DESIGN AND IMPLEMENTATION OF TOOL-CHAIN FRAMEWORK TO SUPPORT OPENMP SINGLE SOURCE COMPILATION ON CELL PLATFORM

by

Yi Jiang

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Master of Science with a major in Electrical and Computer Engineering

Winter 2007

DESIGN AND IMPLEMENTATION OF TOOL-CHAIN FRAMEWORK TO SUPPORT OPENMP SINGLE SOURCE COMPILATION ON CELL PLATFORM

by

Yi Jiang

Approved: _____

Guang R. Gao, Ph.D.
Professor in charge of thesis on behalf of the Advisory Committee

Approved: _____

Gonzalo R. Arce, Ph.D.
Chair of the Department of Electrical and Computer Engineering

Approved: _____

Eric W. Kaler, Ph.D.
Dean of the College of Engineering

Approved: _____

Daniel Rich, Ph.D.
Provost

ACKNOWLEDGMENTS

I would like to thank my advisor Prof. Guang R. Gao. His endless enthusiasm influences me greatly and motivates me to work harder on my research. Furthermore, his broad knowledge, deep insights and concise guidance always guide my research along the right direction. Without his help, it would be impossible for me to learn the most advanced technologies and thoughts in CAPSL.

I would like to thank my project supervisor Dr. Ziang Hu. His solid knowledge and experience on both parallel architecture and system software provide me precious guidance on this research work. I will always be grateful for his patience during our many discussions.

I would like to thank Joseph Manzano, Ge Gan and Andrew Russo. The experience to work with them on Cell platform was simply wonderful. I learned so much from them. Joseph also helped me to review the technical details and writing of this thesis.

I would like to thank Brian Lucas, who helped me to review this paper and gave me many useful suggestions on writing.

Finally, I would like to thank Eunjung Park and Lu Zhang, who helped me a lot with Latex and everybody in CAPSL for their support.

DEDICATION

To my parents.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

The well-known performance and power bottlenecks in traditional architecture design, in conjunction with the sustained demand for high performance in real world applications, stimulated the creation of new designs that utilize multi-cores in one processor. There are two approaches in multi-core design: homogeneous and heterogeneous. The homogeneous design is based on the replication of simple cores. It is easier for developers to port existing applications to this kind of platform. However, great diversity exists among applications and a homogeneous multi-core chip cannot be optimal for heterogeneous workloads. Therefore, more and more multi-core designs tend to utilize heterogeneous cores and specialized accelerators. The Cell Broadband Engine (CBE) [1, 2] is a representative. Every Cell processor integrates one Power Processing Engine(PPE) core and eight Synergistic Processing Engines (SPE). A PPE core has a traditional memory and cache hierarchy and it accesses memory via caching mechanism. On the other hand, each SPE core only has 256K local storage and accesses its own local storage directly. All data exchange between the SPUs and shared system memory is via high-latency DMA operation.

Therefore, this architecture presents great challenges to programmers who want to utilize parallelism: **(1)** Threads running on PPE are far different from the ones on SPEs, both in capability and ISAs. The users have to take care of programming threads in each ISA, as well as their cooperation and synchronization. **(2)** The SPE local storage is so limited that SPE code or data may have to be partitioned into overlay sections. Given the explicit memory hierarchy, it is necessary for the programmers to issue DMA instructions at the appropriate time and transfer

them to or from system memory. **(3)** Shared data in SPE code also needs to be loaded from and stored to main memory. The Cell processor can guarantee the coherence for all DMA transactions. However, the user would have the responsibility to keep data coherence among SPE local storage.

This thesis is inspired by the Open Opell project whose goal is to develop the GNU-Based OpenMP on the CBE platform and address the challenges mentioned above. Our solution involves the whole tool chain and a runtime system. The CBE programmer can take advantage of the convenience of this abstract share-memory programming model. Specifically, we design and implement single source compilation to attack the first challenge. Then the "ghost" mechanism, which invokes overlays and partition manager libraries, helps to deal with the second challenge. Lastly, we try to tackle the third challenge by the software cache library.

