Studies on Automatic Parallelization for Heterogeneous and Homogeneous Multicore Processors

Feb 2012

Akihiro HAYASHI

Studies on Automatic Parallelization for Heterogeneous and Homogeneous Multicore Processors

Feb 2012

Waseda University

Graduate School of Fundamental Science and Engineering,

Major in Computer Science and Engineering,

Research on Advanced Computing Systems

Akihiro HAYASHI

Abstract

There has been a growing interest in heterogeneous and homogeneous multicore processors due to their excellent characteristic, higher performance and lower power consumption than single core processors. In order to exploit the capability of heterogeneous and homonogeneous multicore processor, an automatic parallelization is greatly important since parallel programming by hand is time consuming task.

This thesis proposes an innovative OSCAR (Optimally Scheduled Advanced Multicore Processors) heterogeneous automatic parallelizing compiler, which realizes an automatic parallelization and a power reduction for various heterogeneous and homogeneous multicores.

Performance evaluation and power evaluation on 15 core heterogeneous multicore RP-X using media applications attains speedups of 32.6 times with eight SH-4A cores and four FE-GA cores, 18.8 times with two SH-4A cores and one FE-GA core, 5.4 times with eight SH-4A cores against sequential execution by a single SH4A core and 70% of power reduction for the optical flow.

Performance evaluation on several SMP servers using dose calculation for heavy particle radiotherapy for cancer treatment attains good speedups of 9.0 times with 12 processor cores on Hitachi HA8000/RS220 system based on the Intel Xeon Processor and 50.0 times with the 64 processor cores on Hitachi SR16000 system

based on the IBM Power 7 processor.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background and purpose

Multicore processors, which integrate multiple processors on a chip, are utilized in wide variety of products such as cell phones, digital televisions, car navigation systems, personal computers, workstations and supercomputers. Today, multicore processors exist everywhere due to the following reasons: (1) Achieving speedups by increasing core frequency is technically difficult and it results in higher power consumption since the power consumption is proportional to the frequency times square of the voltage. (2) Achieving speedups by increasing the number of cores improves power efficiency by keeping the core frequency low as hardware vendors have developed many types of multicore processors such as IBM/SONY/TOSHIBA CELL BE[PAB+05], Renesas/Waseda RP-2[IHY+08], RP-X[YIK+10].

Multicore processors are categorized into homogeneous multicores and heterogeneous multicores (Fig.1.1). Homogeneous multicores integrate identical multiple cores on a chip. On the other hand, heterogeneous multicores integrate special-purpose accelerator cores such as dynamically reconfigurable prcessors(DRP) and graphic processing unit(GPU) in addition to general-purpose processor cores on a chip in order to keep up with various demands such as multimedia processing. Especially, integrating accelerators improves power efficiency because the accelerators realize high performance at low frequency and low power.

However, as shown in Fig.1.2, very hard software development efforts for multicores are required since programmers have to manually parallelize a program by taking the following steps: (1) write a program, (2) decomposition of the program into tasks, (3) scheduling these tasks onto general processors and accelerators by inserting synchronization codes and data transfer codes. Especially, for heterogeneous multicores, programmers have to develop a unique code for each accelera-

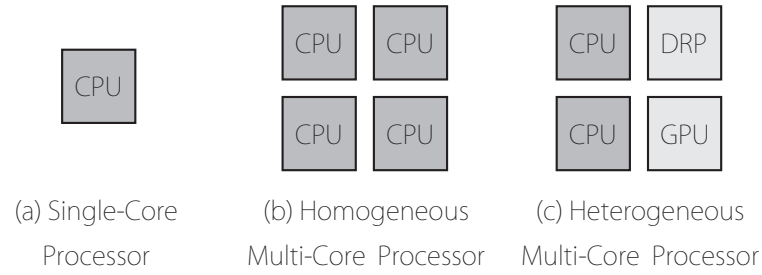


Figure 1.1: Category of Processors

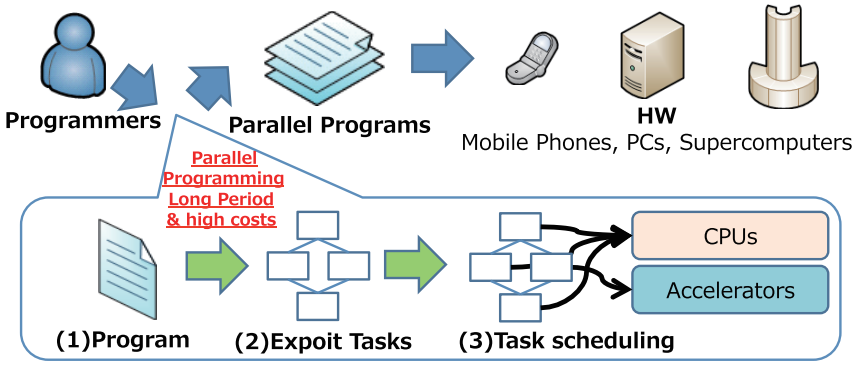


Figure 1.2: Development Flow

tor using special-purpose languages like NVIDIA CUDA[GGN+08] and Khronos OpenCL[khr] and make performance tuning by taking the characteristic of the target accelerators into account. Facilitating programming for heterogeneous and homogeneous multicore is greatly important since it takes several months in order to exploit the full capability of multicores manually.

In order to make the programming for heterogeneous and homogeneous multicores easier, this thesis proposes an innovative OSCAR (Optimally Scheduled Advanced Multicore Processors) heterogeneous automatic parallelizing compiler, which realizes an automatic parallelization for various heterogeneous multicores including homogeneous multicores. Recent many studies have tried to handle this

software development issue. In terms of homogeneous multicores, many works have been trying to realize an automatic parallelization. Polaris compiler[EHP98], SUIF compiler[HAA+96], IBM XL compiler and Intel compiler are an example of automatic parallelizing compilers. However, these parallelizing compilers only exploit the loop level parallism and are designed homogeneous multicores.

In terms of heterogeneous multicores, researchers came up with many solution so as to facilitate the difficulty heterogeneous of parallel programming. For example, NVIDIA and Khronos Group introduced CUDA[GGN+08] and OpenCL[khr]. Also, PGI accelerator compiler[Wol10] and HMPP[DBB07] provides a high-level programming model for accelerators. However, these works focus on facilitating the development for accelerators. Programmers need to distribute tasks among general-purpose processors and accelerator cores by hand. In terms of workload distribution, Qilin[LHK09] automatically decides which task should be executed on a general-purpose processor or an accelerator at runtime. However, programmers still need to parallelize a program by hand. While these works rely on programmers' skills, CellSs[BPBL09] performs an automatic parallelization of a subset of sequential C program with data flow annotations on CELL BE. CellSs automatically schedules tasks onto processor elements at runtime. The task scheduler of CellSs, however, is implemented as a homogeneous task scheduler, namely the scheduler is executed on PPE and just distributes tasks among SPEs.

Therefore, realizing an automatic parallelization for heterogeneous and homogeneous multicore is required. This thesis makes the following contributions:

- A proposal in which the compilation flow of the OSCAR compiler does not depend on the processor configuration, or the number of general-purpose cores and accelerators.

- A proposal, which enables the OSCAR compiler to perform an automatic parallelization for heterogeneous multicore by utilizing existing tools and libraries for accelerators.

- An evaluation of the processing performance and the power efficiency using widely used media applications including motion-tracking algorithm and audio encoding software on the authors developed RP-X heterogeneous multicore chip.

- An evaluation of the processing performance using dose calculation, which is used in a particle radiotherapy for cancer treatment on SMP servers.

1.2 Thesis outline

The thesis consists of 6 chapters.

Chapter 2 "OSCAR Compiler Applicable Heterogeneous Multicore Architecture" firstly defines a generic architecture in order to build the compilation flow of the OSCAR compiler, which can support various kinds of shared memory multiprocessor configurations. The model multicore architecture for the OSCAR compiler is composed of general-purpose processors, accelerators, direct memory access controller, on-chip centralized shared memory, and off-chip CSM. Moreover, the OSCAR heterogeneous multicore architecture can handle accelerators without controllers and accelerators with their controllers, or general-purpose processors. Both general-purpose processors and accelerators with controller may have a local data memory, a distributed shared memory, a data transfer unit, frequency voltage control registers, an instruction cache memory and a data cache memory. The local data memory keeps private data. The distributed shared memory is a dual

port memory, which enables point-to-point direct data transfer and low-latency synchronization among processors. This chapter also shows most of existing heterogeneous and homogeneous multicore can be seen as a subset of the OSCAR architecture and the OSCAR compiler can support various heterogeneous and homogeneous multicores.

Chapter 3 "Compilation Framework for the Heterogeneous Multicore Architecture" describes the proposed framework including an automatic compilation flow for the target architecture. The input of the framework is a sequential program written in Parallelizable C, a kind of C programming style for parallelizing compiler, or Fortran77 and the output is an executable for a target heterogeneous and homogeneous multicore. The compilation flow consists of 4 steps. First of all, accelerator compilers or programmers insert hint directives immediately before loops or function calls, which can be executed on the accelerator, in a sequential program. Then, the OSCAR compiler parallelizes the source program considering with hint directives. The compiler automatically performs (1) decomposition of a program into tasks, (2) scheduling these tasks onto general processors and accelerators by inserting synchronization, (3) data transfer, and (4) power control codes which controls the frequency and the voltage of the chip. After that, the compiler generates a parallelized C or Fortran program for general-purpose processors and accelerator cores by using OSCAR API, multicore application program interface (API) including thread APIs, memory mapping APIs, data transfer APIs and power control APIs. At that time, the compiler generates C or Fortran source codes as separate files for accelerator cores. Each file includes functions to be executed on accelerators when a function is scheduled onto accelerator by the OSCAR compiler. Afterwards, each accelerator compiler generates objects for its own

target accelerator. Finally, an API analyzer prepared for each heterogeneous multicore translates OSCAR APIs into runtime library calls, such as pthread library. Afterwards, an ordinary sequential compiler for each processor from each vender generates an executable. This chapter also describes the proposed framework effectively utilizes the existing accelerator compilers and hand-tuned accelerator libraries and realizes an automatic parallelization for various multicore processors.

Chapter 4 "Parallel Processing Schemes for Media Applications on the Heterogeneous Multicore Architecture" describes parallel processing schemes for the motion-tracking algorithm called optical flow and the audio encoder called advanced audio codec (AAC) encoder. This chapter also evaluates the processing performance and the power reduction by the proposed framework on the RP-X heterogeneous multicore, which integrates eight general-purpose processor cores, or SH4A, and three kinds of accelerators including dynamically reconfigurable processor (DRP) accelerators, or FE-GA. The RP-X is developed by Renesas, Hitachi, Tokyo Institute of Technology and Waseda University . Optical flow calculates velocity field between two images. This program consists of the following parts: dividing the image into 16x16 pixel blocks, searching a similar block in the next image for every block in the current image, shifting 16 pixels and generating the output. The OSCAR compiler exploits the parallelism of the loop, which searches a similar block in the next image. In addition, accelerator compiler developed by Hitachi analyzed that the sum of absolute difference (SAD), which occupies a large part of the program execution time, is to be executed on accelerator. Then the OSCAR compiler schedules these tasks onto general-purpose processors and accelerators and applies power control. AAC encoder is based on the AAC-LC encode program provided by Renesas and Hitachi. This program consists of filter bank,

midside (MS) stereo, quantization and huffman coding. The OSCAR compiler exploits the frame-level parallelism and schedules these tasks onto general-purpose processors and accelerators and applies power control. The hand-tuned library for filter bank, MS stereo and quantization is used for accelerator. As a result, the framework attains speedups of 32.6 times with eight SH-4A cores and four FE-GA cores, 18.8 times with two SH-4A cores and one FE-GA core, 5.4 times with eight SH-4A cores against sequential execution by a single SH4A core and 70% of power reduction for the optical flow on the RP-X. The proposed framework realizes an automatic parallelization for various processor configurations. In addition, the compilation time is within a few minutes even though it takes several months in order to exploit the full capability of multicores manually.

Chapter 5 "A Parallel Processing Scheme for Dose Calculation on SMP Servers" describes a parallel processing scheme of dose calculation for particle radiotherapy for cancer treatment and evaluates processing performance by the proposed framework on SMP servers. This dose calculation engine is based on the clinically used program developed by National Institute of Radiological Sciences (NIRS) and Mitsubishi Electronics. This program simulates treatment plan for cancer: It simulates how much particles attack the cancer but it takes long time for the simulation. In order to reduce the simulation time by parallelization, this thesis proposes a processing scheme for the application and enables the OSCAR compiler to exploit the parallelism of the calculation engine. As a result, the proposed method attains good speedups of 9.0 times with 12 processor cores on Hitachi HA8000/RS220 system based on the Intel Xeon Processor and 50.0 times with the 64 processor cores on Hitachi SR16000 system based on the IBM Power 7 processor.

Chapter 6 “Conclusions” concludes the thesis and explains future works.

Chapter 2

OSCAR Compiler Applicable Heterogeneous Multicore Architecture

2.1 Introduction

The demand for multicore processors has been increasing for several years in order to improve system performance keeping power consumption low. IBM Power7[KSSF10], Intel Single Chip Cloud[HDH+10], Renesas RP2[IHY+08], NEC NaviEngine[MTT+07], Toshiba Venezia[NTF+08] and Fujitsu Sparc64 VIIIfx[MYK+10] are examples of multicore processors. In addition, there has been a growing interest in heterogeneous multicores which integrate special purpose accelerator cores in addition to general-purpose processor cores on a chip. One of the reason for this trend is that heterogeneous multicores allow us to attain high performance with low frequency and low power consumption. Various semiconductor vendors have released heterogeneous multicores such as CELL BE[PAB+05], GPGPU[LHG+06], Uniphier[NYY+07], RP1[YKH+07] and RP-X[YIK+10].

However, the softwares for heterogeneous multicores generally require large development efforts such as the decomposition of a program into tasks, the implementation of accelerator code, the scheduling of the tasks onto general-purpose processors and accelerators, and the insertion of synchronization and data transfer codes. Long software development periods are required even for expert programmers.

In order to facilitate a parallel programming for heterogeneous multicore, this thesis proposes an OSCAR automatic heterogeneous parallelizing compiler as mentioned in Chapter 1. Although conventional homogeneous parallelizing compilers, such as Polaris compiler[EHP98] and SUIF compiler[HAA+96], only exploit the loop level parallelism, the OSCAR compiler exploits multigrain parallelism including corase-grain parallelism, loop level parallelism and fine-grain parallelism[KHM+91]

This chapter in particular discusses the importance of the OSCAR compiler

cooperative heterogeneous multicore architecture and define a generic multicore architecture to which the parallelization of the OSCAR compiler is applicable. It is important to define a generic architecture and build a compilation framework for the architecture since there are various kinds of heterogeneous multicore. Specifically, an memory architecture design, an interconnection network design and an accelerator design are completely depend on an architecture design. For example, Intel Larrabee[SCS+08] integrates scalar units and wide vector units and these units are in the same core. In addition, these units share instruction and data caches. CELL processor[PAB+05] integrates one general puporse processor called "PowerPC Processor Element(PPE)" and eight SIMD processors called "Synergistic Processor Element(SPE)". On-chip interconnection network called "Element Interconnect Bus(EIB)" connects the PPE and SPEs. A DMA controller in the SPE performs data transfer between the PPE and the SPEs. GPU[LHG+06] are connected with off-chip interconnection network called PCI Express.

Therefore, this chaper firstly defines a generic architecture in order to build the automatic parallelizing compilation flow, which can support various kinds of shared memory heterogeneous and homogeneous multicore configurations.

The rest of this chapter is organized as follows. Section 2.2 describes the detail of the architecture. Section 2.3 defines a program execution model on the architecture. Section 2.4 defines a clear distinction between the role of the OSCAR compiler and toolchains for accelerators. Section 2.6 concludes this chapter.

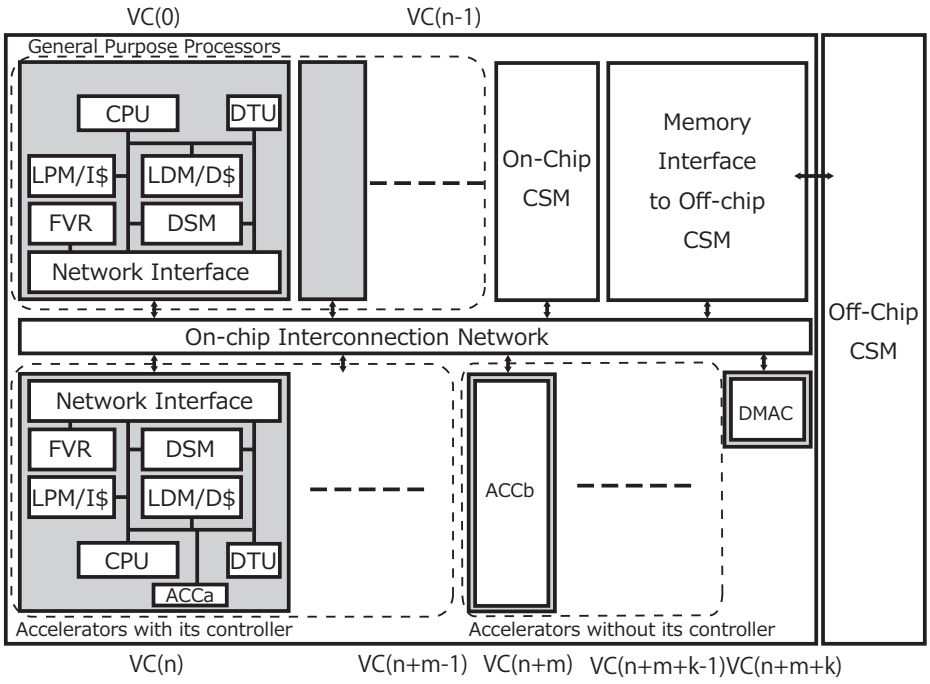


Figure 2.1: OSCAR Applicable heterogeneous multicore architecture

2.2 Target architecture

This section defines generic "OSCAR Applicable Heterogeneous Multicore Architecture" shown in Fig.2.1. This thesis defines a term "controller" as a general-purpose processor that controls an accelerator, that is to say, it performs part of coarse-grain task and data transfers from/to the accelerator and offload the task to the accelerator. The architecture is composed of general-purpose processors, accelerators(ACCs), direct memory access controller(DMAC), on-chip centralized shared memory(CSM), and off-chip CSM. Some accelerators may have its controller, or general-purpose processor. Both general-purpose processors and accelerators with controller may have a local data memory (LDM), a distributed shared memory (DSM), a data transfer unit (DTU), a frequency voltage control

registers (FVR), an instruction cache memory and a data cache memory. The local data memory is a high-speed memory and keeps processor private data. The distributed shared memory is middle-speed memory and is a dual port memory, which enables point-to-point direct data transfer and low-latency synchronization among processors. The data transfer unit in general-purpose processors and accelerator with controller is a kind of DMA controller , which is able to overlap task executions with data transfers. In addition, the frequency voltage control registers in each core improves power efficiency by using dynamic voltage and frequency scaling(DVFS).

Each existing heterogeneous multicore can be seen such as CELL BE[PAB+05], MP211[TST+05] and RP1[YKH+07] as a subset of OSCAR API applicable architecture. Thus, OSCAR API can support such chips and a subset of OSCAR API applicable heterogeneous multicore. The minimum subset of the architecture is a cache-based architecture, which means that each general-purpose processor just consists of CPU core, instruction cache, and data cache. When the target architecture has a local memory, the OSCAR compiler performs local memory management for hard real-time execution. Addition of a data transfer unit and a distributed shared memory accelerates data transfer among memories and synchronization among processors, respectively. Introducing accelerator with its controller realize higher performance than accelerator without its controller because the former can feed data to accelerator with lower cost. The full architecture showin in Fig.2.2 is called the OSCAR compiler cooperative heterogeneous multicore architecture because the OSCAR compiler is able to utilize the data transfer units, the local memory and the distributed shared memory[NMM+09], accelerator with its controller and change the frequency and the voltage of the chip.

