony Danalis, Gabriel Marin, Collin Mccurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, and Jeffrey S. Vetter. The scalable heterogeneous computing (shoc) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU3)*, pages 63–74, 2010.
- [15] H. David, E. Gorbatov, Ulf R. Hanebutte, R. Khanna, and C. Le. Rapl: Memory power estimation and capping. In *Low-Power Electronics and Design (ISLPED), 2010 ACM/IEEE International Symposium on*, pages 189–194, 2010.
- [16] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesus Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: A proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.
- [17] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power provisioning for a warehouse-sized computer. *SIGARCH Comput. Archit. News*, 35(2):13–23, June 2007.
- [18] Krisztián Flautner, Nam Sung Kim, Steve Martin, David Blaauw, and Trevor Mudge. Drowsy caches: Simple techniques for reducing leakage power. In *ISCA '02: Proceed-*

- ings of the 29th annual international symposium on Computer architecture*, pages 148–157, Washington, DC, USA, 2002. IEEE Computer Society.
- [19] Abdullah Gharaibeh, Lauro Beltrão Costa, Elizeu Santos-Neto, and Matei Ripeanu. A yoke of oxen and a thousand chickens for heavy lifting graph processing. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, pages 345–354, New York, NY, USA, 2012. ACM.
- [20] M. Gokhale, J. Cohen, A. Yoo, W.M. Miller, A. Jacob, C. Ulmer, and R. Pearce. Hardware technologies for high-performance data-intensive computing. *Computer*, 41(4):60–68, 2008.
- [21] Dominik Grewe and Michael F. P. O’Boyle. A static task partitioning approach for heterogeneous systems using opencl. In *Proceedings of the 20th international conference on Compiler construction: part of the joint European conferences on theory and practice of software*, CC’11/ETAPS’11, pages 286–305, Berlin, Heidelberg, 2011. Springer-Verlag.
- [22] Tianyi David Han and Tarek S. Abdelrahman. hicuda: a high-level directive-based language for gpu programming. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 52–61, New York, NY, USA, 2009. ACM.
- [23] A. Heinecke, K. Vaidyanathan, M. Smelyanskiy, A. Kobotov, R. Dubtsov, G. Henry, A.G. Shet, G. Chrysos, and P. Dubey. Design and implementation of the linpack benchmark for single and multi-node systems based on intel xeon phi coprocessor. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 126–137, 2013.
- [24] John L. Henning. Spec cpu2000: Measuring cpu performance in the new millennium. *Computer*, 33(7):28–35, 2000.
- [25] Zhigang Hu, Alper Buyuktosunoglu, Viji Srinivasan, Victor Zyuban, Hans Jacobson, and Pradip Bose. Microarchitectural techniques for power gating of execution units. In *ISLPED*

- '04: *Proceedings of the 2004 international symposium on Low power electronics and design*, pages 32–37, New York, NY, USA, 2004. ACM.
- [26] Wenmei W. Hwu. *GPU Computing Gems Emerald Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011.
- [27] ThinkTank Energy Products Inc. watts up? products: Meter. <https://www.wattsupmeters.com/secure/products.php?pn=0>, 2013.
- [28] C. Isci, A. Buyuktosunoglu, C.-Y. Chen, P. Bose, and M. Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *the 39th Annual IEEE/ACM International Symposium on Microarchitecture, 2006. MICRO-39.*, pages 347–358, 2006.
- [29] Deepa Kannan, Aviral Shrivastava, Sarvesh Bhardwaj, and Sarma Vrudhul. Power reduction of functional units considering temperature and process variations. *VLSI Design, International Conference on*, 0:533–539, 2008.
- [30] Stefanos Kaxiras, Zhigang Hu, and Margaret Martonosi. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, pages 240–251, New York, NY, USA, 2001. ACM.
- [31] Jungwon Kim, Honggyu Kim, Joo Hwan Lee, and Jaejin Lee. Achieving a single compute device image in opencl for multiple gpus. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming, PPOPP '11*, pages 277–288, New York, NY, USA, 2011. ACM.
- [32] V.V. Kindratenko, J.J. Enos, Guochun Shi, M.T. Showerman, G.W. Arnold, J.E. Stone, J.C. Phillips, and Wen-Mei Hwu. Gpu clusters for high-performance computing. In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1–8, 2009.