In this thesis, we mainly focus on my work in the assembler and linker part of the Open Opell system, which addresses three specific problems: **(1)** Given a heterogeneous system like CBE, how to support single source compilation? **(2)** How to design a mechanism to automatically invoke code/data overlays and hide all details to the end-user? **(3)** How to make the solution and implementation robust and efficient?

The main contributions of this thesis are: **(1)** We have completed the design of interfaces between different phases in toolchain to support single openMP source program compilation. **(2)** We have designed the "ghost" mechanism to invoke the overlays and hide the details about the overlay to high-level users. It will work with the partition manager run-time library to complete automatic code/data load if needed. **(3)** We have implemented the proposed design in our Open Opell system in a robust and efficient manner. We will focus on the assembler and linker implementation in this thesis. We have used a series of experiments with several non-trivial benchmarks to prove our claim.

Chapter 1

INTRODUCTION

In this chapter, we introduce the background information of this thesis. This chapter is organized as follows: section 1.1 is a brief introduction about multi-core architecture. Then we will focus on the heterogeneous system and review the opportunity and challenge in section 1.2. The main contributions of this thesis are presented in section 1.3.

1.1 Coming Era of Multi-Core Platform

In the past, computer architecture designers have made great efforts to fulfill the request for more computing power by creating new types of architectures. A traditional von Neumann processor consists of a single program counter (PC) which points to the instruction to be executed in the current cycle. That is the only instruction executed in that cycle and therefore the instructions executed per cycle (IPC) would be at most one. By observing that there are a lot of instructions that can actually execute at the same time, later computer architectures try to go beyond an IPC of one by exploiting the instruction level parallelism (ILP). Then higher performance can be achieved. The superscalar and VLIW (Very-Long Instruction Word) architectures are representatives. Intuitively, both of them have multiple function units. In superscalar processor, it maintains an instruction window where the in-order issued instructions can execute immediately when all operands needed are ready. While in VLIW processors, compiler will be assigned the responsibility

to keep instructions in legal order. Then instructions that are considered to be able to execute in parallel are packed in a single very long instruction word.

Now these two kinds of processors are widely used both in academia and industry. In order to continuously improve the performance of processors, architects keep increasing chip clock speed to higher and higher levels. Unfortunately, now we have to face other "walls" that prevent us from pursuing even higher performance.

The first barrier to performance is the Memory Wall [3], which is caused by the increase of memory access latency as measured in cycles, as well as latency induced limits on memory bandwidth. According to Moore's Law, the number of transistors that can be inexpensively placed on an integrated circuit is increasing exponentially, approximately doubling every two years, while main memory speeds double only about every ten years. Therefore the effective DRAM latency increases with every generation. In a multi-GHz processor it is common for DRAM latencies to be measured in the hundreds of cycles; in symmetric multiprocessors (SMP) with shared memory, main memory latency is approaching 1000 processor cycles. As a result, memory accesses dominate code performance.

Secondly, power dissipation is proportional to the clock frequency, which also imposes a potential limit on clock rates. As gates get smaller, leakage power dissipation gets worse, because gate dielectric thicknesses must proportionately decrease. Therefore, unless materials technology breakthroughs, a significant increase in clock speed without heroic and expensive cooling system is not feasible. These difficulties compose the "Power Wall".

Thirdly, though advances in compiler technology together with significantly new and different hardware techniques have improved ILP significantly, there are several limitations that seem very difficult to overcome. These problems make up the "ILP Wall". In superscalar processors, the total instruction window size is limited by the required storage, the comparisons and a limited issue rate, which makes

a larger window less helpful. On the other hand, in the view of application, the instructions that can be parallel executed in one single thread are quite limited even if we increase the optimization scope. Furthermore, the real processors usually have a more limited number of functional units, which implies that the actual instruction, that may be issued, executed or committed in the same cycle is even smaller than the issue size. So we have to face the fact that the point of diminishing return has been reached.