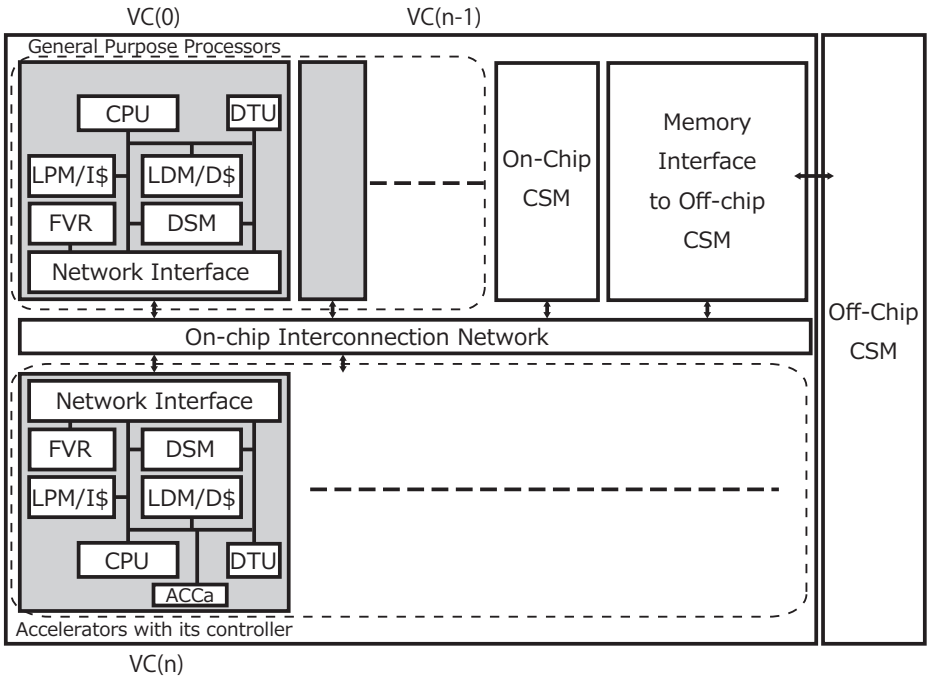


Figure 2.2: OSCAR compiler cooperative heterogeneous multicore architecture

2.2.1 Identifying element in the architecture

This thesis also defines a term "Virtual Core(VC) number" as an ID which identifies the processor element in the architecture.

VC number starts at zero and if there are "n" general-purpose processors, "m" accelerators with controler and "k" accelerators without controller, the numbering is the following:

(1) general-purpose processors: VC(0) - VC(n-1) in Fig.2.1

(2) accelerators with controller: VC(n) - VC(n+m-1) in Fig.2.1

(3) accelerators without controller: VC(n+m) - VC(n+m+k-1) in Fig.2.1

(4) the others: VC(n+m+k) - in Fig.2.1

```
[[chip, ] vc,] MODULE_ARG_LIST
 MODULE_ARG_LIST := MODULE_ARG, MODULE_ARG_LIST
 MODULE_ARG := (module([sub_module])[, arg_list])
```

Figure 2.3: Specifying the module within a chip

Table 2.1: Examples of Modules

module	corresponding module
OSCAR_LDM	Local Data Memory
OSCAR_DSM	Distributed Shared Memory
OSCAR_CSM	Centrilized Shared Memory
OSCAR_DTU	Data Transfer Unit
OSCAR_DMAC	Direct Memory Access Controller
OSCAR_ENTIRECORE	Entire Core
OSCAR_ENTIRECHIP	Entire Chip

In addition, this thesis also defines the way to identify modules in the processor core such as data transfer unit and memory. This notation is used to set the frequncy voltage of the module. Fig.2.3 shows the notation and Table.2.1 shows the exmaple of modules.

In Fig.2.3, “chip” is chip number, “vc” is VC number, “module” is module name, “sub_module” is submodule name and “arg_list” is the argument for the specified module. Here, parameters enclosed in “[ ]” can be omitted. If “chip” and “vc” is “-1”, the target module is whole chip, whole core respectively. By using VC number and this notation, any elements in the target architecture can

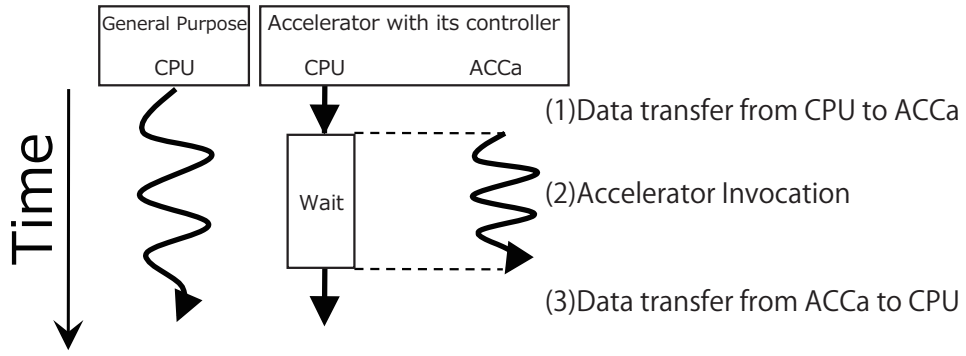


Figure 2.4: Execution Model on Accelerator with its Controller

be specified. For example, a local data memory in VC0 in Chip0 is expressed in "0, 0, OSCAR_LDM". "OSCAR_ENTIRECORE" specifies a VC as a whole. In addition, "OSCAR_ENTIRECHIP" specifies a chip as whole.

2.3 Execution model on the architecture

This section defines a program execution model on the target architecture. In order to minimize the runtime overhead of parallel processing, the OSCAR compiler adopts one-time single level thread creation as its execution model. This execution model creates threads at the program start point. Execution model of accelerator depends on whether accelerator has its own controller or not. Fig.2.4,Fig.2.5 shows the execution model of accelerator.

As shown in Fig.2.4, in accelerator with its controller, the controller performs data transfers and accelerator invocation. On the other hand, as shown in Fig.2.5, in accelerator without its controller, a general-purpose processors play a role as a controller.

In both cases, the task execution on controller is blocked until accelerator exe-

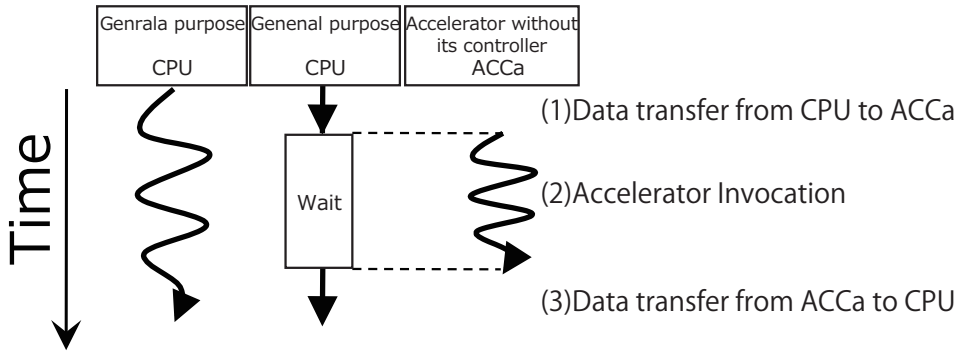


Figure 2.5: Execution Model on Accelerator without its Controller

cution ends.

2.4 Role of the OSCAR compiler and toolchains for accelerators

In order to utilize accelerators, programmers generally identify which parts of an input program is to be accelerated effectively, develop an accelerator binary using domain specific language such as CUDA[GGN+08] and OpenCL[khr] or assembler languages, and implements controller codes including data transfer between a general-purpose processor and an accelerator, accelerator invocations, synchronizations. However, supporting all accelerator programming model is not so feasible for the OSCAR compiler since these efforts are completely depend on the target accelerator. Therefore, it is important to define a clear distinction between the role of the OSCAR compiler and toolchains for accelerators not to lost the general-purpose properties.

This thesis defines an accelerator compiler has the following three properties.

- inserts hint directives immeadiately before loops or function calls, which can be executed on the accelearator, in a sequential program.
- generates accelerator binary of the task, which is assigned by the OSCAR compiler.
- generates accelerator control codes including data transfer codes, accelerator invocation codes and synchronization codes.

In contrast, this thesis defines the OSCAR compiler has the following three properties.

- performs coarse-grain task graph generation and task scheduling considering with hint directives.
- generates C source codes as separate files which include functions to be executed on accelerators when a function is scheduled onto accelerator by the OSCAR compiler.
- generates parallelized C or Fortran program with OSCAR API.

However, not all of existing accelerator compiler support these properties. For example, FE-GA[KTT+06] compiler developed by Hitachi suuports all properties. PGI Accelerator Compiler[Wol10] developed by Portrand Group does not support hint directive insertion. CUDA compiler developed by NVIDIA only supports accelerator binary generation. In order to utilize such accelerator compilers and existing hand-tuned libraries, the proposed framework provides another way. In the way, programmers prepare libraries which include controllers codes and accelerator binary in advance and inserts hint directives manually.

2.5 Related work

There are many examples of heterogeneous multicores. This section shows an overview of the existing heterogeneous multicores.

IBM/SONY/TOSHIBA CELL BE[PAB+05] integrates one general-purpose processor called PPE and eight SIMD accelerator called SPE. PPE and SPEs are connected to on-chip interconnection. SPE incorporates a local memory called "Local Storage" and a direct memory access controller called "Memory Flow Controller(MFC)". Data transfers between main memory and local storage are always done by software. SPE is an accelerator without its controller and PPE plays a role of controller for SPE. Arizona State University/ARM SODA[LLW+06] is designed for software defined radio and is similar to CELL BE. SODA made up of four SIMD cores and one general-purpose ARM processor. Each SIMD core has a local memory and DMAC and is connected to on-chip interconnection network. SIMD core is an accelerator without its controller and ARM processor plays a role of controller for SIMD core. Intel Larrabee[SCS+08] core consists of a scalar unit and a vector unit and these units are connected to intra core network. Larrabee core is accelerator with its controller because these units share L2 cache. University of Michigan/Arizona State University/ARM AnySP[WSM+09] core is called PE and consists of SIMD unit and scalar unit. Both SIMD unit and scalar unit share a local data memory and DMAC. AnySP PE is an accelerator with its controller because these units are also connected to intra core network. Renesas proposed nine general-purpose processors and two matrix processors and these processors are interconnected with on-chip interconnection network. A matrix processor is an accelerator without its controller and general-purpose processors plays a role of controller. University of California at Berkeley VIRAM[KP03] integrates one

general-purpose MIPS core and two vector arithmetic functional units. Both MIPS core and vector units are connected to on-chip interconnection network and a vector unit is an accelerator without its controller. GPGPU such as NVIDIA Tesla GPU, AMD Radeon GPU and Intel Graphic Media Accelerator are a massively parallel processor and is often connected to PCI Express. GPGPU is obviously an accelerator without its controller as well. Some GPGPUs are tightly coupled with general-purpose processors such AMD Fusion GPUs. Universitat Politecnica de Catalunya/University of Illinois at Urbana-Champaign CUBA is an architecture, which includes data-parallel accelerators and general-purpose processors. CUBA accelerator is an accelerator with its controller. Renesas/Hitachi/Tokyo Institute of Technology/Waseda University RP-X integrates eight general-purpose SH-4A processors, four dynamically reconfigurable FE-GA processors, two matrix processors and the other media IPs. Each SH-4A core consists of a local data memory, a local program memory, a distributed shared memory, a data transfer unit and frequency voltage control register. FE-GA is an accelerator without its controller.

In summary, all processor element in heterogeneous architecture can be categorized into general-purpose processor, accelerator with its controller and accelerator without its controller. general-purpose processors and accelerator with its controller would have a local data memory, a local program memory, a distributed shared memory, a data transfer unit and frequency voltage control register. This is why the existing heterogeneous multicore architecture can be seen as a subset of OSCAR Applicable heterogeneous multicore architecture.

2.6 Conclusion

This chapter has defined a generic architecture in order to build the compilation flow of the OSCAR compiler, which can support various kinds of shared memory multiprocessor configurations. The model multicore architecture for the OSCAR compiler is composed of general-purpose processors, accelerators, direct memory access controller, on-chip centralized shared memory, and off-chip CSM. Moreover, OSCAR heterogeneous multicore architecture can handle accelerators without controllers and accelerators with their controllers, or general-purpose processors. Both general-purpose processors and accelerators with controller may have a local data memory, a distributed shared memory, a data transfer unit, frequency voltage control registers, an instruction cache memory and a data cache memory. The local data memory keeps private data. The distributed shared memory is a dual port memory, which enables point-to-point direct data transfer and low-latency synchronization among processors. This chapter also has shown most of existing heterogeneous and homogeneous multicore can be seen as a subset of the OSCAR architecture and the OSCAR compiler can support various heterogeneous and homogeneous multicores. In addition, this chapter has defined a clear distinction between the role of the OSCAR compiler and toolchains for accelerators and a program execution model on the architecture.

Chapter 3
Compilation Framework for the Heterogeneous Multicore Architecture

3.1 Introduction

In recent years, homogeneous multicore processors and heterogeneous multicore processors have attracted much attention due to their excellent characteristic, higher performance and lower power consumption than single core processors.

However, programmers have to carefully find out which parts of a program can be parallelized and decompose the program into tasks manually and schedule these tasks onto processor cores so as to exploit the full capability of homogeneous and heterogeneous multicore processors. Specifically, programmers parallize the program by using pthread and OpenMP[Opec] for shared memory multicore processors, Message Passing Interface(MPI)[sta] for distributed memory multicore processors, and domain specific languages such as OpenCL[khr] for accelerators. Programmers often confront with many difficulty such as race conditions, dead locks. Therefore, long software development periods are required even for expert programmers.

Recent many studies have tried to handle this software development issue. In terms of homogeneous multicores, many works have been trying to realize an automatic parallelization. Polaris compiler[EHP98], SUIF compiler[HAA+96], IBM XL compiler and Intel compiler are an example of automatic parallelizing compilers. However, these parallelizing compilers only exploit the loop level parallelism and are designed for homogeneous multicores.

In terms of heterogeneous multicores, researchers came up with many solution so as to facilitate the difficulty heterogeneous of parallel programming. For example, NVIDIA and Khronos Group introduced CUDA[GGN+08] and OpenCL[khr]. Also, PGI accelerator compiler[Wol10] and HMPP[DBB07] provides a high-level programming model for accelerators. However, these works focus on facilitating

the development for accelerators. Programmers need to distribute tasks among general-purpose processors and accelerator cores by hand. In terms of workload distribution, Qilin[LHK09] automatically decides which task should be executed on a general-purpose processor or an accelerator at runtime. However, programmers still need to parallelize a program by hand. While these works rely on programmers' skills, CellSs[BPBL09] performs an automatic parallelization of a subset of sequential C program with data flow annotations on CELL BE. CellSs automatically schedules tasks onto processor elements at runtime. The task scheduler of CellSs, however, is implemented as a homogeneous task scheduler, namely the scheduler is executed on PPE and just distributes tasks among SPEs.

In the light of above facts, further explorations are needed since it is the responsibility of programmers to parallelize a program and to optimize a data transfer and a power consumption for heterogeneous multicores.

The goal is to realize a fully automatic parallelization of a sequential C or Fortran77 program for heterogeneous multicores. Unlike conventional parallelizing compilers, the OSCAR paralleling compiler exploits multi-level parallelism for homogeneous multicores such as SMP servers and real-time multicores[KOI00, KMM$^+$09, MOKK10]. The OSCAR compiler realizes automatic parallelization of programs written in Fortran77 or Parallelizable C, a kind of C programming style for parallelizing compiler, and power reduction with the support of both the OSCAR compiler and OSCAR API(Application Program Interface)[kas], which supports partioned global address space(PGAS) including local memory, distributed shared memory centrized shared memory, DMA controller. This thesis realizes an automatic parallization and a power reduction for the generic architecture defined in Chapter 2.

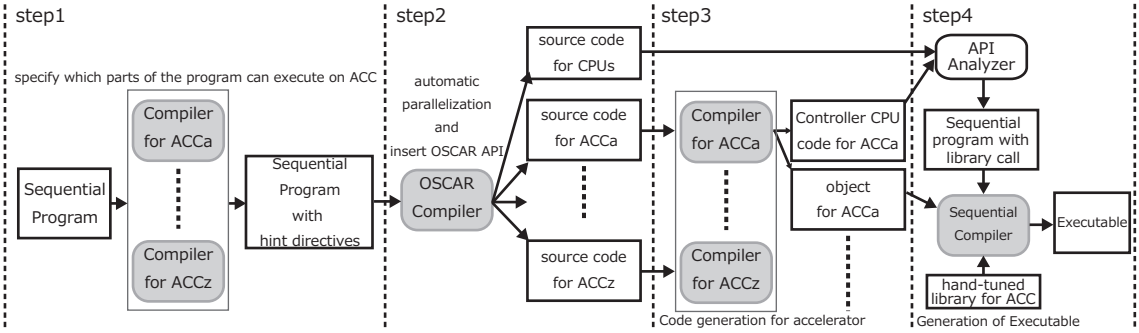


Figure 3.1: Compilation flow of the proposed framework

The rest of this chapter is organized as follows. Section 3.2 provides the compilation flow based on the OSCAR compiler and toolchains for accelerators. Section 3.3 explains the hint directives for the OSCAR compiler. Section 3.4 describes the OSCAR compiler. Section 3.5 describles OSCAR API. Section 3.7 concludes this chapter.

3.2 Compilation flow

Fig.3.1. shows the compilation flow of the proposed OSCAR heterogeneous compiler framework. The input is a sequential program written in Parallelizable C or Fortran77 and the output is an executable for a target heterogeneous multicore. The following describes each step in the proposed compilation flow.

Step 1: Accelerator compilers or programmers insert hint directives immediately before loops or function calls , which can be executed on the accelerator, in a sequential program.

Step 2: the OSCAR compiler parallelizes the source program considering with hint directives: the compiler schedules coarse-grain tasks[WHM+08] to pro-

cessor or accelerator cores and apply the low power control[KMM+09]. Then, the compiler generates a parallelized C or Fortran program for general-purpose processors and accelerator cores by using OSCAR API. At that time, the compiler generates C source codes as separate files for accelerator cores. Each file includes functions to be executed on accelerators when a function is scheduled onto accelerator by the compiler.

Step 3: Each accelerator compiler generates objects for its own target accelerator. Note that each accelerator compiler also generates both data transfer code between controller and accelerator, and accelerator invocation code.

Step 4: An API analyzer prepared for each heterogeneous multicore translates OSCAR APIs into runtime library calls, such as pthread library. Afterwards, an ordinary sequential compiler for each processor from each vender generates an executable.

It is important that the framework also allows programmers to utilize existing hand-tuned libraries for the specific accelerator. This paper defines a term "hand-tuned library" as an accelerator library which includes computation body on the specific accelerator and both data transfer code between general-purpose processors and accelerators and accelerator invocation code.

3.3 A hint directive for the OSCAR compiler

This section explains the hint directives for the OSCAR compiler that advice the OSCAR compiler which parts of the program can be executed by which accelerator core.

```
oscar_hint accelerator_task (accelerator_type) \
           cycle(number, [((trans_mode))]) \
           [workmem(memory_name, number)] \
           [in(list of input variables)] [out(list of output variables)]
           new-line
oscar_comment "string" new-line
```

Figure 3.2: Hint directives for the OSCAR compiler

The list of hint directives is shown in the Fig3.2. Here, parameters enclosed in "[ ]" can be omitted. The first one is "accelerator_task" directive which indicates specified block(e.g. loop, subroutine, basic block) is able to be executed on the specified accelerator. "accelerator_type" specifies the name of the accelerator(e.g. GPU, FE-GA[KTT+06]). "cycle" specifies the number of clock cycles required for the block on the accelerator. "trans_mode" specifies the device transferring between general-purpose processors and accelerators(e.g. DMAC, DTU). If "trans_mode" is omitted, the controller performs the data transfer from/to the accelerator. Note that the cost of a data transfer is included in the number of clock cycles. Some accelerator compiler might allocate the memory for variables that are to be used to control the accelerator. "workmem" specifies the kind of memory utilized and the size of variables allocated by the accelerator compiler. "in" and "out" annotate input/output variables of the block. The second one is "oscar_comment" directive. This directive is inserted so that either programmers or accelerator compilers give a comment to accelerator compiler through the OSCAR compiler. This comment is helpful for the accelerator compiler to optimize the accelerator code. For example, programmers may insert the directive for an

```
int main() {
  int i, x[N], var1 = 0;
  /* loop1 */
  for (i = 0; i < N; i++) { x[i] = i; }
  /* loop2 */
#pragma oscar_hint accelerator_task (ACCa) \
                    cycle(1000,((OSCAR_DMAC()))) workmem(OSCAR_LDM(), 10)
  for (i = 0; i < N; i++)  { x[i]++; }
  /* function3 */
#pragma oscar_hint accelerator_task (ACCb) \
                    cycle(100, ((OSCAR_DTU()))) in(var1,x[2:11]) out(x[2:11])
  call_FFT(var1, x);
  return 0;
}
```

```
void call_FFT(int var, int* x) {
#pragma oscar_comment "XXXXX"
  FFT(var, x);  //hand-tuned library call
}
```

Figure 3.3: Example of source code with hint directives

accelerator compiler like PGI accelerator compiler.