- [33] Klaus Kofler, Ivan Grasso, Biagio Cosenza, and Thomas Fahringer. An automatic input-sensitive approach for heterogeneous task partitioning. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 149–160, New York, NY, USA, 2013. ACM.
- [34] Seyong Lee, Seungjai Min, and Rudolf Eigenmann. Openmp to gpgpu: a compiler framework for automatic translation and optimization. *ACM SIGPLAN Notices*, 44(4):101–110, February 2009.
- [35] Seyong Lee and Jeffrey S. Vetter. Early evaluation of directive-based gpu programming models for productive exascale computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 23:1–23:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.
- [36] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. *ACM SIGARCH Computer Architecture News*, 38(3):451–460, June 2010.
- [37] Charles Lefurgy, Xiaorui Wang, and Malcolm Ware. Power capping: a prelude to power shifting. *Cluster Computing*, 11(2):183–195, 2008.
- [38] John M. Levesque, Ramanan Sankaran, and Ray Grout. Hybridizing s3d into an exascale application using openacc: an approach for moving to multi-petaflops and beyond. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 15:1–15:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.
- [39] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42nd Annual*

- IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 45–55, New York, NY, USA, 2009. ACM.
- [40] Anita Lungu, Pradip Bose, Alper Buyuktosunoglu, and Daniel J. Sorin. Dynamic power gating with quality guarantees. In *Proceedings of the 14th ACM/IEEE international symposium on Low power electronics and design*, ISLPED '09, pages 377–382, New York, NY, USA, 2009. ACM.
- [41] Kai Ma, Xue Li, Ming Chen, and Xiaorui Wang. Scalable power control for many-core architectures running multi-threaded applications. In *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 449–460, New York, NY, USA, 2011. ACM.
- [42] Kai Ma, Xue Li, Wei Chen, Chi Zhang, and Xiaorui Wang. Greengpu: A holistic approach to energy efficiency in gpu-cpu heterogeneous architectures. In *the 41st International Conference on Parallel Processing (ICPP'12)*, pages 48–57, 2012.
- [43] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka. Physis: An implicitly parallel programming model for stencil computations on large-scale gpu-accelerated supercomputers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2011*, pages 1–12, nov. 2011.
- [44] Tapasya Patki, David K. Lowenthal, Barry Rountree, Martin Schulz, and Bronis 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*, ICS '13, pages 173–182, New York, NY, USA, 2013. ACM.
- [45] Rajesh Kumar and Glenn Hinton. A Family of 45nm IA Processors. In *Proc. IEEE International Solid-State Circuits Conference*, California, USA, Feb 2009.
- [46] Vignesh T. Ravi, Wenjing Ma, David Chiu, and Gagan Agrawal. Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel config-

- urations. In *Proceedings of the 24th ACM International Conference on Supercomputing, ICS '10*, pages 137–146, New York, NY, USA, 2010. ACM.
- [47] Siddharth Rele, Santosh Pande, Soner Önder, and Rajiv Gupta. Optimizing static power dissipation by functional units in superscalar processors. In *Proceedings of the 11th International Conference on Compiler Construction, CC '02*, pages 261–275, London, UK, 2002. Springer-Verlag.
- [48] Ruymn Reyes, Ivn Lpez-Rodrguez, JuanJ. Fumero, and Francisco Sande. accull: An openacc implementation with cuda and opencl support. In *Proceedings of the Euro-Par 2012 Parallel Processing*, volume 7484 of *Lecture Notes in Computer Science*, pages 871–882. Springer Berlin Heidelberg, 2012.
- [49] Soumyaroop Roy, Srinivas Katkoori, and Nagarajan Ranganathan. A compiler based leakage reduction technique by power-gating functional units in embedded microprocessors. In *VLSID '07: Proceedings of the 20th International Conference on VLSI Design held jointly with 6th International Conference*, pages 215–220, Washington, DC, USA, 2007. IEEE Computer Society.
- [50] T.R.W. Scogland, B. Rountree, Wu chun Feng, and B.R. de Supinski. Heterogeneous task scheduling for accelerated openmp. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 144 –155, may 2012.
- [51] Naomi Seki, Lei Zhao, Jo Kei, Daisuke Ikebuchi, Yu Kojima, Yohei Hasegawa, Hideharu Amano, Toshihiro Kashima, Seidai Takeda, Toshiaki Shirai, Mitsutaka Nakata, Kimiyoshi Usami, Tetsuya Sunata, Jun Kanai, Mitaro Kanai, Masaaki Kondo, and Hiroshi Nakamura. A fine-grain dynamic sleep control scheme in mips r3000. In *ICCD '08: Proceedings of the 2008 international conference on computer design*, pages 612–617, Washington, DC, USA, 2008. IEEE Computer Society.