Upon reaching these walls, the designers move to a new type of architecture: multi-core architecture. Instead of building a single and complex processor in a single integrated circuit(IC), called a die, the multi-core architecture design combines a number of simple processors on a single die. Providing reductions in frequency for each single core, the whole power consumptions drops dramatically. We can see that multi-core architectures can alleviate the power dissipation problem without losing the computation power [4]. Furthermore, the multi-core architecture design naturally exploits thread-level parallelism (TLP), which implies that the scope we search for parallelism expands significantly and it is expected to be widespread in future applications optimization [5]. As a result, both the industry and academia are moving their microarchitecture design to multi-core. Currently, dual-core and quad-core design dominates the PC market, and the IBM Cell chip [6] is used in a very popular game console, the PlayStation 3. Besides that, some large-scale multi-core chips like the IBM Cyclops-64 [7] chip and Intel 80-cores chip [8] were announced.

1.2 Heterogeneous Multi-core:Opportunities and Challenges

There are two approaches in multi-core design: homogeneous and heterogeneous. While homogeneous multi-core processors like dual-core and quad-core are currently lead in high performance computing market, their design, which is largely

based on replication of simple scalar cores, exhibits well-known limitations. The reason is that great diversity exists among applications and a homogeneous chip cannot be optimal for heterogeneous workloads. If every application of interest has different run-time behavior, making decisions in homogeneous multi-core design can be very difficult, thus making the struggle between general purpose and a proliferation of independent "optimal" design one of the key challenges. As applications become more complex and more demanding of computational performance, the pressure to tap other types of processing engines will increase.

A increasing number of designs tend to utilize heterogeneous cores and specialized accelerators. This expansion adds many new degrees of freedom in the design space. For example, if the architecture is designed to run multi-media applications very efficiently, large register files and SIMD (single instruction multiple data) instructions are required. The myriad possibilities created by this extra set of degrees of freedom are rather exciting. At the same time, it could also expose great challenges.

One example is FPGAs (Field Programmable Gate Arrays). They are always hot topics in the supercomputing community because they are reconfigurable, and have wide applicability for HPC applications. Also, unlike coprocessors and vector processors, FPGAs are more general-purpose compute engines. Some commercial products of this kind have been developed by companies like DRC [9].

The growing interest of the HPC community in IBM's Cell chip is another example. This chip contains both a scalar (PowerPC) CPU and eight vector compute engines. The scalar CPU is used to control the vector cores and manage the chip's memory hierarchy. We will focus on this chip in this thesis. However, these designs can be easily applied to other heterogeneous platforms.

In general, developing new practical computer architecture represents a huge investment which requires convincing customers, either in industry or academia, to

switch to the new architecture. However performance is not the only criterion. Good programmability would also play a key role in success. There was no easy way to extract performance from those architectures. Only experienced parallel programmers can achieve real benefits from new features in new architecture. So improving the programmability is a key challenge to architecture and system software designers.

With the appearance of these heterogeneous systems, the programmers face even more challenges of adapting their applications so that they can make good use of all the hardware features. We would like to explain these challenges raised by the CBE as a typical example.

The first challenge is to program the different components, PPE and SPE, in the system. The PPE is designed as general purpose and to run full-fledged operation systems. On the other hand, the bulk of the computing power of the Cell BE is provided by the eight SPEs. The SPE is a processor designed to accelerate media and streaming workloads. The programmer has to design the programs to adapt to both of them. Furthermore, the thread running on PPE and SPE have different purpose, which makes coordination much more difficult.

The limited local storage and explicit memory raise the second challenge. In this case, SPE code may need to be partitioned into multiple overlaid binary sections instead of generating it as a large monolithic section. Also, the developers have to inspect the run-time behavior of program and transfer appropriate SPE code or data to and from system.

The third challenge that the users have to face is shared data in SPE code. It also needs to be loaded from and store to main memory. Then the programmer should handle the memory model very carefully since the Cell hardware only ensures DMA transactions are coherent but does not provide coherence for data residing in the SPE local stores. As far as performance is concerned, the developer would also like a mechanism to allow reuse of temporary buffers in the local memory, and

therefore there is no need for DMA transfers for all accesses to main memory.