Fig.3.3. shows an example code. As shown in Fig.3.3., there are two types of hint directives inserted to a sequential C program, namely "accelerator_task" and "oscar_comment". In this example, there are "#pragma oscar_hint accelerator_task (ACCa) cycle(1000, ((OSCAR_DMAC()))) workmem(OSCAR_LDM(), 10)" and "#pragma oscar_hint accelerator_task (ACCb) cycle(100, ((OSCAR_DTU()))) in(var1, x[2:11]) out(x[2:11])". In these directives, accelerators represented as "ACCa" and "ACCb" is able to execute a loop named "loop2" and a function named "function3", respectively. The hint directive for "loop2" specifies that "loop2" requires 1000 cycles including the cost of a data transfer performed by DMAC if the loop is processed by "ACCa". This directive also specifies that 10 bytes in local data memory are required in order to control "ACCa". Similarly, for "function3", it takes 100 cycles including the cost of a data transfer by DTU. Input variables are scalar variable "var1" and array variable "x" ranging 2 to 11. Also, output

variable is array variable “x”. “oscar_comment” directive is inserted so that either programmers or accelerator compilers give a comment to accelerator compiler through the OSCAR compiler.

3.4 OSCAR parallelizing compiler

This section describes the OSCAR compiler. The OSCAR compiler exploits multi-level parallelism including coarse grain parallelism, loop level parallelism, and fine grain parallelism. In addition, the OSCAR compiler is the only compiler, which is able to apply power control to the target architecture, by inserting a dynamic voltage frequency scaling(DVFS) instruction for power efficiency.

3.4.1 Overview of the OSCAR compiler

As shown in Fig.3.4 the OSCAR compiler consists of the three phases: front end(FE), middle path(MP), and back end(BE). The front end performs lexical analysis and syntax analysis and generates intermediate representation(IR). The middle path performs many parallelization and power reduction techniques including control flow analysis, data dependency analysis, task generation, task scheduling and generate intermediate representation. The back end generates a binary for a target machine or a C/Fortran source code with OSCAR API[KMM+09].

Specifically, the middle path performs the following scheme:

- Macro-Task Generation
- Exploiting Coarse Grain Parallelism
- Processor Grouping

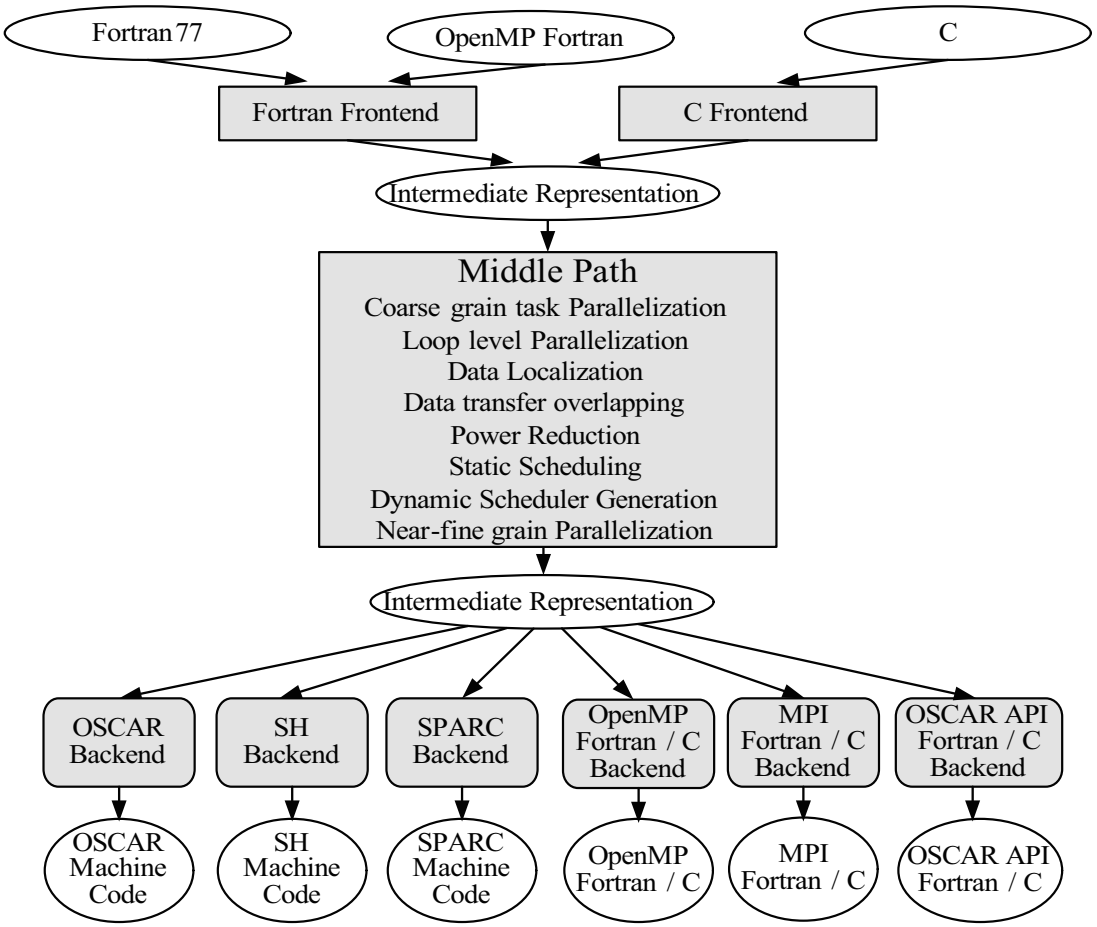


Figure 3.4: OSCAR multigrain parallelizing compiler

- Macro-Task Scheduing and Power Reduction

The detail is described in the Section 3.4.2, 3.4.3, 3.4.4, 3.4.5.

3.4.2 Macro-Task generation

The compiler decomposes a program into coarse grain tasks, namely macro-tasks (MTs), such as basic block (BPA), loop (RB), and function call or subroutine call (SB).

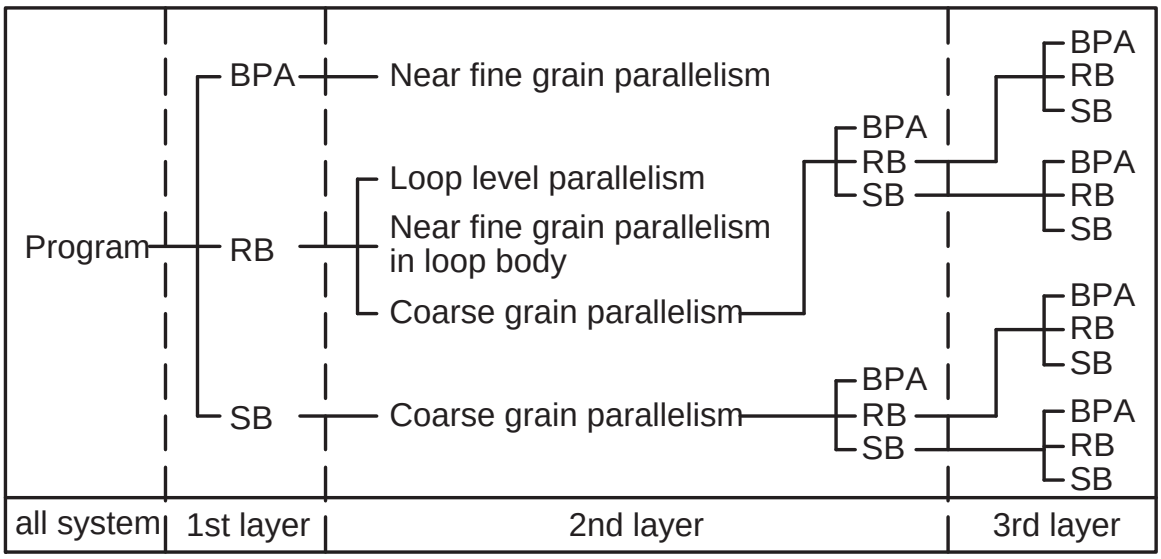


Figure 3.5: Hierarchical macro-task definition

Macro-tasks can be hierarchically defined inside each sequential loop, which can't be parallelized, and a subroutine block as shown in Figure 3.5. Repeating this macro-task generation hierarchically, a original program is decomposed into nested macro-tasks as in Figure 3.5.

3.4.3 Exploiting coarse grain parallelism

After generation of macro-tasks, data dependencies and control flow among macro-tasks are analyzed in each nested layer, and hierarchical macro-flow graphs (MFGs) representing control flow and data dependencies among macro-tasks are generated as shown in Figure 3.6(a)[HIK90, KHM+91, Kas03]. In this figure, nodes represent macro-tasks, solid edges represent data dependencies among macro-tasks and dotted edges represent control flow. A small circle inside a node represents a conditional branch.

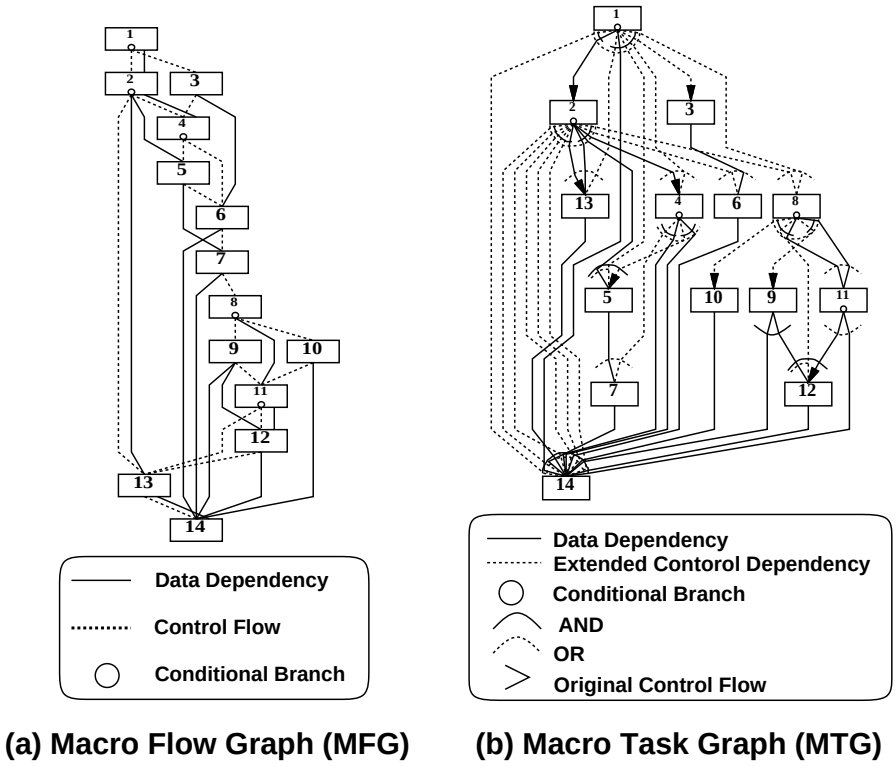


Figure 3.6: Macro-flow graph and macro-task graph

Next, to extract coarse grain task parallelism among macro-tasks, Earliest Executable Condition analysis [HIK90, KHM+91, Kas03, KHIH90, KHN90] is applied to each macro-flow graph. It analyzes control dependencies and data dependencies among macro-tasks simultaneously and determines the conditions on which macro-tasks may begin their execution earliest. By this analysis, a macro-task graph (MTG) [HIK90, KHM+91, Kas03, KHIH90] is generated for each macro-flow graph. This graph represents coarse grain task parallelism among macro-tasks. Here, nodes in a macro-task graph are macro-tasks, a small circle is conditional branch and solid edges represent data dependencies. Dotted edges represent extended control dependencies. Extended control dependency means an ordinary

control dependency and the condition on which a data dependent predecessor macro-task is not executed. A solid arc represents that edges connected by the arc are in AND relationship. A dotted arc represents that edges connected by the arc are in OR relation ship.

For example, the earliest executable condition of macro-task 6 in Figure 3.6(b) is "the conditional branch of macro-task 2 jumps to macro-task 4 or the execution of macro-task 3 is finished."

When the compiler cannot analyze the input source for some reason, like hand-tuned accelerator library call, "in/out" clause of "accelerator_task" gives the data dependency information to the OSCAR compiler. Then, the compiler calculates the cost of MT and finds the layer which is expected to apply coarse-grain parallel processing most effectively. "cycle" clause of "accelerator_task" tells the cost of accelerator execution to the compiler.

3.4.4 Processor grouping

The OSCAR compiler logically divides given processors into processor group(PG) in order to execute hierarchical macro-task graphs efficiently. A processor group consists of processor elements(PE), which are equivalent to physical general-purpose processors. A coarse grain task(MT) is assigned to processor group. On the other hand, a loop iteration and a near fine grain task are assigned to a processor element. Processor grouping is applied recursively so as to exploit the many kinds of parallelism in a whole program. That is why the OSCAR compiler realizes the multi grain parallel processing. Because the number of accelerators are normally fewer than the number of general-purpose processors, accelerators are independent from the processor grouping. In other words, macro task in any layer is assigned

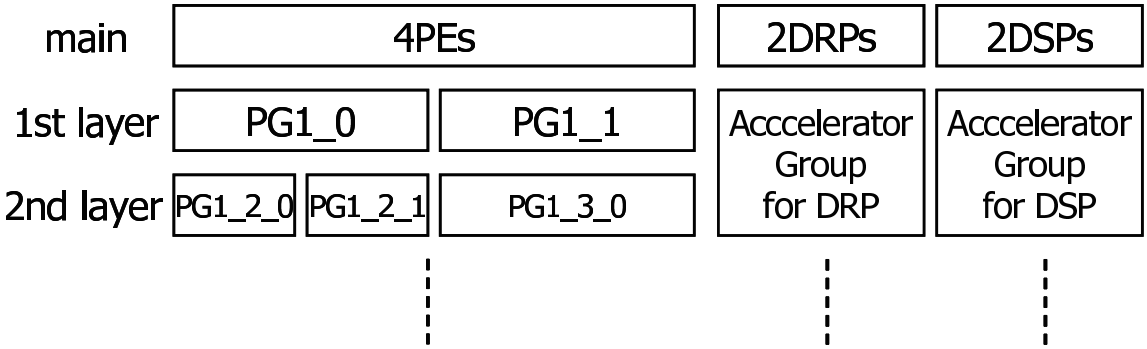


Figure 3.7: Processor Grouping on Heterogeneous Multicores

to accelerators.

Fig.3.7 shows an example of processor grouping for the architecture which integrates four general-purpose processors, two dynamically reconfigurable processors and two digital signal processors. As shown in Fig.3.7, general-purpose processors are divided into two processor groups(PG1_0 and PG1_1) in the first layer, which means each processor group has two processor elements. In the second layer, PG1_0 is divided into two processor groups(PG1_2_0 and PG1_2_1) in order to perform coarse grain parallel processing. On the other hand, PG1_1 is not divided into new processor group because there are loop level or fine grain parallelism.

3.4.5 Macro-Task scheduling and power reduction

The task scheduler of the compiler statically schedules macro-tasks to each core[WHM+08].

The scheduling algorithm are the following[WHM+08].

Step 1. Preparation.

Step 1-1. Calculate the cost of macro-task on a general-purpose processor

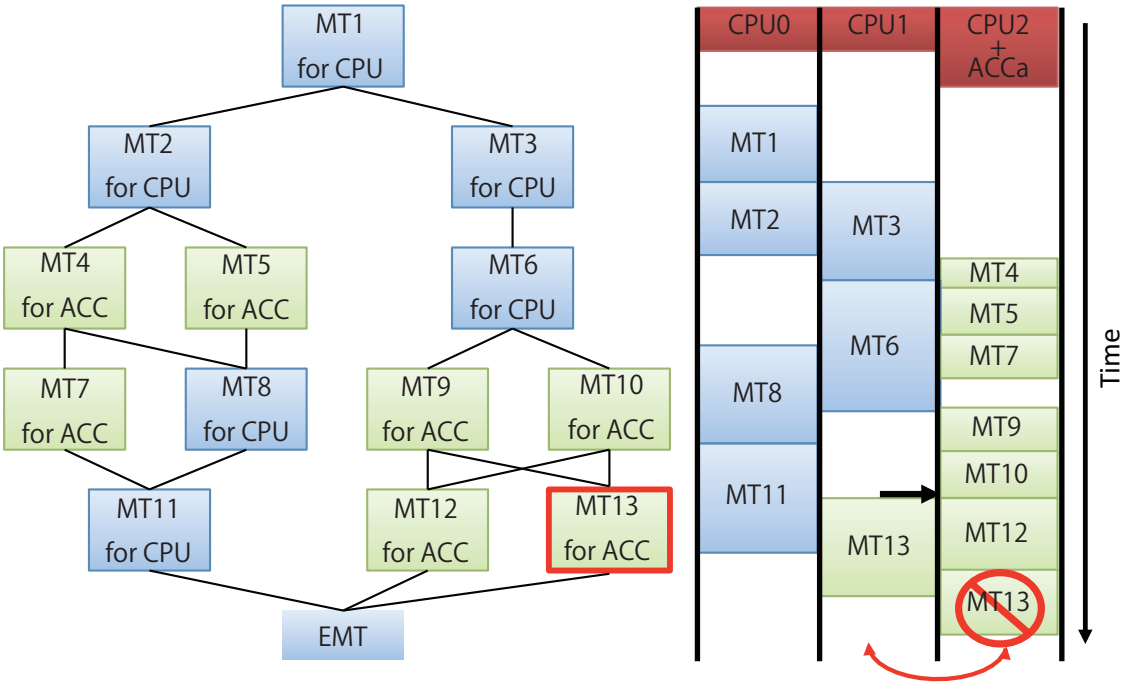


Figure 3.8: An Example of Task Scheduling Result

and each accelerator.

Step 1-2. Calculate the critical path length as the scheduling priority.

Step 2. Initialization.

Step 2-1. Set the scheduling time to zero.

Step 2-2. Add the 1st layer macro-task graph to the list of macro-task graphs which is under scheduling process.

Step 3. Get ready macro-tasks from the list in Step 2-2. Ready tasks satisfy earliest executable condition[KHM+91] at current scheduling time. If there is no ready Macro-Task, then go to Step 8.

Step 4. Select a macro-task to be scheduled from the ready macro-tasks according to the priorities.

Step 5. Estimate execution completion time of the macro-task on

- each general-purpose processor which is free at current scheduling time
- each accelerator which can execute the macro-task

Step 6. Assign the macro-task to a general-purpose processor or an accelerator which gives the earliest completion time.

Step 7. If the assigned macro-task has macro-task graphs to be applied coarse grain parallel processing inside, add it to the list in Step 2-2.

Step 8. Updating the scheduling time.

Step 8-1. Update the scheduling time until the time when a macro-task is completed next.

Step 8-2. If there is a macro-task graph that all of the Macro-Tasks inside have been completed at the updated scheduling time, remove it from the list in Step 2-2.

Step 8-3. If all of the macro-tasks are completed, then exit. If not, then go to Step 3.