- [52] A. Sidelnik, S. Maleki, B.L. Chamberlain, M.J. Garzaran, and D. Padua. Performance portability with the chapel language. In *Proceedings of the IEEE 26th International Parallel Distributed Processing Symposium (IPDPS), 2012*, pages 582–594, May.
- [53] Shuaiwen Song, Rong Ge, Xizhou Feng, and Kirk W. Cameron. Energy profiling and analysis of the hpc challenge benchmarks. *IJHPCA*, (3):265–276.
- [54] Shuaiwen Song, Chunyi Su, Barry Rountree, and Kirk W. Cameron. A simplified and accurate model of power-performance efficiency on emergent gpu architectures. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, IPDPS '13*, pages 673–686, Washington, DC, USA, 2013. IEEE Computer Society.
- [55] Vinod Tipparaju and Jeffrey S. Vetter. Ga-gpu: Extending a library-based global address spaceprogramming model for scalable heterogeneous computing systems. In *Proceedings of the 9th conference on Computing Frontiers, CF '12*, pages 53–64, New York, NY, USA, 2012. ACM.
- [56] Didem Unat, Xing Cai, and Scott B. Baden. Mint: Realizing cuda performance in 3d stencil methods with annotated c. In *Proceedings of the international conference on Supercomputing, ICS '11*, pages 214–224, New York, NY, USA, 2011. ACM.
- [57] J.S. Vetter, R. Glassbrook, J. Dongarra, K. Schwan, B. Loftis, S. McNally, J. Meredith, J. Rogers, P. Roth, K. Spafford, and S. Yalamanchili. Keeneland: Bringing heterogeneous gpu computing to the computational science community. *Computing in Science Engineering*, 13(5):90–95, Sept.-Oct.
- [58] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter Mey. Openacc: First experiences with real-world applications. In *Proceedings of the Euro-Par 2012 Parallel Processing*, volume 7484 of *Lecture Notes in Computer Science*, pages 859–870. Springer Berlin Heidelberg, 2012.

- [59] Michael Wolfe. Implementing the pgi accelerator model. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 43–50, New York, NY, USA, 2010. ACM.
- [60] Yi-Ping You, Chingren Lee, and Jenq Kuen Lee. Compilers for leakage power reduction. *ACM Trans. Des. Autom. Electron. Syst.*, 11(1):147–164, 2006.

List of Publications by the Authors

Journal Publications

[J-1] Eishi Arima, Toshiya Komoda, Takashi Nakda, Shinobu Miwa and Hiroshi Nakamura. Lost Data Prefetching to Reduce Performance Degradation Caused by Powering off Caches (in Japanese). *IP SJ Transactions on Advanced Computing Systems*, Vol 6, No. 3, pages 118–130, September 2013.

[J-2] Toshiya Komoda, Hiroshi Sasaki, Masaaki Kondo and Hiroshi Nakamura. Compiler Directed Leakage Reduction Technique Considering the Effects of Fine Grained Idle Periods (in Japanese). *IP SJ Transactions on Advanced Computing Systems*, Vol 4, No. 4, pages 36–50, October 2011.