1.3 Contributions

By figuring out these challenges in the last section, we consider that reliable and easy-to-use programming models must be offered to the CBE architecture programming communities that enable to attack these challenges and exploit the heterogeneous and parallel characteristics of this architecture. With this goal, in this paper we present our OpenMP system named Open Opell, which provides the programmers with the abstraction of shared-memory address space.

The Open Opell system involves the whole tool chain and a runtime system. Specifically, we design and implement single source compilation to attack the first challenge. Then the "ghost" mechanism, which invokes overlays and partition manager libraries, helps to deal with the second one. And then the third challenge can be managed by software cache library. All these features were implemented in GCC compiler version 3.3 and all extensions to assembly language and object format, such as ELF file format, follow the standards in GNU family.

In this paper, We will focus on my contributions in the tool chain part of Open Opell system:

(1) We have completed the design of interfaces between different phases in toolchain to support single openMP source program compilation. **(2)** We have designed the "ghost" mechanism to invoke overlays and hide the details of the overlay to high-level users. It will work with partition manager run-time library to complete automatic code/data loading if needed. **(3)** We have implemented the proposed design in the Open Opell system in a robust and efficient manner. We will focus on the assembler and linker implementation in this thesis. We have used a series of experiment with several non-trivial benchmarks to prove our claim.

The thesis is organized as follows. Chapter 2 will provide a brief introduction about the CBE architecture and our research project. In Chapter 3, we will discuss

our design to support single source openMP program compilation. The design of the "ghost" mechanism will be explained in Chapter 4. In Chapter 5, we will report the implementation and evaluation of our design in CBE platform. Related Work will be mentioned in Chapter 6. Finally, in Chapter 7 we will make a conclusion and propose the future work.

Chapter 2

PROJECT OVERVIEW

In the last chapter, we explained that CELL is a typical heterogeneous architecture which presents several challenges to programmers. This fact motivates our research group to support a reliable and easy-to-use programming model. In this chapter, we will present an overview of our research project which plans to address these challenges by supporting a GNU-Based OpenMP on the CBE platform. This chapter is organized as follows: in section 2.1, we will describe the CELL architecture briefly and focus on the parts we are interested in. Then, we will also review the OpenMP programming model in section 2.2. After that, the overview of our research project Open Opell, as well as the key components of it, is provided in section 2.3.

2.1 Cell Architecture

A simple objective of the CBE architecture is to provide power-efficiency and cost-effective high-performance for one of the most popular consumer applications: video games. Therefore, the design was based on the analysis of a broad range of workloads in areas such as graphics transformation and lighting, cryptography, stream media processing, fast-Fourier transforms (FFT) and scientific workloads. While featured with low cost and modest power consumption [10], it is also able to achieve very impressive peak performance. A number of studies [11, 12, 13, 14] have been done to demonstrate the performance potential of the CBE. Cell has a theoretical peak performance of over 200 Gflops for single-precision FP calculations

and a peak memory bandwidth of over 25 GB/s. Furthermore, IBM recently announced a system with 16000 Cells with a projected peak performance reaching a Petaflop[15].

Cell employees heterogeneous multi-core chip architecture that integrates one Power Processing Engine (PPE) core and eight Synergistic Processing Engines (SPE) which are based on a novel single-instruction multiple-data (SIMD) architecture. A brief architecture picture is presented in Figure 2.1. According to Figure 2.2, provided with rich vector processing units and register files, the SPE SIMD cores are targeted for data-intensive processing, like that found in cryptography, media and scientific applications. The EIB plays the role of communication with each core. This configuration combines the flexibility of PPE core with the functionality and performance-optimized SPE SIMD cores.