Fig.3.8. shows an example of heterogeneous task scheduling result. First the scheduler gets ready macro-tasks from MTG(MT1 in Fig.3.8 in initial state). Ready tasks satisfy earliest executable condition[KHM+91]. Then, the scheduler selects a macro-task to be scheduled from the ready macro-tasks and schedules the macro-task onto general-purpose processor or accelerator considering data transfer

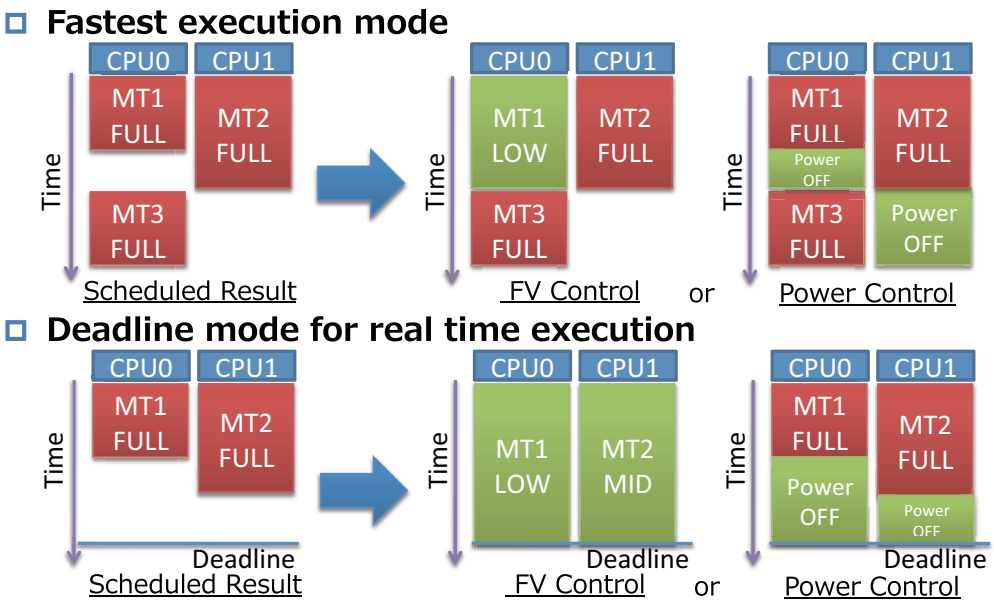


Figure 3.9: Power control by compiler

overhead, according to the priorities, namely CP length. The scheduler performs above sequences until all macro-tasks are scheduled. Note that a task for an accelerator is not always assigned to the accelerator. At this case, the task may be assigned to general-purpose processor to minimize total execution time.

After task scheduling, the compiler tries to minimize total power consumption by changing frequency and voltage(DVFS) or shutting power down the core during the idle time considering transition time[SOW+07]. The OSCAR compiler assumes a target architecture has the following properties:

- Frequency of each processor can be changed in several levels such as FULL, MID and LOW
- Voltage can be changed with Frequency
- Each processor can be powered on and off individually

- Frequency, voltage and power status can be changed by software, which means the OSCAR compiler controls frequency voltage control register shown in Fig.2.1

As shown in Fig.3.9, there are two modes for compiler-controlled power saving. The first one is called “Fastest Execution Mode” and the compiler determines suitable voltage and frequency for a task which does not belong to critical path in this mode. In this example, there is an idle time after MT1 execution on CPU0. The OSCAR compiler applies “FV Control” or “Power Control” for the idle time. “FV Control” means MT1 is executed in LOW mode. On the other hands, “Power Control” means CPU0 is powered off after MT1 finishes in order to avoid leakage power dissipation. The OSCAR compiler selects the best control to minimize power consumption. The second one is called “Deadline mode for real time execution” and the compiler determines suitable voltage and frequency for each macro-task based on the result of static task assignment in order to satisfy the deadline for real-time execution. The OSCAR compiler also applies “FV Control” or “Power Control” and selects the best control. In Fig.3.9., FULL is 648MHz, MID is 324MHz, and LOW is 162MHz respectively. Each of which is applicable in RP-X processor described in Section 4.

Finally, the compiler generates parallelized C or Fortran program with OSCAR API. The OSCAR compiler generates the function which includes original source for accelerator. Generation of data transfer codes and accelerator invocation code is responsible for accelerator compiler.

The OSCAR compiler uses processor configurations, such as number of cores, cache or local memory size, available power control mechanisms, and so on. This information is provided by compiler options.

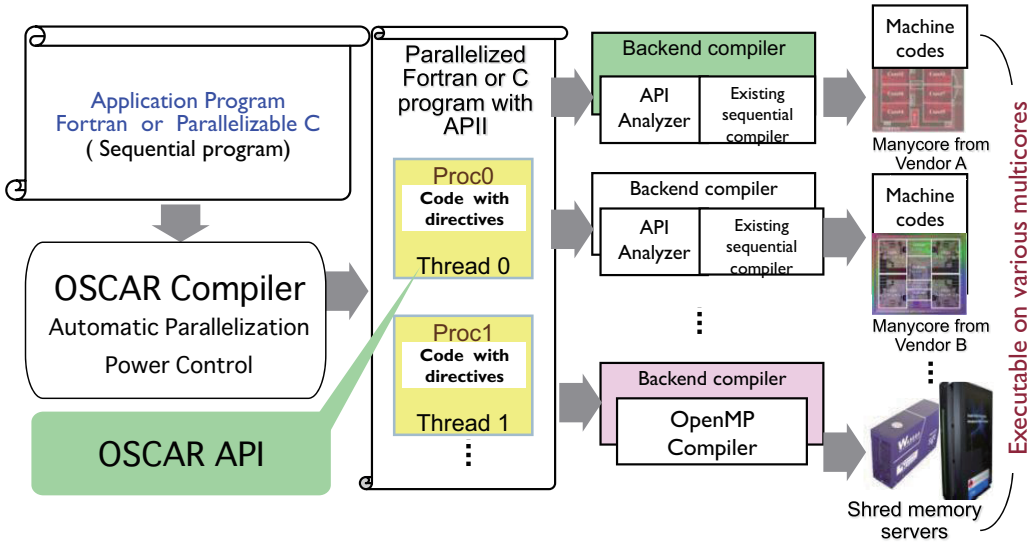


Figure 3.10: Compilation flow of OSCAR API

3.5 OSCAR API

This section describes OSCAR API in detail.

3.5.1 Overview of OSCAR API

This subsection describes an overview of OSCAR API. Fig.3.10 shows a brief overview of the compilation flow using OSCAR API. As described in Section 3.2, The OSCAR compiler generates the parallelized Fortran or C program with OSCAR API and API analyzer translates OSCAR APIs into runtime library calls, such as pthread library. Afterwards, an sequential compiler generates an executable. That's why OSCAR API is multiplatform multicore API. Fig.3.11 shows the list of OSCAR API. OSCAR API consists of parallel execution APIs, memory mapping APIs, data transfer APIs, power control APIs, synchronization APIs, timer APIs, cache control APIs and accelerator APIs. Note that the notation

- Parallel Execution API
  - parallel sections
  - flush
  - critical
  - execution
- Memory Mapping API
  - threadprivate
  - distributedshared
  - onchipshared
- Data Transfer API
  - dma_transfer
  - dma_contiguous_parameter
  - dma_stride_parameter
  - dma_flag_check
  - dma_flag_send
- Cache Control API
  - cache_writeback
  - cache_selfinvalidate
  - complete_memop
- Power Control API
  - fvcontrol
  - get_fvstatus
- Synchronization API
  - groupbarrier
- Timer API
  - get_current_time
- Accelerator API
  - accelerator_task_entry

Figure 3.11: API List of OSCAR API 2.0

described in Section 2.2.1 for specifying modules in the target architecture.

Parallel execution API

Parallel execution APIs support thread creation and mutual exclusion by using "parallel sections" API, "flush" API, "critical" API on the target platform. All API except "execution" API derived from OpenMP. "execution" API means specified function is executed by the specified VC.

Memory mapping API

Memory mapping APIs enable the OSCAR compiler to map variables and arrays to specified memory. "threadprivate" API, "distributedshared" API, "onchipshared" API map specified variables and arrays to a local memory, distributed shared memory, and on-chip centralized shared memory respectively.

Data transfer API

The OSCAR compiler also inserts codes which perform data transfer by using data transfer unit. “dma_contiguous_parameter” API for a contiguous transfer and “dma_stride_parameter” API for a stride transfer means a data transfer among processors and memories. These APIs are enclosed by “dma_tranfer” API, The completion of the transfer is to be notified and checked by using “dma_flag_check” and “dma_flag_send”.

Power control API

Frequency and voltage of chip can be changed and monitered by using “fvcontrol” and “get_fvstatus”. For example, “#pragma fvcontrol (0, 0, OSCAR_ENTIRECORE, -1)” turn off VC0 in Chip0.

Synchronization API

Synchronization API realizes hardware-supported barrier synchronization for low-latency synchronization. “groupbarrier” API performs hardware barrier synchronization among specified VCs.

Timer API

Timer API provides an interface of timer unit. “get_current_time” API returns current timestep in the architecture. The returned timer value is usually used for real-time execution.

Cache control API

Cache control APIs supports non-coherent cache architechtures which do not have hardware supported cache coherent mechanism such as RP-2[IHY+08], RP-X[YIK+10] and Rigel[KJJ+09]. "cache_writeback" API writes back a cache line of specified variable. "cache_selfinvalidate" API invalidates a cache line of specified variable. "complete_memop" API ensures the completion of a memory operation.

Accelerator API

Accelerator API provides an interface between the OSCAR compiler and accelerator compilers. The detail is described in Section 3.5.2.

3.5.2 The Extension of OSCAR API for heterogeneous multicores

This subsection describes API extension for heterogeneous multicores to be the output of the OSCAR compiler. Thee extension is very simple. Only one directive "accelerator_task_entry" is added to OSCAR API. This directive specifies the function's name where general-purpose processor invokes an accelerator.

Let us consider an example where the compiler parallelizes the program in Fig.3.3. We assume a target multicore includes two general-purpose processors, one ACCa as an accelerator with its controller and one ACCb as an accelerator without its controller. One of general-purpose processors, namely CPU1, is used as controller for ACCb in this case. Fig.3.12. shows as example of the parallelized C code with OSCAR heterogeneous directive generated by the OSCAR compiler. As shown in Fig.3.12., functions named "MAIN_CPU0()", "MAIN_CPU1()" and

```
int main() {
#pragma omp parallel sections
  {
#pragma omp section
    { MAIN_CPU0(); }
#pragma omp section
    { MAIN_CPU1(); }
#pragma omp section
    { MAIN_CPU2(); }
  }
  return 0;
}
```

```
int MAIN_CPU1() {
  ...
  oscartask_CTRL1_call_FFT(var1, &x);
  ...
}
int MAIN_CPU2() {
  ...
  oscartask_CTRL2_call_loop2(&x);
  ...
}
```

Source Code for CPUs

```
#pragma oscar accelerator_task_entry controller(2) \
                                    oscartask_CTRL2_loop2
void oscartask_CTRL2_loop2(int *x) {
  int i;
  for (i = 0; i <= 9; i += 1) { x[i]++; }
}
```

Source Code for ACCa

```
#pragma oscar accelerator_task_entry controller(1) \
                                    oscartask_CTRL1_call_FFT
void oscartask_CTRL1_call_FFT(int var1, int *x) {
#pragma oscar_comment "XXXXX"
  oscarlib_CTRL1_ACCEL3_FFT(var1, x);
}
```

Source Code for ACCb

Figure 3.12: Example of parallelized source code with OSCAR API

"MAIN_CPU2()" are invoked in omp parallel sections. These functions are executed on general-purpose processors. In addition, hand-tuned library "oscartask_CTRL1_call_FFT()" executed on ACCa is called by controller "MAIN_CPU1()". "MAIN_CPU2" also calls kernel function "oscartask_CTRL2_call_loop2()" executed on ACCb. "accelerator_task_entry" directive specifies these two functions. "controller" clause of the directive specifies id of general-purpose CPU which controls the accelerator. Note that there exist "oscar_comment" directives at same place shown in Fig.3.3.. "oscar_comment" directives may be used to give accelerator specific directives, such as PGI accelerator directives, to accelerator compilers. Afterwards, accelerator compilers generates the source code for the controller and objects for the accelerator, interpreting these directives.

3.6 Related work

Recent many studies have tried to realize an automatic parallelization.

In terms of homogeneous multicores, University of Illinois at Urbana-Champaign Polaris compiler[EHP98] exploits loop level parallelism by using symbolic analysis, array privatization, interprocedual analysis. Stanford University SUIF compiler[HAA+96] exploits loop level parallelism and performs an optimization for data locality by using unimodular transform and affine partitioning[WL91, LCL99]. IBM XL compiler and Intel compiler also exploits loop level parallelism. However, these parallelizing compilers only exploit the loop level parallism and are designed for homogeneous multicores. Universitat Politecnica de Catalunya NANOS compiler[GMO+00] exploits multi-level parallelism from Fortran program. Waseda University OSCAR compiler also exploits multi-level parallelism from C or Fortran program[KOI00, KMM+09, MOKK10].

In terms of heterogeneous multicores, researchers came up with many solution so as to facilitate the difficulty heterogeneous of parallel programming. A core assignment strategy for Single-ISA heterogeneous multicore processsor is proposed to improve performance efficiency[KTR+04]. NVIDIA and Khronos Group introduced CUDA[GGN+08] and OpenCL[khr], a programming model for GPGPU. Programmers develop a program for accelerator by using these special-purpose languages. On the other hand, PGI accelerator compiler[Wol10] and CAPS HMPP[DBB07] compiler PGI, Cray, NVIDIA, and CAPS OpenACC[Opea] provide a high-level programming model for accelerators, which means that these compiler generates an accelerator binary from a C or Fortran program with annotation directives. FE-GA compiler[HWW+11] also generates an accelerator binary from a C program without any annotation. OmpSs[FPB+11] proposes OpenMP extensions to

deal with GPGPU. EXOCHI[WCC+07] provides a programming model that facilitates heterogeneous programming by extending OpenMP pragmas. Merge, which extends EXOCHI, also provides a programming model and dynamically chooses function-intrinsics for each processor to utilize all available heterogeneous processors. Qilin[LHK09] automatically decides which task should be executed on a general-purpose processor or an accelerator at runtime. CIGAR[KGM+07] allows programmer to map an application into CPU/FPGA platform by using profiling. CellSs[BPBL09] performs an automatic parallelization of a subset of sequential C program with data flow annotations on CELL BE. CellSs automatically schedules tasks onto processor elements at runtime. PEPPHER[BPT+11] provides the variant-based programming model and realize performance portability across architecture. StarPU[ATNW11] provides a task abstraction called "codelet" and schedules these codelet onto heterogeneous multicore as efficiently as possible.

3.7 Conclusion

This chapter has described the proposed framework including the OSCAR automatic parallelizing compiler, an accelerator compiler, and API analyzer and interfaces among them. The input of the framework is a sequential program written in Parallelizable C, a kind of C programming style for parallelizing compiler, or Fortran77 and the output is an executable for a target heterogeneous and homogeneous multicore. The compilation flow consists of 4 steps. First of all, accelerator compilers or programmers insert hint directives immediately before loops or function calls, which can be executed on the accelerator, in a sequential program. Then, the OSCAR compiler parallelizes the source program considering with hint directives. The compiler automatically performs (1) decomposition of a program into tasks, (2) scheduling these tasks onto general processors and accelerators by inserting synchronization, (3) data transfer, and (4) power control codes which controls the frequency and the voltage of the chip. After that, the compiler generates a parallelized C or Fortran program for general-purpose processors and accelerator cores by using OSCAR API, multicore application program interface (API) including thread APIs, memory mapping APIs, data transfer APIs and power control APIs. At that time, the compiler generates C or Fortran source codes as separate files for accelerator cores. Each file includes functions to be executed on accelerators when a function is scheduled onto accelerator by the OSCAR compiler. Afterwards, each accelerator compiler generates objects for its own target accelerator. Finally, an API analyzer prepared for each heterogeneous multicore translates OSCAR APIs into runtime library calls, such as pthread library. Afterwards, an ordinary sequential compiler for each processor from each vender generates an executable.

Chapter 4
Parallel Processing Schemes for Media Applications on the Heterogeneous Multicore Architecture

4.1 Introduction

Multimedia applications such as audio encoding/decoding, movie encoding/decoding, and image processing have been used for car navigation systems, cell phones, digital televisions in recent years. Furthermore, these applications have to be executed immediately with low power.

Heterogeneous multicore is greatly good platform due to its characteristic, higher performance and lower power consumption by utlizing accelerators.

The OSCAR compiler automatically parallelizes a C or Fortran program and applies power control for heterogeneous multicores as described in Chapter 2 and Chapter 3. This Chapter evaluates the performance of the proposed framework on 15 core heterogeneous multicore RP-X[YIK+10] using media applications.

The rest of this chapter is organized as follows. Section 4.2 explains the RP-X processor. Section 4.3 describes the details of evaluated applications. Section 4.4 evaluates the processing performance on the RP-X processor. Section 4.5 evaluates the power consumption on the RP-X processor. Section 4.6 concludes this chapter.

4.2 RP-X heterogeneous multicore for consumer electronics

This section describes the RP-X processor. The RP-X processor is composed of eight 648MHz SH-4A general-purpose processor cores and four 324MHz FE-GA accelerator cores, the other dedicated hardware IP such as matrix processor "MX-2" and video processing unit "VPU5", as shown in Fig.4.1.. Each SH-4A core consists of a 32KB instruction cache, a 32KB data cache, a 16KB local in-

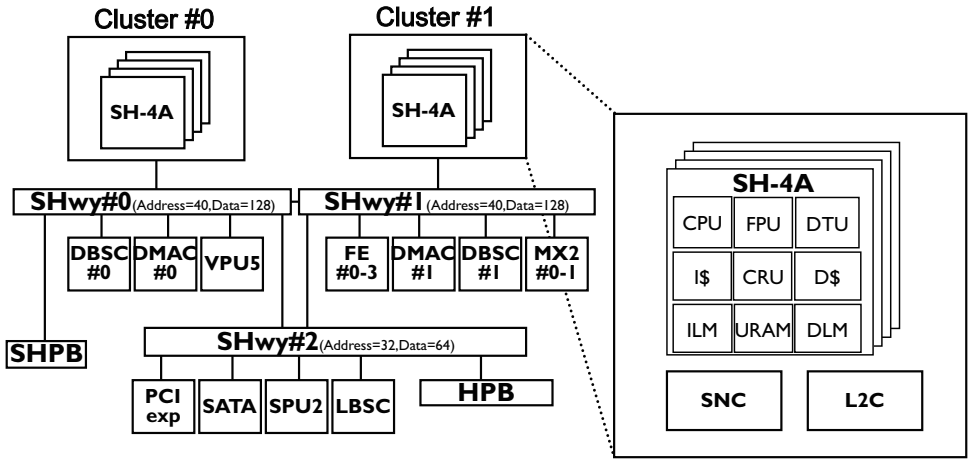


Figure 4.1: RP-X heterogeneous multicore for consumer electronics

Table 4.1: Frequency Voltage Status in SH-4A

	Frequency	Voltage
FULL	648MHz	1.3V
MID	324MHz	1.1V
LOW	162MHz	1.0V
VLOW	81MHz	1.0V

struction/data memory(ILM and DLM in Fig.4.1.), a 64KB distributed shared memory(URAM in Fig.4.1) and a data transfer unit. Furthermore, FE-GA is used as an accelerator without controller because FE-GA is directly connected with on-chip interconnection network named "SHwy#1", a split transaction bus. With regard to the power reduction control mechanism of RP-X, DVFS and clock gating for each SH-4A core can be controlled independently using special power control register by a user. This hardware mechanism is low overhead, for example fre-

quency change needs a few clocks. Table 4.1 shows the frequency voltage status in SH-4A. As shown in Table 4.1, “FULL”, “MID”, “LOW”, and “VLOW” means 648MHz on 1.3V, 324MHz on 1.1V, 162MHz on 1.0V, 81MHz on 1.0V respectively. DVFS for FE-GAs cannot be applicable.

This paper evaluates both generating the object code by accelerator compiler and using the hand-tuned library on RP-X processor.