International Conference and Workshop Publications

[I-1] Masaaki Kondo, Hiroaki Kobayashi, Ryuichi Sakamoto, Motoki Wada, Jun Tsukamoto, Mitaro Namiki, Weihang Wang, Hideharu Amano, Matsunaga Kensaku, Kudo Masaru, Kimiyoshi Usami, Toshiya Komoda and Hiroshi Nakamura. Design and Evaluation of Fine-Grained Power-Gating for Embedded Microprocessors. In *Proceedings of the Design, Automation and Test in Europe 2014 (DATE'14)*, March, 2014(to appear).

[I-2] Toshiya Komoda, Shingo Hayashi, Takashi Nakada, Shinobu Miwa and Hiroshi Nakamura. Power Capping of CPU-GPU Heterogeneous Systems through Coordinating DVFS and Task Mapping. In *Proceedings of the 31st IEEE International Conference on Computer Design, (ICCD'13)*, pages 349–356, October, 2013.

- [I-3] Toshiya Komoda, Shinobu Miwa, Hiroshi Nakamura and Naoya Maruyama. Integrating Multi-GPU Execution into an OpenACC Compiler. *Proceedings of the 42nd International Conference on Parallel Processing, (ICPP'13)*, pages 260–269, October, 2013.
- [I-4] Toshiya Komoda, Shinobu Miwa and Hiroshi Nakamura. Communication Library to Overlap Computation and Communication. *Proceedings of the 17th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'12, IPDPS Workshop)*, pages 567–573, May 2012.
- [I-5] Toshiya Komoda, Hiroshi Sasaki, Masaaki Kondo and Hiroshi Nakamura. Compiler Directed Fine Grain Power Gating for Leakage Power Reduction in Microprocessor Functional Units. *Proceedings of the 7th Workshop on Optimizations for DSP and Embedded Systems(ODES'09)*, pages 39–48, March 2009.

Domestic Conference and Technical Society Meeting Publications

- [D-1] Toshiya Komoda, Shinobu Miwa, and Hiroshi Nakamura. Software Prefetcher for Communication between CPUs and GPUs (in Japanese). *IPSJ SIG Notes*, ARC-201(25), pages 1–7, August 2012.
- [D-2] Eishi Arima, Toshiya Komoda, Shinobu Miwa, and Hiroshi Nakamura. An Implementation of Lost Data Prefetching for Reducing Performance Degradation caused by powering off caches during idle periods (in Japanese). *IPSJ SIG Notes*, ARC-201(15), pages 1–7, August 2012.
- [D-3] Eishi Arima, Toshiya Komoda, Shinobu Miwa, Hiroki Noguchi, Nomura Kumiko, Abe Keiko, Shinobu Fujita, and Hiroshi Nakamura. Comparison between Leakage Reduction Methods for Last Level Caches under OS Power Management (in Japanese). *Proceedings of the 25th Workshop on Circuits and Systems*, pages 402–407, July 2012.
- [D-4] Eishi Arima, Toshiya Komoda, Shinobu Miwa, and Hiroshi Nakamura. An Evaluation of Lost Data Prefetching for Reducing Performance Degradation Caused by Powering off

- Caches during Idle Periods (in Japanese). *IPSJ SIG Notes*, ARC-198(2), pages 1–6, January 2012.
- [D-5] Toshiya Komoda, Shinobu Miwa, and Hiroshi Nakamura. Pipeline Parallel Programming API based on OpenCL (in Japanese). *IPSJ SIG Notes*, HPC-132(10), pages 1–7, December 2011.
- [D-6] Hiroaki Kobayashi, Isamu Mogi, Kazuki Kimura, Toshiya Komoda, Mikiko Sato, Masaaki Kondo, Hiroshi Nakamura, Mitaro Namiki. A Study on an Operating System for Managing Object Code Optimized for Fine-grain Power Gating Control (in Japanese). *IPSJ SIG Notes*, ARC-195(1), pages 1–8, April 2011.
- [D-7] Toshiya Komoda, Hiroshi Sasaki, Masaaki Kondo and Hiroshi Nakamura. Compiler Directed Leakage Reduction Technique Considering the Effects of Fine Grained Idle Periods (in Japanese). *Proceedings of the Symposium on Advanced Computing Systems and Infrastructures*, SACSIS'2009, pages 11–18, May 2009.