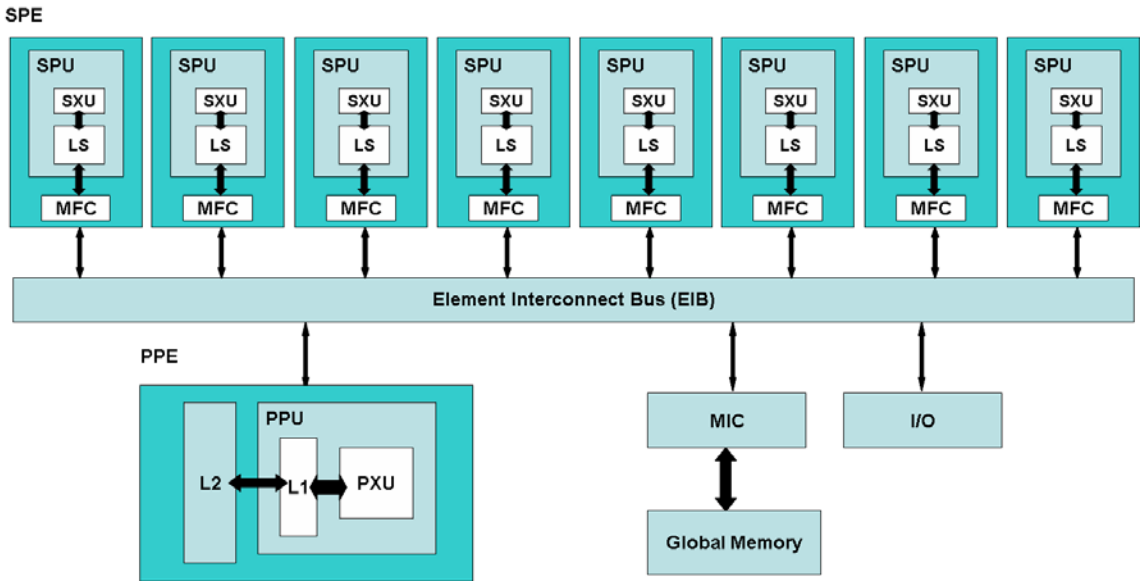

Figure 2.1: Overview of Cell processor architecture

The PPE includes the Power Processing Unit (PPU) that consists of a traditional memory hierarchy with main memory, L1 and L2 caches. Each SPE consists of a Synergistic Processor Unit (SPU), 256KB Local Storage (LS), and a Memory