4.3 Evaluated media applications

This section explains the detail of the evaluated media application. AAC encoding program, optical flow from OpenCV[opeb], optical flow from Hitach Ltd and Tohoku university are used for the evaluation.

4.3.1 AAC encoding

AAC encoding program is based on the AAC-LC encode program provided by Renesas Technology and Hitachi Ltd. The algorithm is a type of an audio compression system. The input is a PCM file and the output is a compressed AAC format file. The program is modified in Parallelizable C[MOKK10].

Fig.4.2 shows the program structure of the AAC encoder. As shown in Fig.4.2, this program consists of filter bank, midside(MS) stereo, quantization and huffman coding. These processes are applied to each frame.

The OSCAR compiler parallelizes the main loop which encodes a frame. The hand-tuned library for filter bank, MS stereo and quantization is used for FE-GA. Data transfer between SH-4A and FE-GA is performed by DTU via distributed shared memory.

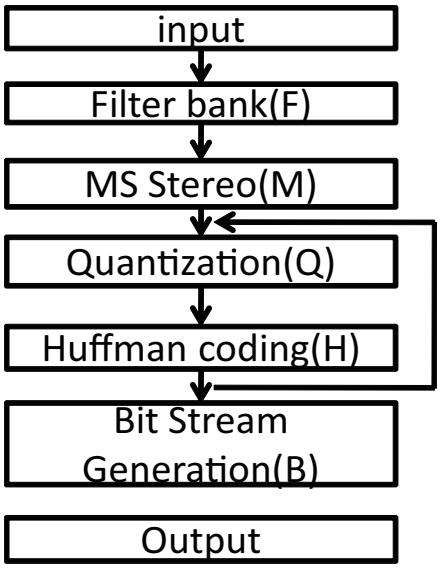


Figure 4.2: Program Structure of the AAC Encoder

4.3.2 Optical flow(OpenCV)

Optical flow program is from OpenCV[opeb]. The algorithm is a type of an object tracking system. The input is two images and the output is a velocity field between two images . The program is modified in Parallelizable C[MOKK10].

Fig.4.3 shows the program structure of the optical flow. The algorithm divides a image into small block. Then for every block in the first image the algorithm tries to find a block of the same size in the second image that is most similar to the block in the first image[Cor].

The block size is 16x16 pixels and sum of absolute difference calculation(SAD calculation) is used for comparing simirality between two blocks. Amount of block shifting is 16 pixel.

The OSCAR compiler parallelizes the loop for Y-direction in searching block. In addition, FE-GA compiler developed by Hitachi analyzed that SAD calculation, which occupies a large part of the program execution time, is to be executed

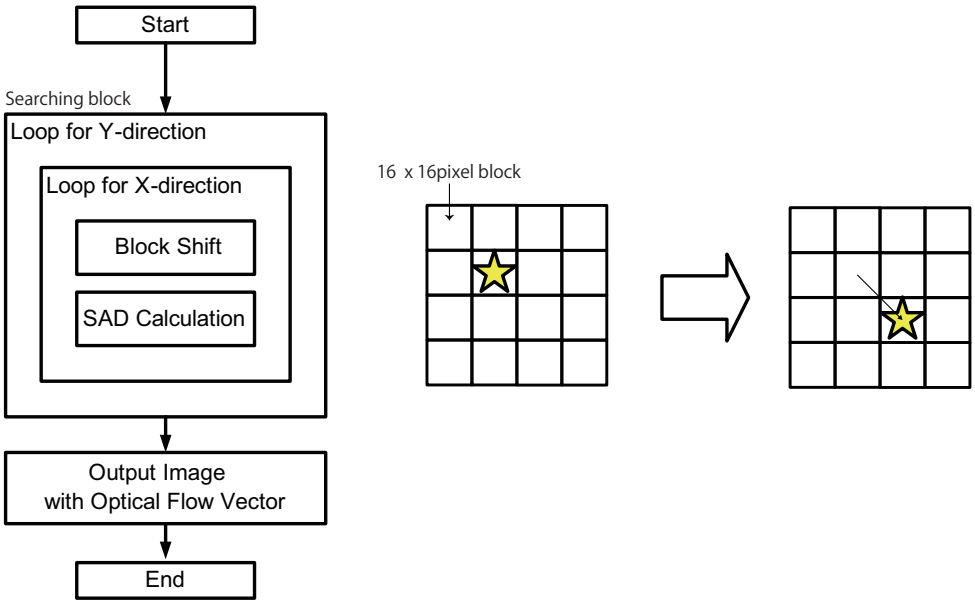


Figure 4.3: Program Structure of the Optical Flow(OpenCV)

on FE-GA. As described in Chapter 3, FE-GA compiler automatically inserts the hint directives to the C program. The OSCAR compiler generates parallel C program with OSCAR API. The parallel program is translated into parallel executable binary by using API analyzer which translates the directives to library calls and sequential compiler and FE-GA compiler translates the program parts in the accelerator files to FE-GA binary. Input images are two 320x352 bitmap images. Data transfer between SH-4A and FE-GA is performed by SH-4A via data cache.

4.3.3 Optical flow(Hand-tuned)

Another Optical flow program is developed by Hitachi Ltd and Tohoku University. As described in the previous subsection, the algorithm is a type of an object tracking system. The difference between OpenCV version and this version

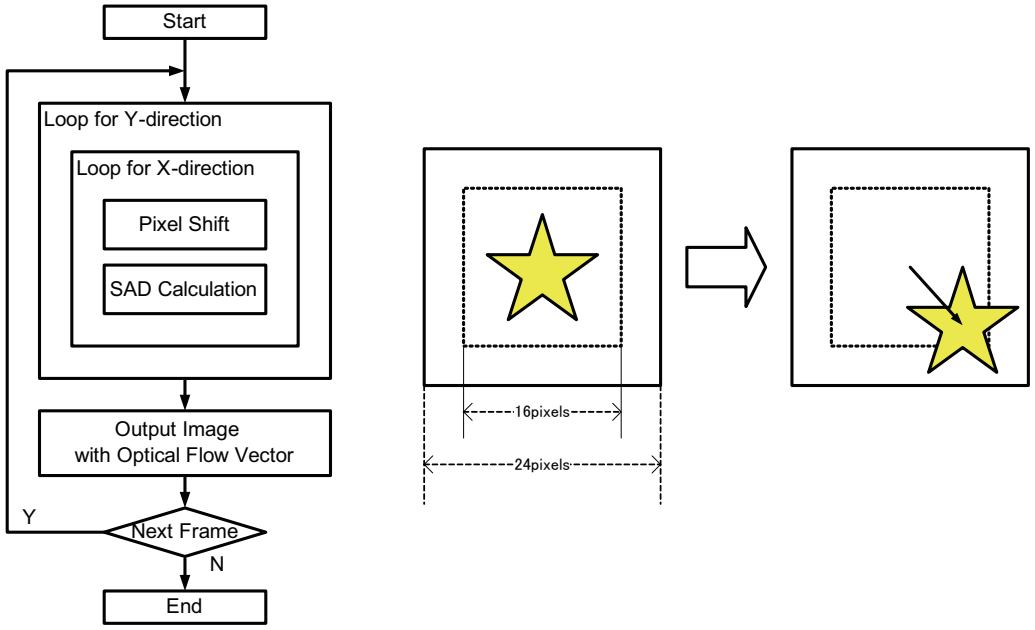


Figure 4.4: Program Structure of the Optical Flow(Hand-tuned)

is (1) the input is sequence of images, (2) amount of block shifting is 1 pixel. The program is modified in Parallelizable C[MOKK10].

Fig.4.4 shows the program structure of the optical flow. As shown in Fig.4.4, for a sequence of images the algorithm calculates the optical flow. The block size is 16x16 pixels and sum of absolute difference calculation(SAD calculation) is also used for comparing simirality between two blocks.

The OSCAR compiler parallelizes the same loop, which is shown in the previous subsection. The hand-tuned library, which executes 81 SAD functions in parallel, is used for FE-GA. The hint directives are inserted to the parallelizable C program by the programmer. The OSCAR compiler generates parallel C program with OSCAR API or directives for these library function calls. The directives in the parallel program is translated to library calls by using API analyzer. Then, sequential compiler generates the executables linking with hand-tuned library for SAD. Input

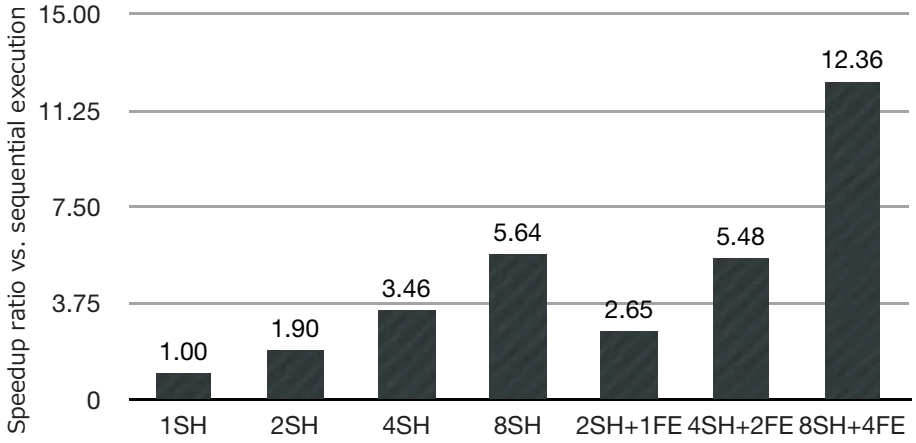


Figure 4.5: Performance by The OSCAR compiler and FE-GA Compiler(Optical Flow)

image size, number of frames and block size is 352x240, 450, 16x16, respectively. Data transfer between SH-4A and FE-GA is performed by SH-4A via data cache.

4.4 Performance evaluation

4.4.1 Performance by The OSCAR compiler with accelerator compiler

Fig.4.5. shows parallel processing performance of the optical flow on RP-X. The horizontal axis shows the processor configurations. For example, 8SH+4FE represents for the configuration with eight SH-4A general-purpose cores and four FE-GA accelerator cores. The vertical axis shows the speedup against the sequential execution by a SH-4A core. As shown in Fig.4.5, the proposed compilation framework achieves speedups of up to 12.36x with 8SH+4FE.

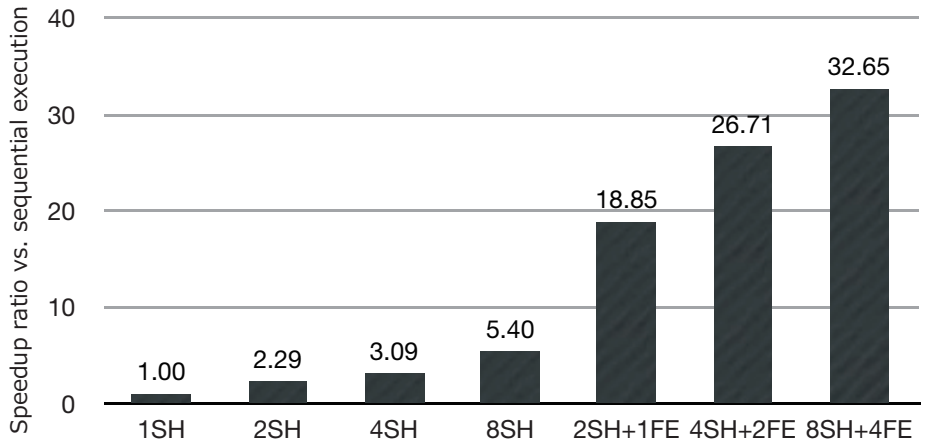


Figure 4.6: Performance by The OSCAR compiler and Hand-tuned Library(Optical Flow)

4.4.2 Performance by The OSCAR compiler with hand-tuned library

Fig.4.6 shows parallel processing performance of the optical flow at RP-X. The horizontal axis shows the processor configurations. For example, 8SH+4FE represents for the configuration with eight SH-4A general-purpose cores and four FE-GA accelerator cores. The vertical axis shows the speedup against the sequential execution by a SH-4A core. As shown in Fig.4.6, the proposed framework achieved speedups of up to 32.65x with 8SH+4FE.

Fig.4.7. shows parallel processing performance of the AAC at RP-X. As shown in Fig.4.7, the proposed framework achieved speedups of up to 16.08x with 8SH+4FE.

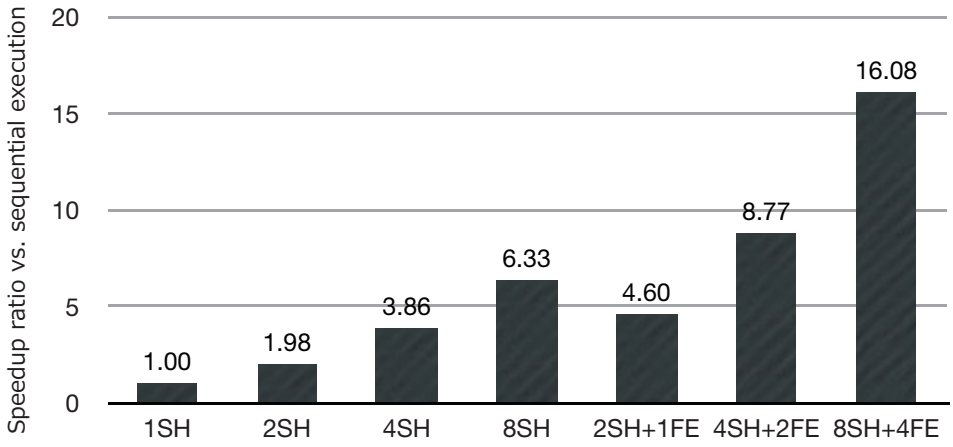


Figure 4.7: Performance by The OSCAR compiler and Hand-tuned Library(AAC)

4.5 Power consumption evaluation

This section evaluates a power consumption by using optical flow and AAC encoding for real-time execution on RP-X. Fig.4.8 shows the power reduction by The OSCAR compiler's power control, under the condition satisfying the deadline. The deadline of the optical flow is set to 33ms for each frame processing so that standard 30 [frames/sec] for moving picture processing can be achieved. The minimum number of cores required for the deadline satisfaction of optical flow calculation is 2SH+1FE. As shown in Fig.4.8, OSCAR heterogeneous multicore compiler reduces from 65% to 75% of power consumption for each processor configuration. Although power consumption is increased by the augmentation of processor core, the proposed framework reduces the power consumption.

Fig.4.9 shows the waveforms of power consumption in the case of optical flow using 8SH+4FE. The horizontal axis and the vertical axis show elapsed time and a power consumption, respectively. In the Fig.4.9, the arrow shows a processing

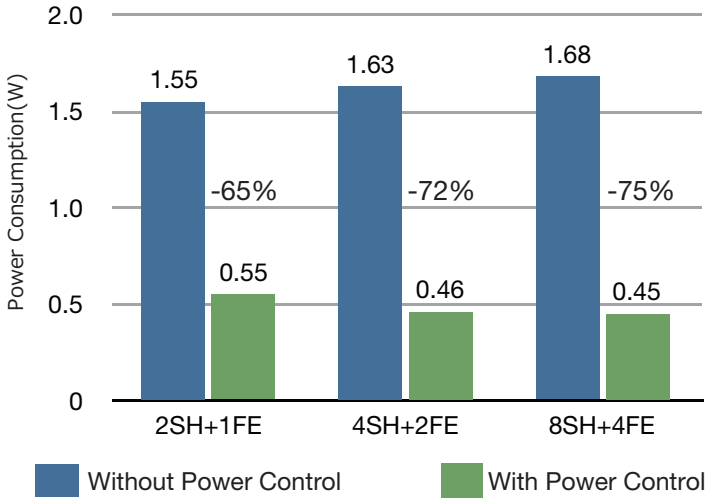
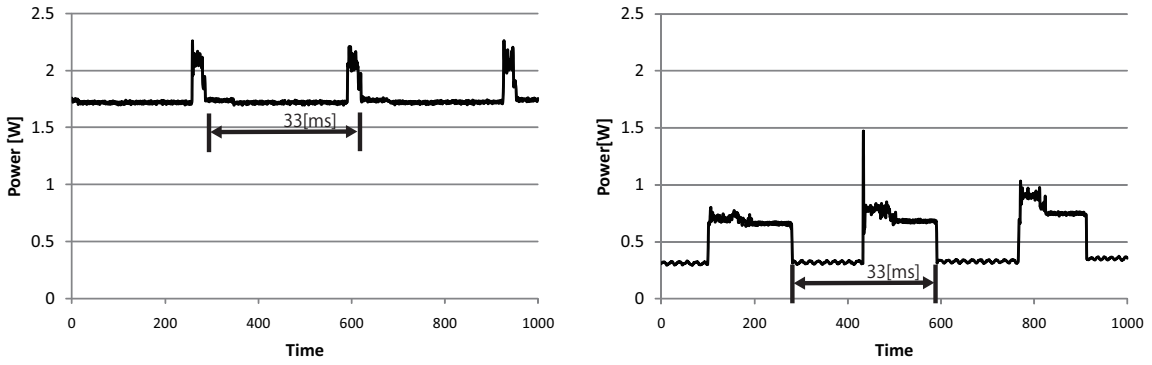


Figure 4.8: Power reduction by The OSCAR compiler's power control (Optical Flow)

period for one frame, or 33ms. In the case of applying power control(shown in Fig.4.9. b), each core executes the calculation by changing the frequency and the voltage on a chip. As a result, the consumed power ranges 0.3 to 0.7[W] by the OSCAR compiler's power control. On the contrary, in the case of applying no power control(shown in Fig.4.9. a), the consumed power ranges 2.25[W] to 1.75[W].

Fig.4.10 shows the summary of frequency and voltage status for optical flow calculation with 8SH+4FE. In this figure, FULL is 648MHz with 1.3V, MID is 324MHz with 1.1V, and LOW is 162MHz with 1.0V. Each box labeled "MID" and "timer" "Sleep" represents macro-task. As shown in Fig.4.10., four SAD tasks are assigned to each FE-GA, and the tasks are executed at MID. All SH-4A core except "CPU0" is shutdown until the deadline comes. "CPU0" executes "timer" task for satisfying the deadline. In other words, "CPU0" boot up other SH-4A



a) Without Power Saving(Average:1.68W) b) With Power Saving(Average:0.45W)

Figure 4.9: Waveforms of Power Consumption(Optical Flow)

cores when the program execution reaches the deadline. Note that FE-GA core is not shutdown after task execution because DVFS is only applicable.

For AAC program, an audio stream is processed per frame. The deadline of AAC is set to encode 1 [sec] audio data within 1 [sec]. Fig.4.11 shows the waveforms of power consumption in the case of AAC using 8SH+4FE. In the case of applying power control(shown in Fig.4.11. b)), each core executes the calculation by changing the frequency and the voltage on a chip. As a result, the consumed power ranges 0.4 to 0.55[W]. On the contrary, in the case of applying no power control(shown in Fig.4.11. a), the consumed power ranges 1.9[W] to 3.1[W]. In summary, the proposed framework realizes the automatically power reduction of heterogeneous multicore for several applications.