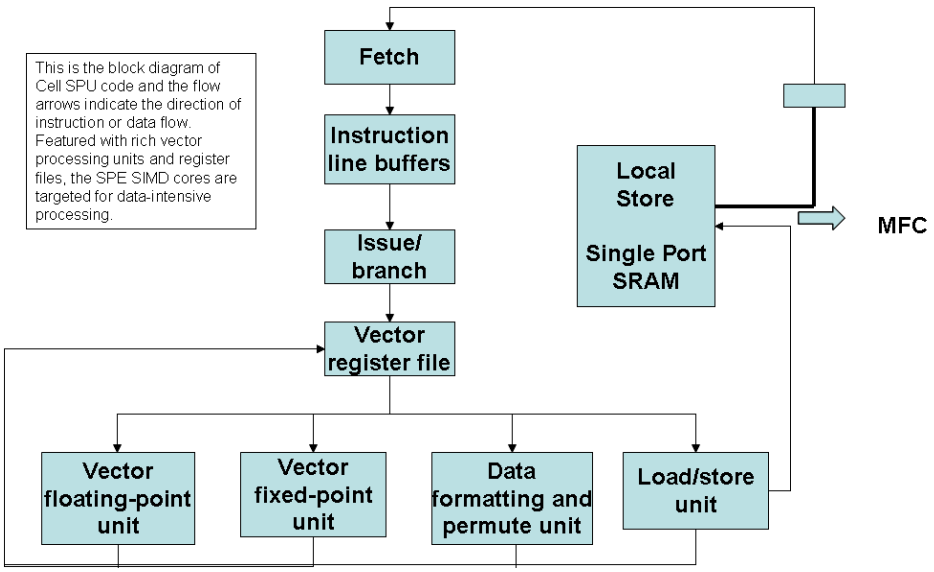

Figure 2.2: Cell SPU Block Diagram

Flow Controller (MFC). Unsurprisingly, the PPE core can access memory via the traditional caching mechanism. While on an SPU, memory access instructions can only load from and store to locations in the LS of that SPE. Since each SPU only has 256KB LS, if it needs to access main memory or the LS of another SPE, it must issue a DMA command to its MFC explicitly instructing the MFC to do a data transfer between its LS and the destination. All the SPE LS, the Memory Interface Controller (MIC) , and the PPE's L2 cache are inter-connected via a high bandwidth Element Interconnect Bus (EIB) with 16 Bytes/cycle for each link. Each LS in SPU is completely managed by software. Moreover it must contain both the code and data used in SPU execution. The latency of DMA operations between LS and main memory is quite high, approximately in the order of 100-200 SPU cycles [16]. At the same time it is possible for a DMA transaction on the EIB that involves the LS of one SPE to be initiated by another SPU or by the PPU.

A series of real world applications, such as graphics, AI algorithms and physics simulations, have been developed in the Cell processor to demonstrate how

to translate raw chip performance into application performance. An example demonstrating the efficient use of the CBE architecture is the implementation of a smooth particle hydrodynamics simulation by Hjelte [17]. These studies show that using Cell for real-world HPC applications is feasible.

However, programming the Cell processor and dealing with coupled PPE and eight SPE processors is still a very complex task. First of all, the programmers need to partition an application carefully to fit the limited local memory constraints of the SPE. Secondly, they need to consider parallelization across the multiple SPEs. Finally, the user will have to explicitly manage DMA data transfers for SPE computation and get source code compiled for two distinct ISAs. Therefore, programmability in the Cell processor [18] becomes a very hot topic and has been significantly enhanced by a series of research [19]. These efforts have resulted in the development of several programming models for the Cell Broadband Engine Architecture ranging from function offload model [20], the composition of functional processing pipelines [21], and lightweight process organization to compiler directed auto-parallelization from a single source program [22]. The goal of our research group is also to provide a programming model that allows developers to make the best use of development resources and achieve both high productivity and high performance. We decided to support OpenMP on the Cell processor to manage this complexity.

2.2 OpenMP Review

The OpenMP Application Program Interface (API)[23, 24] supports multi-platform shared memory parallel programming in C/C++ and Fortran on different architectures. since the era of multi-core processors has begun, more and more researches have been done on OpenMP[25, 26, 27, 28, 29]. It provides programmers a simple and flexible interface for developing parallel applications for each

platform[30, 31, 32]. OpenMP is an explicit programming model, offering the programmer full control over parallelization. The API mainly covers user-directed parallelization, wherein the user explicitly specifies the actions to be taken by the compiler and runtime system in order to execute the program in parallel. OpenMP-compliant implementations[33, 34] are not required to check for dependencies, conflicts, deadlocks, race conditions, or other problems that result from non-conforming programs. The user is responsible for using OpenMP in his application to produce a conforming program. The motivation for us to choose OpenMP is that OpenMP is currently an industrial standard for writing parallel programs and it is productively used [1].

The OpemMP API is composed of three primary API components: compiler directives, runtime library routines and environment variables. Figure 2.3 briefly represents the OpenMP Architecture.

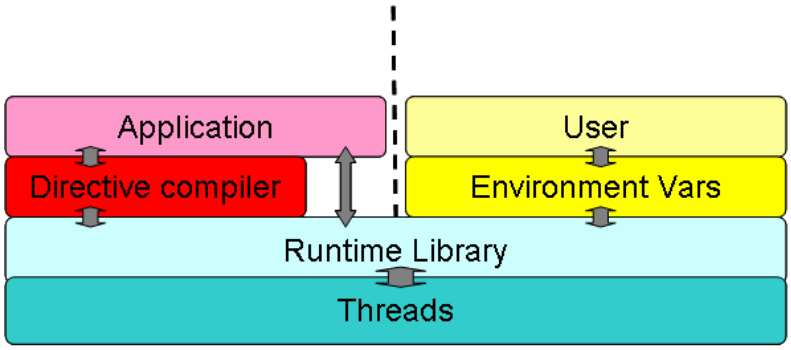

Figure 2.3: OpenMP Architecture

OpenMP uses the fork-join model of parallel execution as shown in Figure 2.4. All OpenMP programs begin as a single process: the master thread. The master thread executes sequentially until the first parallel region construct is encountered. A parallel region is a region of code that is designated to be run by a team of threads. This is achieved either by a #pragma omp parallel directive, or by using one of the

[1] It has about 10% of parallel programming user community

compound pragmas, like #pragma omp parallel for or #pragma omp sections. The master thread then creates a team of parallel threads. This process is named a "fork". The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads. When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread. This process is called a

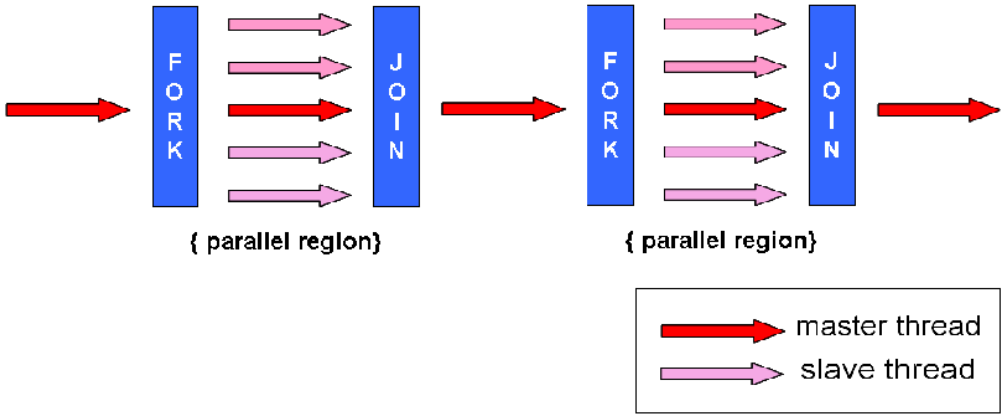

Figure 2.4: OpenMP fork-join model

OpenMP is a flexible model and has been eventually accepted as an industry standard for writing parallel programs on shared memory architecture. The OpenMP Architecture Review Board include almost all mainstream IT companies like Compaq and HP, Intel Corporation, International Business Machines (IBM), Kuck & Associates, Inc. (KAI), Silicon Graphics, Inc., Sun Microsystems, Inc. and the U.S. Department of Energy ASC program. At the same time, OpenMP is available in some famous compilers like the GNU compiler, IBM compiler and Intel Compiler.

2.3 Open Opell: GNU-Based OpenMP system

In the last section, we mentioned that the OpenMP programming model is usually used for share-memory platforms. While the Cell platform is not based on traditional share-memory architecture, our research project, Open Opell system, plan to support the OpenMP on CBE platform and provide the users the convenience of programming. Specifically in our system, we support the following features:

(1) Single Source Compilation Compiler. Since the Cell processor employees heterogeneous design, we want to provide the users an integrated single source compilation framework and avoid their extra efforts on programming on two different ISAs on Cell. This part will be presented in section 2.3.1

(2) Execution Handler Runtime. In section 2.3.2, we will expalin our OpenMP execution model. In this model, all GOMP service calls in SPU cores will be actually sent to PPU core and execute there. Therefore, we can implement very light-weighted SPU threads.

(3) Software Cache. This feature is to help the users handle the irregular main memory accesses. Otherwise, the users have to issue DMA instructions manually every time. This feature will be described in section 2.3.3.

(4) Overlay and Partition Manager Runtime. Overlay technology enables several program sections to share the same runtime address. This feature, as well as the partition manager runtime library, will be used to attack the limited local storage problem in Cell processor. We will briefly introduce it in section 2.3.4. In chapter 4, we will present more details on them.