4.6 Conclusion

This chapter has described parallel processing schemes for the motion-tracking algorithm called optical flow and the audio encoder called advanced audio codec

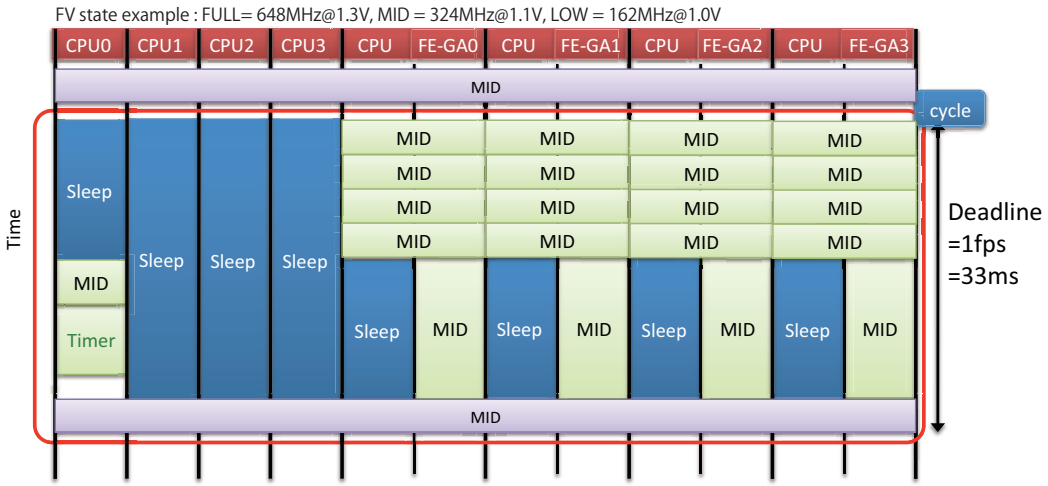
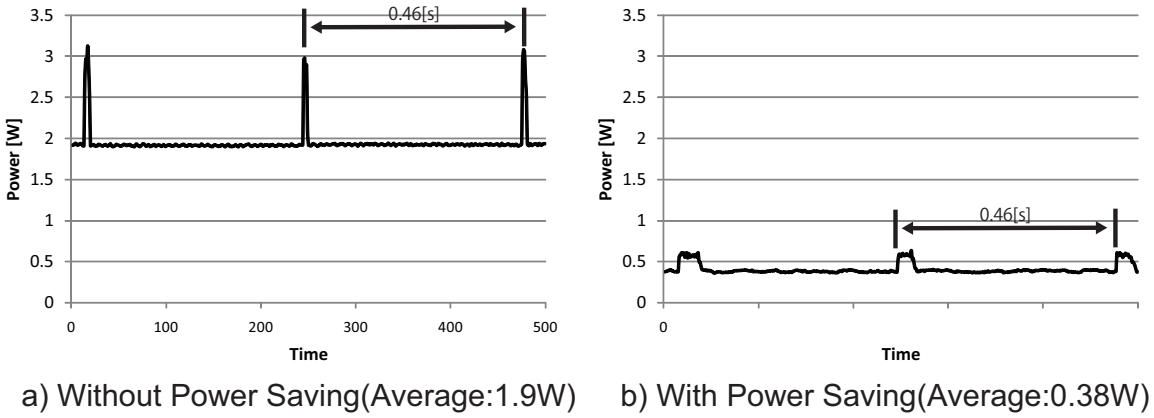


Figure 4.10: Power Control for 8SH+4FE(Optical Flow)

(AAC) encoder. This chapter also has evaluated the processing performance and the power reduction by the proposed framework on the RP-X heterogeneous multicore, which integrates eight general-purpose processor cores, or SH4A, and three kinds of accelerators including dynamically reconfigurable processor (DRP) accelerators, or FE-GA. The RP-X is developed by Renesas, Hitachi, Tokyo Institute of Technology and Waseda University . As a result, the framework attains speedups of 32.6 times with eight SH-4A cores and four FE-GA cores, 18.8 times with two SH-4A cores and one FE-GA core, 5.4 times with eight SH-4A cores against sequential execution by a single SH4A core and 70% of power reduction for the optical flow on the RP-X. The proposed framework has realized an automatic parallelization for various processor configurations.



a) Without Power Saving(Average:1.9W) b) With Power Saving(Average:0.38W)

Figure 4.11: Waveforms of Power Consumption(AAC)

Chapter 5

A Parallel Processing Scheme for Dose Calculation on SMP Servers

5.1 Introduction

Cancer is the primary cause of a person's death in Japan. One in three people died of cancer according to an investigation by the Ministry of Health, Labor and Welfare. Recently, there has been a growing interest in heavy particle radiotherapy due to its excellent recovery rate. This is because heavy particle radiotherapy is capable of giving much more damage to abnormal cells and less damage to normal cells than conventional X-ray radiotherapy. Many hospitals, from the National Institute of Radiological Sciences (NIRS) perform this treatment in Japan. Because the affected area depends on a patient, a doctor must perform dose calculation, which calculates optimal angle of beam and quantity of irradiance, by using computer in advance. This calculation performs 3-D physical simulations, which is called "Dose calculation engine". It is necessary to repeat the dose calculation in order to maximize the effect of the therapy during the planning process. However, the dose calculation is time consuming, which means that only a small number of patients is treated.

From a computer science points of view, multicore processors have attracted much attention over years in order to attain high performance with low power consumption. Multicore processors must be able to execute the dose calculation in shorter time by parallelization. However, a parallel programming is greatly difficult because a programmer have to decompose a program into tasks, and schedule these tasks onto processor elements by inserting synchronization and data transfer codes. In previous work, manual parallelization only achieves 2.8 times speedup by using eight general-purpose processors[YA10]. It is essential to achieve good speedups with the increase of general-purpose processors.

As described in Chapter 3, the OSCAR compiler is able to parallelize a Fortran

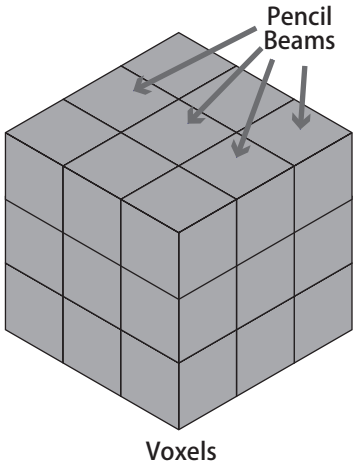


Figure 5.1: Dose Calculation using pencil beam algorithm

77 or Parallelizable C, a kind of programming style of C language, and automatically generates a parallelized program for various kinds of multicore including homogeneous and heterogeneous multcores.

This chapter proposes a parallel processing scheme of the dose calculation, and evaluates a performance of automatically parallelized dose calculation program on SMP servers by using OSCAR API. The rest of the chapter is organized as follows: Section 5.2 describes an overview of the dose calculation. Section 5.3 proposes a parallel processing scheme of the dose calculation. Section 5.4 evaluates a performance on several SMP servers. Section 5.5 conclude this chapter.

5.2 Overview of dose calculation

This section describes an overview of the dose calculation engine and profile result of the dose calculation.

The calculation engine is developed by National Institute of Radiological Sci-

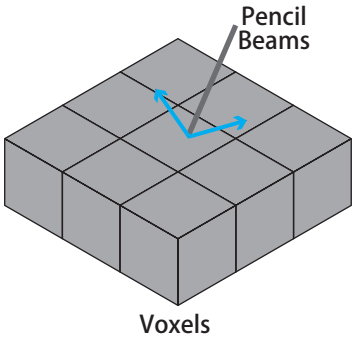


Figure 5.2: Scatter Calculation

ences (NIRS) and Mitsubishi Electric Corporation[YAT+10, TAS+10]. The dose calculation engine realizes accurate dose calculation using an algorithm which simulates a natural phenomenon such as scatter in addition to a dose calculation algorithm called pencil beam methodology.

The calculation engine calculates quantity of dose for each 3-D voxel. The input are 3-D computerizing tomograph(CT) images and an outline of the cancer affected area. The output is quantity of dose for each voxel. Although an original program is written C++ language, the calculation engine is written in Parallelicable C.

The calculation engine consists of initialization, dose calculation, scatter calculation and modify calculation. The algorithms are the following:

step 1: An initialization of data

Allocate arrays which correspond to voxels. Each element of arrays is initialized to zero.

step 2: Dose Calculation using Pencil Beam Methodology

Fig.5.1 shows an image of the dose calculation. Particle beams which irradiates to a patient are expressed in the composition of small pencil beam. In

```
for (the number of pencil beams)
    for (the number of passed voxels) {
      // Dose Calculation
      dose = dcalc();
      Voxel1[xyz] += dose;
    }
```

Figure 5.3: The Dose Calculation

Fig.5.1, “Voxels” and “Pencil Beams” are 3-D voxel arrays and pencil beams respectively.

The pseudo code in Fig.5.3 shows that for each voxel and pencil beam the value calculated by function called“dcalc” is added to array called “Voxel1”. In this code, “Voxel1” represents the voxel array, “xyz” indicates one voxel in the voxel array.

step 3: Scatter Calculation Fig.5.2 shows an image of scatter calculation. quantity of scatter values which influences on neighbor voxels are added to voxel arrays calculated in the previous step.

Scatter phenomenon influences on all axes and the scattered value depends on a quantity of dose in a voxel.

Fig.5.2 shows a pencil beam influences upon neighbor cell.

The pseudo code in Fig.5.4 shows that for each voxel the scattered value calculated by function called “scalc” is added to array called “Voxel2”. In this code, “xyz” indicates one voxel in the voxel array.

step 4: Modify Calculation

```
for (Z-Axis)
  for (Y-Axis)
    for (X-Axis) {
      scatter = scalc(Voxel1);
      for (neighbor voxels) {
        // scatter is added to Voxel2
        Voxel2[xyz] += scatter;
      }
    }
```

Figure 5.4: The Scatter Calculation

Modify calculation performs modification of dose values.

Fig.5.5 shows a profile result of the calculation engine on Intel Xeon processor and IBM Power 7 processor. The parameter of these processors are described in Section 5.4.

As shown in Fig.5.5, the dose calculation occupies 90% time of a whole execution time on both processors. The scatter calculation is second in the execution time. "The others" in Fig.5.5 means percentage of initialization and modification part.

5.3 Enhancing parallelism

This section proposes a parallel processing scheme of the dose calculation and apply a tuning in order to improve a parallelism of the dose calculation.

The dose calculation is consists of an initialization, dose calculation, scatter calculation and modify calculation as described in the previous section. This

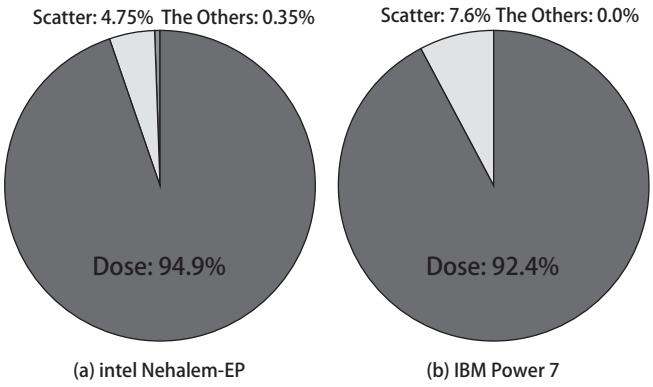


Figure 5.5: Profile results on Intel/IBM processor

section explores a possibility of enhancing parallelism in terms of a code rewriting because the original code can limit the potential parallelism without any change.

Each of the following sections explains the code rewriting details for each step.

5.3.1 Dose calculation

In the dose calculation, it is essential to exploit the beam level parallelism because the number of beams is a large number.

However, because transit area of each beam interferes with each other and the subscript of the accessed array is not available at compile-time due to an indirect array access, exploiting the parallelism of the dose calculation is difficult. To expands the dimension of voxel arrays can enhance the parallelism of the loop. In other words, each processor calculates the dose value from a part of given pencil beams in parallel. This rewriting causes a side effect, which means that each array calculated in parallel has to be accumulated to one array. The code which accumulates each array is moved to the scatter calculation. The pseudo code in Fig.5.6 shows that parallelizable version of the dose calculation.

```
/* parallelizable loop */
for (p = 0; p < nCPUs; p++)
  for (the number of pencil beams/nCPUs)
    for (the number of passed voxels) {
      // Dose calculation
      dose = dcalc();
      Voxel1[xyz][p] += dose;
    }
```

Figure 5.6: The Parallelizable Dose Calculation

In this code , "nCPUs" means the number of processors, "p" means the ID of each processor. This rewriting allow the compiler to exploit the loop level parallelism in the outer most loop because transit area of pencil beams calculated by each processor does not interfere.

5.3.2 Scatter calculation

Scatter calculation calculates an influence on neighbor voxels depending on amount of the dose value.

However, the subscript of the accessed array is not available at compile-time because an influenced voxel cannot be identified statically. Exploiting the parallelism of the scatter calculation is difficult.

To expands the dimension of voxel arrays also can enhance the parallelism of the loop as described in the previous subsection. The pseudo code Fig.5.7 shows that parallelizable version of the scatter calculation.

```
 /* parallelizable loop */
for (p = 0; p < nCPUs; p++)
  for (Z-Axis/nCPUs)
    for (Y-Axis)
      for (X-Axis) {
        for (q = 0; q < nCPUs; q++) {
          Voxel1_1[xyz] += Volel1[xyz][q];
        }
        scatter = scalc(Voxel1_1);
        for (nrighbor voxels) {
          // scatter is added to Voxel2
          Voxel2[xyz] += scatter;
        }
      }
```

Figure 5.7: The Parallelizable Scatter Calculation

In this code, "q" indicates the 1-D array for each processor from the expanded 2-D array, "Voxel1_1" means the array to which each 1-D is accumulated, and "Voxel2" means represent the voxel array.

This rewriting allow the compiler to exploit the loop level parallelism in the outer most Z-Axis loop because the scatter influence calculated by each processor does not interfere. In other words, each processor calculates the scatter value to neighbor voxels in parallel.

This rewriting also causes a side effect, which means that each array calculated in parallel has to be accumulated to one array. The code in Fig.5.8 accumulates

```
/* parallelizable loop */
for (p = 0; p < nCPUs; p++) {
  for (all voxel/nCPUs) {
   Voxel3[xyz] += Voxel2[p][xyz];
  }
}
```

Figure 5.8: The Accumulation Calculation

each array.

5.3.3 Initialization

In the initialization, arrays which corresponds to voxels is allocated by using "malloc" function and are initialized to zero by using for-loop. "malloc" function is changed to static allocation in order to avoid an overhead of "malloc" function including heap-locking cost and initialization cost as described in Section 5.3.5.

5.3.4 Modification

The modification calculation modifies the value depending on the result of the scatter calculation.

However, this part is not parallelized because the percentage of the modification part is less than 0.15% in both Intel and IBM processor as shown in Fig.5.5.

A parallelization of this part is needed when 512 or more CPUs are used theoretically. In detail, if using 512 CPUs achieves 512 times speedups of the dose and the scatter calculation ideally the accelerated cost equals to the cost of modifica-

tion calculation. Therefore the modification calculation is not parallelized in this thesis.

5.3.5 Code rewriting for enhancing scalability

It is essential to parallelize program and to enhance its parallelism in the whole program in order to exploit the capability of SMP server. Although small part of the program limits the parallelism when many CPUs are utilized, parallelizing the whole program is not always leads to speedups. Parallelizing for-loop which allocates 2-D array by using "malloc" function must be avoided because "malloc" function internally locks heap-area and it leads to performance degradation. That is why "malloc" function is changed to static allocation.

5.4 Performance evaluation on SMP servers

This section evaluates the performance of the dose calculation engine, the parallelism of which is enhanced, automatically parallelized by the OSCAR compiler, on SMP servers and analyze the results.

5.4.1 Evaluation environment

This section evaluates the dose calculation engine on two SMP servers: these are SR16000, which integrates IBM Power 7 processors and HA8000/RS220, which integrates Intel Xeon processors. Table 5.1 shows the configuration of each server.

Parallelized program is capable of executing on the SMP servers by native compilers for each server because the OSCAR compiler works as a source-to-source

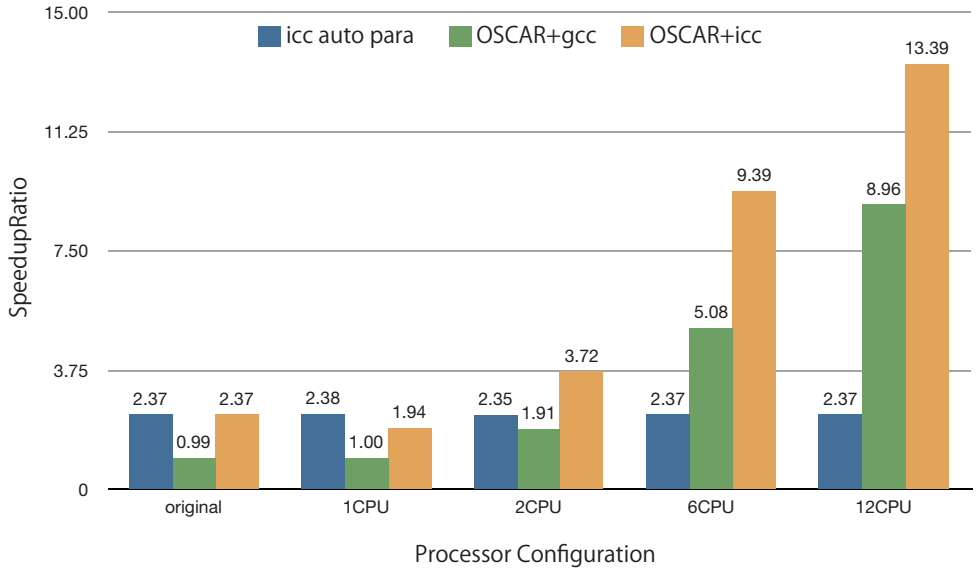


Figure 5.9: Evaluation Result on Intel Xeon Processor

compiler and generates OSCAR API, which extends the subset of OpenMP[Opec] as described in Section 3.5.

As shown in Table 5.1, GNU GCC compiler, IBM XLC compiler, ICC compiler are used for generating the executables. In addition, this section also evaluates the performance of an automatic parallelization by XLC and ICC because XLC and ICC are able to perform an automatic parallelizaton.

Note that a compiler option which means the compiler generates 32-bit binary is used for Intel processor because there is not the 64-bit shared library which is linked with the dose calculation engine.

5.4.2 Performance on HA8000/RS220(Intel Xeon)

Fig.5.9 shows the parallel processing performance of the dose calculation engine at HA80000/RS220. The horizontal axis shows the processor configurations, "original" means a sequential performance of the original code by using ICC and GCC, "nCPU" means a sequential and parallel performance of the parallelism-enhanced code as described in Section 5.3 by using the OSCAR compiler for a parallelization and by using ICC and GCC for generating the executables. Note that the performance of an automatic parallelization by ICC shows the performance of the original code because the performance is better than enhanced code. The vertical axis shows the speedup against the sequential execution by 1CPU with GCC.

As shown in Fig.5.9, the performance by an automatic parallelization by ICC does not show the scalability such as 2.37x with 12CPUs. On the other hand, the performance by the OSCAR compiler shows the good scalability such as 6.89x with 12CPUs and ICC, 8.96x with 12CPUs and GCC.

Fig 5.10 shows an analysis for the execution time of the dose calculation engine. The horizontal axis shows the processor configurations, "gcc" and "icc" is a native compiler which generates the executables. The vertical axis shows a relative execution time which is normalized to the execution time of the original code compiled by gcc. As described in Section 5.2, the dose calculation and the scatter calculation occupies more than 96% time of a whole execution.

For GCC compiler, as the number of the CPUs increase from 1CPU to 12CPUs, the execution time of the dose calculation decreases from 93.70 to 9.05 and the execution time of the scatter calculation decreases from 5.25 to 1.9.

For ICC compiler, as the number of the CPUs increase from 1CPU to 12CPUs, the execution time of the dose calculation decreases from 47.60 to 5.45 and the

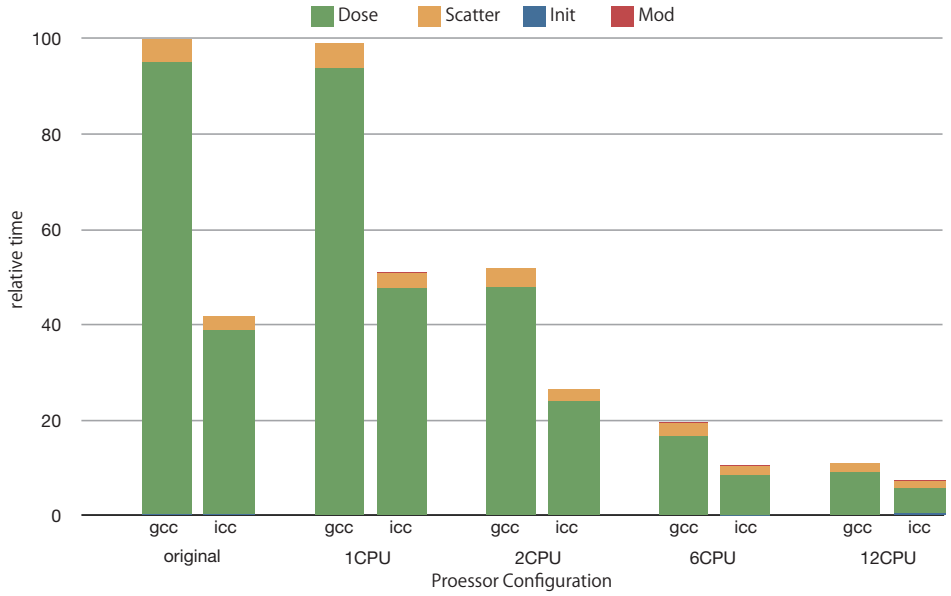


Figure 5.10: Performance Analysis on Intel Xeon Processor

execution time of the scatter calculation decreases from 3.35 to 1.45.

That is why the OSCAR compiler realize good scalability.

In terms of an automatic parallelization by ICC, ICC does not show the scalability as shown in Fig.5.9. According to a parallelization report by ICC, ICC does not parallelize important loops in the dose and scatter calculation due to loop carried dependences. Although some loops in the scatter and modification calculation are analyzed to be parallelized, ICC actually does not parallelize these loops because the number of the iterations is insufficient. As a result, ICC achieves speedups of 2.45x for the dose calculation and 1.60x for the scatter calculation by a automatic vectorization compared to sequential execution by GCC. In contrast, the OSCAR compiler parallelizes important loops in the dose and scatter calculation due to its rich analysis, pointer analysis and so on.

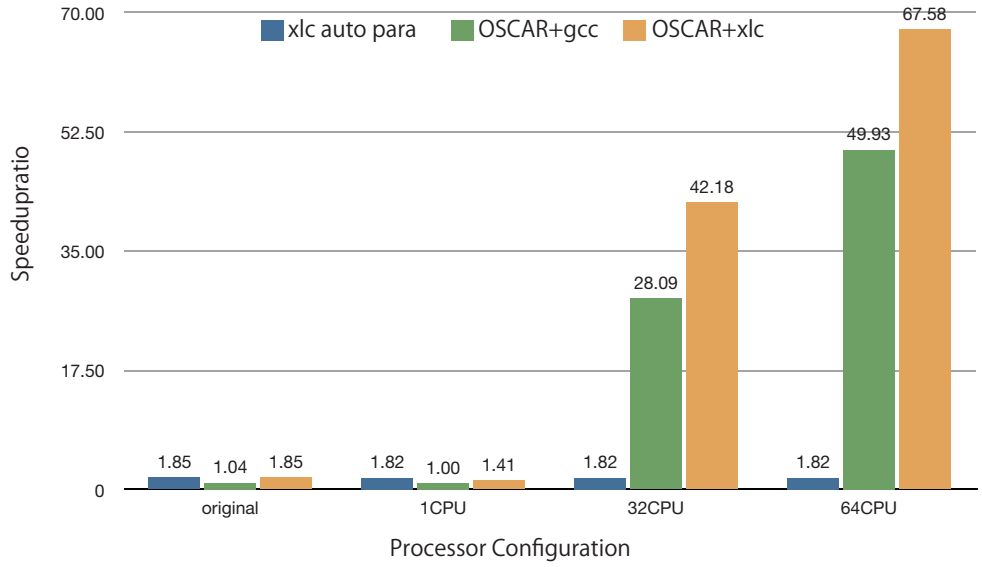


Figure 5.11: Evaluation Result on IBM Power7 Processor

5.4.3 Performance on Hitachi SR16000(IBM Power7)

Fig.5.11 shows the parallel processing performance of the dose calculation engine at SR16000. The horizontal axis shows the processor configurations, "original" means a sequential performance of the original code by using XLC and GCC, "nCPU" means a sequential and parallel performance of the parallelism-enhanced code as described in Section 5.3 by using the OSCAR compiler for a parallelization and by using XLC and GCC for generating the executables. Note that the performance of an automatic parallelization by XLC shows the performance of the original code because the performance is better than enhanced code. The vertical axis shows the speedup against the sequential execution by 1CPU with GCC.

As shown in Fig.5.11, the performance by an automatic parallelization by XLC does not show the scalability such as 1.82x with 64CPUs. On the other hand, the

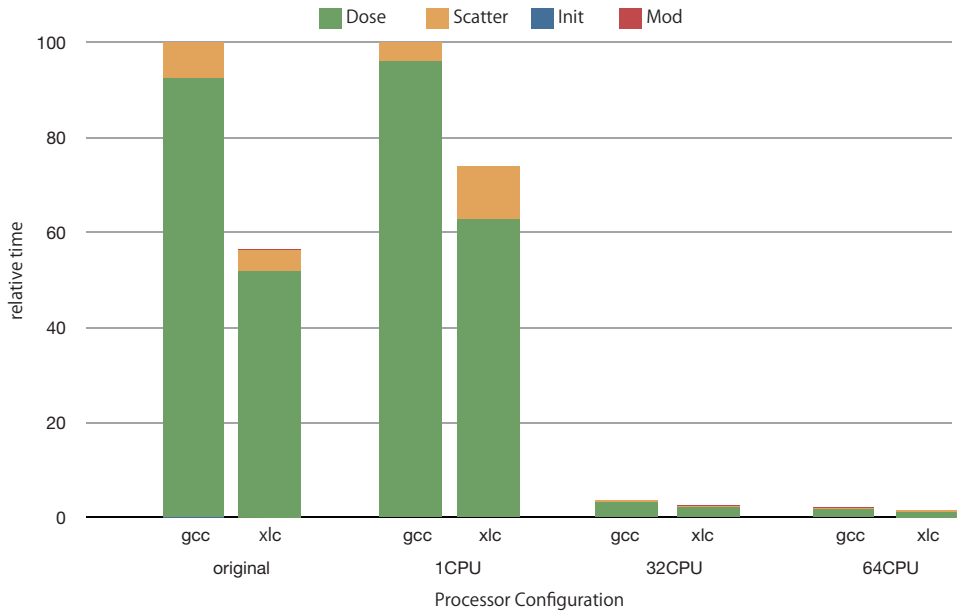


Figure 5.12: Performance Analysis on IBM Power7 Processor

performance by the OSCAR compiler shows the good scalability such as 48.06x with 64CPUs and XLC, 49.93x with 64CPUs and ICC.

Fig 5.12 shows an analysis for the execution time of the dose calculation engine. The horizontal axis shows the processor configurations, "gcc" and "xlc" is a native compiler which generates the executables. The vertical axis shows a relative execution time which is normalized to the execution time of the original code compiled by gcc. As described in Section 5.2, the dose calculation and the scatter calculation occupies more than 96% time of a whole execution.

For GCC compiler, as the number of the CPUs increase from 1CPU to 64CPUs, the execution time of the dose calculation decreases from 95.90 to 1.69 and the execution time of the scatter calculation decreases from 7.60 to 0.35.

For XLC compiler, as the number of the CPUs increase from 1CPU to 64CPUs, the execution time of the dose calculation decreases from 62.60 to 1.2 and the

execution time of the scatter calculation decreases from 11.00 to 0.29.

That is why the OSCAR compiler realize good scalability.

In terms of an automatic parallelization by XLC, XLC does not show the scalability as shown in Fig.5.9. According to a parallelization report by XLC, XLC does not parallelize important loops in the dose and scatter calculation due to loop carried dependences. Some loops in the initialization, dose, scatter and modification calculation are analyzed to be parallelized, As a result, XLC achieves speedups of 1.30x for the initialize calculation, 1.77x for the dose calculation and 1.57x for the scatter calculation compared to sequenatial execution by GCC. In contrast, the OSCAR compiler parallelizes important loops in the dose and scatter calculation due to its rich analysis, pointer analysis and so on.

5.5 Conclusion

This chapter has described a parallel processing scheme of dose calculation for heavy particle radiotherapy for cancer treatment and evaluates processing performance by the proposed framework on SMP servers. This dose calculation engine is based on the clinically used program developed by National Institute of Radiological Sciences (NIRS) and Mitsubishi Electronics. This calculation performs 3-D physical simuations, which is called “Dose calculation engine”. It is necessary to repeat the dose calculation in order to maximize the effect of the therapy during the planning process. In order to reduce the simulation time by parallelization, this thesis has proposed a processing scheme for the application and enables the OSCAR compiler to exploit the parallelism of the calculation engine. As a result, the proposed method attains good speedups of 9.0 times with 12 processor cores

on Hitachi HA8000/RS220 system based on the Intel Xeon Processor and 50.0 times with the 64 processor cores on Hitachi SR16000 system based on the IBM Power 7 processor.

Table 5.1: Evaluation environment

	SR16000	HA8000/RS220
CPU	IBM Power 7 ((4.00GHz × 8) × 4) × 4	Intel Xeon X5670 (2.93GHz × 6) × 2
L1 D-Cache	32KB / 1 CPU	32KB / 1 CPU
L1 I-Cache	32KB / 1 CPU	32KB / 1 CPU
L2 cache	256KB / 1 CPU	256KB / 1 CPU
L3 cache	32MB / 8 CPU	12MB / 6 CPU
Operating System	Red Hat Enterprise Linux	Ubuntu Linux
Native Compiler1	GNU C Compiler version 4.4.5	GNU C Compiler version 4.4.3
Compile Option 1	-O3 -m32 -fopenmp	-O3 -m32 -fopenmp
Native Compiler2	IBM XLC Compiler 11.0	Intel Parallel Studio 12.0
Compile Option 2-1	-q64 -qsmp=omp -O4 -qarch=pwr7 -qmaxmem=-1	-m32 -fast -openmp
Compile Option 2-2	-q64 -qsmp=auto -O4 -qarch=pwr7 -qmaxmem=-1	-m32 -fast -parallel
Voxel Size	0.5mm	1.5mm
# of Beam	165018	18369

Chapter 6

Conclusions

6.1 Summary of results

This thesis has proposed the automatic parallelizing compiler framework for various heterogeneous multicores.

The input of the framework is a sequential program written in Parallelizable C, a kind of C programming style for parallelizing compiler, or Fortran77 and the output is an executable for a target heterogeneous and homogeneous multicore. The compilation flow consists of the following 4 steps.

Step 1: Accelerator compilers or programmers insert hint directives immediately before loops or function calls , which can be executed on the accelerator, in a sequential program.

Step 2: the OSCAR compiler parallelizes the source program considering with hint directives: the compiler schedules coarse-grain tasks[WHM+08] to processor or accelerator cores and apply the low power control[KMM+09]. Then, the compiler generates a parallelized C or Fortran program for general-purpose processors and accelerator cores by using OSCAR API. At that time, the compiler generates C source codes as separate files for accelerator cores. Each file includes functions to be executed on accelerators when a function is scheduled onto accelerator by the compiler.

Step 3: Each accelerator compiler generates objects for its own target accelerator. Note that each accelerator compiler also generates both data transfer code between controller and accelerator, and accelerator invocation code.

Step 4: An API analyzer prepared for each heterogeneous multicore translates OSCAR APIs into runtime library calls, such as pthread library. After-

wards, an ordinary sequential compiler for each processor from each vender generates an executable.

This thesis has evaluated the performance and the power consumption

Results of this thesis are summarized as follows.

6.1.1 Media applications on RP-X processor

This thesis has evaluated the performance of the proposed framework on 15 core heterogeneous multicore RP-X[YIK+10] using media applications.

The framework attains speedups of 32.6 times with eight SH-4A cores and four FE-GA cores, 18.8 times with two SH-4A cores and one FE-GA core, 5.4 times with eight SH-4A cores against sequential execution by a single SH4A core and 70% of power reduction for the optical flow on the RP-X.

6.1.2 Dose calculation engine on SMP servers

This thesis has proposed a parallel processing scheme of dose calculation for heavy particle radiotherapy for cancer treatment and evaluates processing performance by the proposed framework on SMP servers. This dose calculation engine is based on the clinically used program developed by National Institute of Radiological Sciences (NIRS) and Mitsubishi Electronics. As a result, the proposed method attains good speedups of 9.0 times with 12 processor cores on Hitachi HA8000/RS220 system based on the Intel Xeon Processor and 50.0 times with the 64 processor cores on Hitachi SR16000 system based on the IBM Power 7 processor.

6.2 Future works

This thesis has realized an automatic parallelization for heterogeneous multicores.

The future work is the following:

- Fully Automatic Parallelization of C program

  Fully automatic parallelization of C program by enhancing analyzer such as pointer analyzer is important because real applications usually use structures/unions.

- Power capping control

  Power capping control by a compiler is required for battery powered mobile devices including solar-powered devices

- Automatic detection of acceleration part

  Detecting which parts of the program can be executed on specified accelerator automatically is important because many frameworks leave it to programmers.

- Compiler-Directed Local Memory Management for Heterogeneous Multicore

  Utilizing local memory by compiler is necessary for real-time heterogeneous multicores.

Bibliography

[ATNW11] 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*, Vol. 23, pp. 187–198, February 2011.

[BPBL09] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. Cellss: a programming model for the cell be architecture. *In Proceedings of the 2006 ACM/IEEE Conference on Supercomputing(SC'06)*, pp. 5–5, 2009.

[BPT+11] S. Benkner, S. Pllana, J.L. Traff, P. Tsigas, U. Dolinsky, C. Augonnet, B. Bachmayer, C. Kessler, D. Moloney, and V. Osipov. Peppher: Efficient and productive usage of hybrid computing systems. *Micro, IEEE*, Vol. 31, No. 5, pp. 28 –41, sept.-oct. 2011.

[Cor] Intel Corporation. Open source computer vision library reference manual. http://itee.uq.edu.au/ iris/CVsource/OpenCVreferencemanual.pdf.

Bibliography

[DBB07] R. Dolbeau, S. Bihan, and F. Bodin. Hmpp(tm):a hybrid multi-core parallel programmingg environment. In *GPGPU '07: Proceedings of the 1st Workshop on General Purpose Processing on Graphics Processing Units*, 2007.

[EHP98] R. Eigenmann, J. Hoeflinger, and D. Padua. On the automatic parallelization of the perfect benchmarks(r). *Parallel and Distributed Systems, IEEE Transactions on*, Vol. 9, No. 1, pp. 5 –23, jan 1998.

[FPB+11] Roger Ferrer, Judit Planas, Pieter Bellens, Alejandro Duran, Marc Gonzalez, Xavier Martorell, Rosa M. Badia, Eduard Ayguade, and Jesus Labarta. Optimizing the exploitation of multicore processors and gpus with openmp and opencl. In *Proceedings of the 23rd international conference on Languages and compilers for parallel computing*, LCPC'10, pp. 215–229, Berlin, Heidelberg, 2011. Springer-Verlag.

[GGN+08] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov. Parallel computing experiences with cuda. *IEEE Micro*, Vol. 28, No. 4, pp. 13–27, 2008.

[GMO+00] Marc Gonzalez, Xavier Martorell, Jose Oliver, Eduard Ayguade, and Jesus Labarta. Code generation and run-time support for multi-level parallelism exploitation. In *In Proc. of the 8th International Workshop on Compilers for Parallel Computing*, 2000.

[HAA+96] M.W. Hall, J.M. Anderson, S.P. Amarasinghe, B.R. Murphy, Shih-Wei Liao, and E. Bu. Maximizing multiprocessor performance with the suif compiler. *Computer*, Vol. 29, No. 12, pp. 84 –89, dec 1996.

[HDH+10] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pp. 108 –109, feb. 2010.

[HIK90] H. Honda, M. Iwata, and H. Kasahara. Coarse grain parallelism detection scheme of a fortran program. *Trans. of IEICE*, Vol. J73-D-1, No. 12, pp. 951–960, Dec. 1990.

[HWW+11] Akihiro Hayashi, Yasutaka Wada, Takeshi Watanabe, Takeshi Sekiguchi, Masayoshi Mase, Jun Shirako, Keiji Kimura, and Hironori Kasahara. Parallelizing compiler framework and api for power reduction and software productivity of real-time heterogeneous multicores. In *Proceedings of the 23rd international conference on Languages and compilers for parallel computing*, LCPC'10, pp. 184–198, Berlin, Heidelberg, 2011. Springer-Verlag.

[IHY+08] M. Ito, T. Hattori, Y. Yoshida, K. Hayase, T. Hayashi, O. Nishii, Y. Yasu, A. Hasegawa, M. Takada, H. Mizuno, K. Uchiyama, T. Odaka, J. Shirako, M. Mase, K. Kimura, and H. Kasahara. An 8640 mips soc with independent power-off control of 8 cpus and 8 rams by an automatic parallelizing compiler. In *Solid-State Circuits*

Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International, pp. 90 –598, feb. 2008.

[kas] kasahara.cs.waseda.ac.jp. Oscar-api v1.0. http://www.kasahara.cs.waseda.ac.jp/.

[Kas03] Hironori Kasahara. Advanced automatic parallelizing compiler technology. *IPSJ MAGAZINE*, Apr 2003.

[KGM+07] J. H. Kelm, I. Gelado, M. J. Murphy, N. Navarro, S. Lumetta, and W. M Hwu. Cigar: Application partitioning for a cpu/coprocessor architecture. In *16th International Conference on Parallel Architecture and Compilation Techniques, PACT 2007*, pp. 317–326, 15 September 2007 through 19 September 2007 2007.

[KHIH90] H. Kasahara, H. Honda, M. Iwata, and M. Hirota. A compilation scheme for macro-dataflow computation on hierarchical multiprocessor system. *Proc. Int Conf. on Parallel Processing*, 1990.

[KHM+91] H. Kasahara, H. Honda, A. Mogi, A. Ogura, K. Fujiwara, and S. Narita. A multi-grain parallelizing compilation scheme for OSCAR (Optimally scheduled advanced multiprocessor). In *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*, pp. 283–297, August 1991.

[KHN90] H. Kasahara, H. Honda, and S. Narita. Parallel processing of near fine grain tasks using static scheduling on OSCAR. *Proceedings of Supercomputing '90*, Nov. 1990.

[khr] khronos.org. Opencl. http://www.khronos.org/opencl/.

[KJJ+09] John H. Kelm, Daniel R. Johnson, Matthew R. Johnson, Neal C. Crago, William Tuohy, Aqeel Mahesri, Steven S. Lumetta, Matthew I. Frank, and Sanjay J. Patel. Rigel: an architecture and scalable programming interface for a 1000-core accelerator. *SIGARCH Comput. Archit. News*, Vol. 37, pp. 140–151, June 2009.

[KMM+09] K. Kimura, M. Mase, H. Mikami, T. Miyamoto, and J. Shirako H. Kasahara. Oscar api for real-time low-power multicores nad its performance on multicores and smp servers. *Proc of The 22nd International Workship on Languages and Compilers for Parallel Computing(LCPC2009)*, 2009.

[KOI00] H. Kasahara, M. Obata, and K. Ishizaka. Automatic coarse grain task parallel processing on smp using openmp. *Proc of The 13th International Workship on Languages and Compilers for Parallel Computing(LCPC2000)*, 2000.

[KP03] C.E. Kozyrakis and D.A. Patterson. Scalable, vector processors for embedded systems. *Micro, IEEE*, Vol. 23, No. 6, pp. 36 – 45, nov.-dec. 2003.

[KSSF10] R. Kalla, B. Sinharoy, W.J. Starke, and M. Floyd. Power7: Ibm's next-generation server processor. *Micro, IEEE*, Vol. 30, No. 2, pp. 7 –15, march-april 2010.

[KTR+04] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings -31st Annual Inter-*

national Symposium on Computer Architecture, pp. 64–75, 19 June 2004 through 23 June 2004 2004.

[KTT+06] T. Kodama, T. Tsunoda, M. Takada, H. Tanaka, Y. Akita, M. Sato, and M. Ito. Flexible engine: A dynamic reconfigurable accelerator with high performance and low power consumption. In *Proc. of 9th IEEE Symposium on Low-Power and High-Speed Chips(COOL Chips IX)*, Apr. 2006.

[LCL99] Amy W. Lim, Gerald I. Cheong, and Monica S. Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *Proceedings of the 13th international conference on Supercomputing*, ICS '99, pp. 228–237, New York, NY, USA, 1999. ACM.

[LHG+06] D. Luebke, M. Harris, N. Govindaraju, A. Lefohn, M. Houston, J. Owens, M. Segal, M. Papakipos, and I. Buck. Gpgpu: General-purpose computation on graphics hardware. In *2006 ACM/IEEE Conference on Supercomputing, SC'06*, 11 November 2006 through 17 November 2006 2006.

[LHK09] C. Luk, S. Hong, and H. Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping, microarchitecture. *2009. MICRO-42. Proceedings. 42th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 45–55, 2009.

[LLW+06] Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner. Soda: A low-power architecture for

software radio. In *33rd International Symposium on Computer Architecture, ISCA 2006*, Vol. 2006, pp. 89–100, 17 June 2006 through 21 June 2006 2006.

[MOKK10] M. Mase, Y. Onozaki, K. Kimura, and H. Kasahara. Parallelizable c and its performance on low power high performance multicore processors. In *Proc. of 15th Workshop on Compilers for Parallel Computing*, Jul. 2010.

[MTT+07] YOSHIDA Masayasu, SUGIHARA Takeshi, TAKAHASHI Toshiaki, KOUMOTO Yasuhiko, and ISHIHARA Toshinori. Naviengine 1, system lsi for smp-based car navigation systems. *NEC TECHNICAL JOURNAL*, Vol. 2, No. 4, 2007.

[MYK+10] T. Maruyama, T. Yoshida, R. Kan, I. Yamazaki, S. Yamamura, N. Takahashi, M. Hondou, and H. Okano. Sparc64 viiifx: A new-generation octocore processor for petascale computing. *Micro, IEEE*, Vol. 30, No. 2, pp. 30 –40, 2010.

[NMM+09] H. Nakano, T. Momozono, M. Mase, K. Kimura, and H. Kasahara. Local memory management scheme by a compiler on a multicore processor for coarse grain task parallel processing. *Trans. of IPSJ on Computing Systems(in Japanese)*, Vol. 2, No. 2, pp. 63–74, 2009.

[NTF+08] S. Nomura, F. Tachibana, T. Fujita, Chen Kong Teh, H. Usui, F. Yamane, Y. Miyamoto, C. Kumtornkittikul, H. Hara, T. Yamashita, J. Tanabe, M. Uchiyama, Y. Tsuboi, T. Miyamori, T. Kitahara, H. Sato, Y. Homma, S. Matsumoto, K. Seki, Y. Watanabe,

M. Hamada, and M. Takahashi. A 9.7mw aac-decoding, 620mw h.264 720p 60fps decoding, 8-core media processor with embedded forward-body-biasing and power-gating circuit in 65nm cmos technology. In *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pp. 262 –612, feb. 2008.

[NYY+07] M. Nakajima, T. Yamamoto, M. Yamasaki, T. Hosoki, and M. Sumita. Low power techniques for mobile application socs based on integrated platform "uniphier". In *ASP-DAC '07: Proceedings of the 2007 Asia and South Pacific Design Automation Conference*, 2007.

[Opea] OpenACC. Openacc. http://www.openacc-standard.org/.

[opeb] opencv.org. Opencv. http://opencv.org/.

[Opec] OpenMP.org. Openmp. http://www.openmp.org/.

[PAB+05] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first-generation cell processor. In *2005 IEEE International Solid-State Circuits Conference, ISSCC*, Vol. 1, pp. 184–592, 6 February 2005 through 10 February 2005 2005.

[SCS+08] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: A many-core x86

architecture for visual computing. *ACM Transactions on Graphics*, Vol. 27, No. 3, 2008.

[SOW+07] J. Shirako, N. Oshiyama, Y. Wada, H. Shikano, K. Kimura, and H. Kasahara. Compiler control power saving scheme for multi core processors. *Lecture Notes in Computer Science 4339*, pp. 362–376, 2007.

[sta] The Message Passing Interface(MPI) standard. Mpi. http://www.mcs.anl.gov/research/projects/mpi/.

[TAS+10] Takatani, Abe, Sakamoto, Kondo, Fuji, Yamaji, Hamada, Takahashi, Yamamoto, Adachi, and Kanematsu. Development of radiation treatment planning system for heavy-ion therapy. *Technical Report of Japan Society of Medical Physics(in Japanese)*, 2010.

[TST+05] S. Torii, S. Suzuki, H. Tomonaga, T. Tokue, J. Sakai, N. Suzuki, K. Murakami, T. Hiraga, K. Shigemoto, Y. Tatebe, E. Obuchi, N. Kayama, M. Edahiro, T. Kusano, and N. Nishi. A 600mips 120mw 70microa leakage triple-cpu mobile application processor chip. *ISSCC*, 2005.

[WCC+07] P. H. Wang, J. D. Collins, G. N. Chinya, H. Jiang, X. Tian, M. Girkar, N. Y. Yang, G. Y Lueh, and H. Wang. Exochi: Architecture and programming environment for a heterogeneous multi-core multithreaded system. In *PLDI'07: 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 156–166, 10 June 2007 through 13 June 2007 2007.

[WHM+08] Y. Wada, A. Hayashi, T. Masuura, J. Shirako, H. Nakano, H. Shikano, K. Kimura, and H. Kasahara. Parallelizing compiler cooperative heterogeneous multicore. In *Proceedings of Workshop on Software and Hardware Challenges of Manycore Platforms, SHCMP'08*, Jun. 2008.

[WL91] M.E. Wolf and M.S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *Parallel and Distributed Systems, IEEE Transactions on*, Vol. 2, No. 4, pp. 452 –471, oct 1991.

[Wol10] Michael Wolfe. Implementing the pgi accelerator model. In *GPGPU '10: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, 2010.

[WSM+09] Mark Woh, Sangwon Seo, Scott Mahlke, Trevor Mudge, Chaitali Chakrabarti, and Krisztian Flautner. Anysp: anytime anywhere anyway signal processing. *SIGARCH Comput. Archit. News*, Vol. 37, pp. 128–139, June 2009.

[YA10] Yamamoto and Adachi. The development of a dose calculation engine for a particle therapy. *Technical Report of MSS(in Japanese)*, Vol. 21, pp. 36–42, december 2010.

[YAT+10] Yamamoto, Adachi, Takatani, Abe, Sakamoto, and Kanematsu. Development of dose calculation engine for heavy-ion therapy. *Technical Report of Japan Society of Medical Physics(in Japanese)*, 2010.

[YIK+10] Y. Yuyama, M. Ito, Y. Kiyoshige, Y. Nitta, S. Matsui, O. Nishii, A. Hasegawa, M. Ishikawa, T. Yamada, J. Miyakoshi, K. Terada, T. Nojiri, M. Satoh, H. Mizuno, K. Uchiyama, Y. Wada, K. Kimura,

H. Kasahara, and H. Maejima. A 45nm 37.3gops/w heterogeneous multi-core soc. *IEEE International Solid-State Circuits Conference, ISSCC*, pp. 100–101, Feb. 2010.

[YKH+07] Y. Yoshida, T. Kamei, K. Hayase, S. Shibahara, O. Nishii, T. Hattori, A. Hasegawa, M. Takada, N. Irie, K. Uchiyama, T. Odaka, K. Takada, K. Kimura, and H. Kasahara. A 4320mips four-processor core smp/amp with individually managed clock frequency for low power consumption. *IEEE International Solid-State Circuits Conference, ISSCC*, pp. 100–590, Feb. 2007.

Acknowledgements

First of all, I express my sincere appreciation to Professor Hironori Kasahara of Waseda University for his encouragement, constant guidance and support during this work. I also express deep thanks to Associate Professor Keiji kimura of Waseda University for his kind advices and help through my doctoral course. I also express my gratitude to Professor Satoshi Goto and Professor Nozomu Togawa of Waseda University for reviewing this work. A part of this research has been supported by NEDO "Advanced Heterogeneous Multiprocessor", NEDO "Heterogeneous Multicore for Consumer Electronics", and MEXT project "Global COE Ambient Soc". I would like to express sincere appreciation for their valuable technical discussion to Dr. Kunio Uchiyama, Dr. Toru Nojiri, Mr. Koichi Terada, Dr. Hiroaki Shikano from Hitachi, Mr. Jun Hasegawa, Mr. Masayuki Ito from Renesas Electronics, Dr. Moriyuki Takamura from Fujitsu laboratories, Mr. Takahiro Kumura from NEC Corporation, Dr. Masato Edahiro from Nagoya University. Dr. Makoto Sato from Renesas Solutions, Mr. Yasuyuki Takatani from Mitsubishi electric corporation, Mr. Hironori Saki and Mr. Keiji Yamamoto from Mitsubishi space software, Mr. Kenji Nishikawa, Mr. Shigeru Sasaki and Mr. Mitsuo Sawada from Toyota motor corporation, Mr. Yuji Mori, Mr. Mitsuhiro Tani from denso corporation.

I express my sincere thanks to Dr. Yasutaka Wada from Waseda University for his direct guidance, helpful support and continuous encouragement during the

researches. I would like to deep thank to Dr. Hirofumi Nakano Dr. Jun Shirako Dr. Fumiyo Takano Dr. Takamichi Miyamoto Dr. Masayoshi Mase for many kind advices and supports. I also would like to thank to the all students and alumni in Kasahara and Kimura Laboratory, Waseda University.

I would like to give a special thanks to Prof Takeshi Ikenaga from Waseda University, Mr. Keita Miyamura from IBM, Mr. Yoichi Matsuyama from Waseda University, Ms. Miki Yajima from NTT comware, Mr. Hiroaki Tano from SONY, Mr. Yuta Murata from SONY, Mr. Toru Hotta from Yahoo, Mr. Shingo Morita from IBM, Mr. Shu Miyakita from NTT communications, Mr. Kenichi Honma from NTT communications, Mr. Atsushi Yamasaki from KDDI, Mr. Norihiro Sugimoto from NTT data, Mr. Tetsuma Yoshino from SONY, Mr. Takeshi Masuura from SONY, Mr. Teruo Kamiyama from Panasonic, Mr. Masato Hayashi from Hitachi, Mr. Takeshi Watanabe from Fujitsu, Mr. Tetsuya Yamamoto from Japan management system, Mr. Takeshi Sekiguchi from Panasonic.

Finally, I thank my family for their kind support over the years.

Publications

Papers

○ Akihiro Hayashi, Mamoru Shimaoka, Hiroki Mikmi, Masayoshi Mase, Yasutaka Wada, Jun Shirako, Keiji Kimura, and Hironori Kasahara, "OSCAR Parallelizing Compiler and API for Real-time Low Power Heterogeneous Multicores", The 16th Workshop on Compilers for Parallel Computing(CPC2012), January 11-13, 2012, Padova, Italy.

○ Akihiro Hayashi, Yasutaka Wada, Takeshi Watanabe, Takeshi Sekiguchi,Masayoshi Mase, Jun Shirako, Keiji Kimura, and Hironori Kasahara, "Parallelizing Compiler Framework and API for Heterogeneous Multicores", Trans. of IPSJ on Computing Systems, Vol.5(ACS36), 2012 (in Japanese)

○ Akihiro Hayashi, Yasutaka Wada, Takeshi Watanabe, Takeshi Sekiguchi,Masayoshi Mase, Jun Shirako, Keiji Kimura, and Hironori Kasahara, "Parallelizing Compiler Framework and API for Power Reduction and Software Productivity of Real-time Heterogeneous Multicores", Lecture Notes in Computer Science, Springer, Vol. 6548, pp.184-198,Feb., 2011.

Publications

○ Akihiro Hayashi, Yasutaka Wada, Takeshi Watanabe, Takeshi Sekiguchi, Masayoshi Mase, Jun Shirako, Keiji Kimura and Hironori Kasahara, "Parallelizing Compiler Framework and API for Power Reduction and Software Productivity of Real-time Heterogeneous Multicores", The 23rd International Workshop on Languages and Compilers for Parallel Computing (LCPC2010), Rice University, Houston, Texas, Oct. 2010.

Hiroki Mikami, Shumpei Kitaki, Masayoshi Mase, Akihiro Hayashi, Mamoru Shimaoka, Keiji Kimura, Masato Edahiro, and Hironori Kasahara, "Evaluation of Power Consumption at Execution of Multiple Automatically Parallelized and Power Controlled Media Applications on the RP2 Low-power Multicore", The 24th International Workshop on Languages and Compilers for Parallel Computing (LCPC2011), Colorado State University, Fort Collins, Colorado, Sept 8-10, 2011.

Yasutaka Wada, Akihiro Hayashi, Takeshi Masuura, Jun Shirako, Hirofumi Nakano, Hiroaki Shikano, Keiji Kimura, and Hironori Kasahara, "A Parallelizing Compiler Cooperative Heterogeneous Multicore Processor Architecture", Transactions on High-Performance Embedded Architectures and Compilers IV, Lecture Note in Computer Science, Springer, Vol. 6760, pp. 215-233, Nov., 2011.

Takumi Nito, Yoichi Yuyama, Masayuki Ito, Yoshikazu Kiyoshige, Yusuke Nitta, Osamu Nishii, Atsushi Hasegawa, Makoto Ishikawa, Tetsuya Yamada, Junichi Miyakoshi, Koichi Terada, Tohru Nojiri, Masashi Takada, Makoto Satoh, Hiroyuki Mizuno, Kunio Uchiyama, Yasutaka Wada, Akihiro Hayashi, Keiji Kimura, Hironori Kasahara, Hideo Maejima, "A 45nm Heterogeneous Multi-core SoC Sup-

porting an over 32-bits Physical Address Space for Digital Appliance", Proc. of 13th IEEE Symposium on Low-power and High-Speed Chips(COOL Chips XIII), Yokohama, Japan, Apr. 2010.

Hiroki Mikami, Jun Shirako, Masayoshi Mase, Takamichi Miyamoto, Hirofumi Nakano, Fumiyo Takano, Akihiro Hayashi, Yasutaka Wada, Keiji Kimura, Hironori Kasahara, "Performance of OSCAR Multigrain Parallelizing Compiler on Multicore Processors", The 14th Workshop on Compilers for Parallel Computing(CPC 2009), Zurich, Switzerland Jan. 2009.

Yasutaka Wada, Akihiro Hayashi, Takeshi Masuura, Jun Shirako, Hirofumi Nakano, Hiroaki Shikano, Keiji Kimura, Hironori Kasahara, "Parallelizing Compiler Cooperative Heterogeneous Multicore", Proc. of Workshop on Software and Hardware Challenges of Manycore Platforms (SHCMP 2008), Beijing, China, Jun. 2008.

Yasutaka Wada, Akihiro Hayashi, Takeshi Masuura, Jun Shirako, Hirofumi Nakano, Hiroaki Shikano, Keiji Kimura, Hironori Kasahara, "Parallelization of MP3 Encoder using Static Scheduling on a Heterogeneous Multicore. Trans. of IPSJ on Computing Systems, Vol. 49(ACS22), 2008 (in Japanese)

Hiroaki Shikano, Masaki Ito, Kunio Uchiyama, Toshihiko Odaka, Akihiro Hayashi, Takeshi Masuura, Masayoshi Mase, Jun Shirako, Yasutaka Wada, Keiji Kimura, Hironori Kasahara "Software-Cooperative Power-Efficient Heterogeneous Multi-Core for Media Processing" The 13th Asia and South Pacific Design Automation Conference(ASP-DAC2008), Seoul, Korea, Jan. 2008.

Technical Reports (in Japanese)

Akihiro Hayashi, Takuji Matsumoto, Hiroki Mikami, Keiji Kimura, Keiji Yamamoto, Hironori Saki, Yasuyuki Takatani, Hironori Kasahara, "Automatic Parallelization of Dose Calculation Engine for A Particle Therapy on SMP Servers" Technical Report of IPSJ, Vol.2011-ARC189HPC132-2(HOKKE2011), Nov. 2011

Akihiro Hayashi, Takeshi Sekiguchi, Masayoshi Mase, Yasutaka Wada, Keiji Kimura, Hironori Kasahara, "Hiding I/O overheads with Parallelizing Compiler for Media Applications", Technical Report of IPSJ, Vol.2011-ARC-195OS117-14, Apr. 2011.

Takuya Sato, Hiroki Mikami, Akihiro Hayashi, Masayoshi Mase, Keiji Kimura, Hironori Kasahara, "Evaluation of Parallelizable C Programs by the OSCAR API Standard Translator", Technical Report of IPSJ, Vol.2010-ARC-191-2, Oct. 2010.

Akihiro Hayashi, Yasutaka Wada, Takeshi Watanabe, Takeshi Sekiguchi, Masayoshi Mase, Keiji Kimura, Masayuki Ito, Jun Hasegawa, Makoto Sato, Toru Nojiri, Kunio Uchiyama, Hironori Kasahara, "A Compiler Framework for Heterogeneous Multicores for Consumer Electronics", Technical Report of IPSJ, Vol.2010-ARC-190-7(SWoPP2010), Aug. 2010.

Yasutaka Wada, Akihiro Hayashi, Takeshi Watanabe, Takeshi Sekiguchi, Masayoshi Mase, Jun Shirako, Keiji Kimura, Masayuki Ito, Jun Hasegawa, Makoto Sato,

Toru Nojiri, Kunio Uchiyama, Hironori Kasahara, “Performance of Power Reduction Scheme by a Compiler on Heterogeneous Multicore for Consumer Electronics RP-X”, Technical Report of IPSJ, Vol.2010-ARC-190-8(SWoPP2010), Aug. 2010.

Teruo Kamiyama, Yasutaka Wada, Akihiro Hayashi, Masayoshi Mase, Hirofumi Nakano, Takeshi Watanabe, Keiji Kimura, Hironori Kasahara, “Performance Evaluation of Parallelizing Compiler Cooperated Heterogeneous Multicore Architecture Using Media Applications”, Technical Report of IPSJ, Vol.2009-ARC-173, Jan. 2009.

Yasutaka Wada, Akihiro Hayashi, Taketo Iyoku, Jun Shirako, Hirofumi Nakano, Hiroaki Shikano, Keiji Kimura, Hironori Kasahara, “A Hierarchical Coarse Grain Task Static Scheduling Scheme on a Heterogeneous Multicore” Technical Report of IPSJ, Vol.2007-ARC-174-17(SWoPP2007), Aug. 2007.

Akihiro Hayashi, Taketo Iyoku, Ryo Nakagawa, Shigeru Matsumoto, Kaito Yamada, Naoto Oshiyama, Jun Shirako, Yasutaka Wada, Hirofumi Nakano, Hiroaki Shikano, Keiji Kimura, Hironori Kasahara, “Compiler Control Power Saving for Heterogeneous Multicore Processor”, Technical Report of IPSJ, Vol.2007-ARC-174-18(SWoPP2007), Aug. 2007.

Symposium

Akihiro Hayashi, Takuji Matsumoto, Hiroki Mikami, Keiji Kimura, Keiji Yamamoto, Hironori Saki, Yasuyuki Takatani, Hironori Kasahara, "Automatic Parallelization of Dose Calculation Engine for A Particle Therapy", IPSJ Symposium on High Performance Computing and Computer Science(HPCS2012), Jan 2012.(with review)

Akihiro Hayashi, "OSCAR Parallelizing Compiler Cooperative Heterogeneous Multi-core Architecture", The 5th Ambient GCOE Symposium "Software Technologies and its Service for Ambient SoC", Sep., 2009.

Posters

Akihiro Hayashi, Yasutaka Wada, Hiroaki Shikano, Jun Shirako, Keiji Kimura, Hironori Kasahara, "Compiler Cooperative Heterogeneous Multicore Architecture", Poster session of Waseda University Global COE Program the 2nd International Symposium "Ambient SoC; Recent Topics and Nano-Technology and Information Technology Applications"., Jul. 2008.

Akihiro Hayashi, Yasutaka Wada, Hiroaki Shikano, Teruo Kamiayama, Takeshi Watanabe, Takeshi Sekiguchi, Masayoshi Mase, "OSCAR Parallelizing Compiler Cooperative Heterogeneous Multi-core Architecture", The Eighteenth International Conference on Parallel Architectures and Compilation Techniques (PACT2009), Raleigh, North Carolina., Sep., 2009.(with review)

Invited Talks

Akihiro Hayashi, Yoichi Matsuyama, Ri Goto, The 6th Ambient GCOE Symposium "Toward the Realization of Ambient SoC", Jun. 2010(in Japanese)

Akihiro Hayashi, "Orchestrating multi-core processors is difficult. How can we cope with them?", Global COE Workshop for RA Members., Jan., 2010.