2.3.1 Single Source Compilation Compiler

The motivation for single source compilation is to address the inconvenience of programming on two different ISAs on Cell. The Listing 2.1 and 2.2 show the source code with the same functionality and different programming model. In the program, we will add vector a[N] to b[N] and store the results to c[N]. This program

is very suitable to be executed in parallel because there is no dependence between iterations. In OpenMP program as List 2.1, we will use the pragma *omp for* to indicate that this loop could be executed once with different threads performing different iterations of the loop in parallel. Some parameters are also specified to guide the paralization. The shared variables are the vectors, *a, b* and *c*, as well as the number of iterations that each thread process at one time, named *chunk*. Every thread will keep their own copy of variable *i*. All the iterations in this loop will be scheduled *dynamically*. List 2.2 is an implementation of the same function via spe_thread programming model. The users need to program in both PPU and SPU cores. In PPU side, the users need to divide the data manually, compute all parameters like upper and lower bound, and use a structure variable, called *cb*, to pass these parameters to SPUs. The SPU threads can be created via the API "spe_create_thread". Implementation in SPU side is even more complicated, the programmers first need to get the parameters by acquiring *cb*. Then some other DMA instructions will be issued to get the vectors. After computation, the result will also be written back via DMA. In this simple case we just assume the data can fit the local storage. If it is not the case, the users also need to partition the data and DMA them piece by piece.

Listing 2.1: Single Source Compilation OpenMP Program

```
1  main() {
2     ...
3  #pragma omp parallel for \
4                 shared(a, b, c, chunk) \
5                 private(i) \
6                 schedule(dynamic, chunk)
7  for (i=0; i<N; i++)
8    {
9      c[i] = a[i] + b[i];
```

```
10      }
11      ...
12    }
```

Listing 2.2: SPE_THREAD Program

```
1  /* PPU side header file */
2  typedef struct _control_block {
3    unsigned int chunk_size;
4    unsigned int i;
5    unsigned long long a;
6    unsigned long long b;
7    unsigned long long c;
8  } control_block;
9
10 /* PPU side source code */
11 /* we allocate one control block for each SPE */
12 control_block cb __attribute__ ((aligned (128)));
13
14 /* this is the pointer to the SPE code,
15  to be used at thread creation time */
16 extern spe_program_handle_t simpleDMA_spu;
17
18  ...
19 int main()
20 {
21    ...
22 /*parameters will be put in the control block */
23 for (int j = 0; j < NUM_THREADS; j++) {
24   cb.addr.a = &a[0];
25   cb.addr.b = &b[0];
26   cb.addr.c = &c[0];
27   cb.addr.chunk = chunk_size;
```

```
28    cb.addr.i = N / NUM_THREADS * j;
29
30    speid = spe_create_thread (gid, &add_spu, \
31            (unsigned long long *) &cb, NULL, -1, 0);
32    if (speid == NULL) {
33      exit (1);
34    }
35  }
36   /* wait for the single SPE to complete */
37  spe_wait(speid, &status, 0);
38     ...
39  return 0;
40  }
41
42  /* SPU side source code */
43  /* Local copy of the control block*/
44  control_block cb __attribute__ ((aligned (128)));
45
46  /* Here's the local copy of the arrays;
47     BUF_SIZE is less than the max possible DMA size*/
48  int a[BUF_SIZE] __attribute__ ((aligned (128)));
49  int b[BUF_SIZE] __attribute__ ((aligned (128)));
50  int c[BUF_SIZE] __attribute__ ((aligned (128)));
51
52  int main(unsigned long long speid,
53            addr64 argp, addr64 envp)
54  {
55    int i;
56
57    /* DMA control block from system memory. */
58
59    mfc_get(&cb, argp.ull, sizeof(cb), 31, 0, 0);
60    mfc_write_tag_mask(1<<31);
```

```
61    mfc_read_tag_status_all();
62
63    mfc_get(&a, cb.a, BUF_SIZE, 31, 0, 0);
64    mfc_get(&b, cb.b, BUF_SIZE, 23, 0, 0);
65
66    mfc_write_tag_mask(1<<31);
67    mfc_read_tag_status_all();
68    mfc_write_tag_mask(1<<23);
69    mfc_read_tag_status_all();
70
71   for (int j = 0; j < BUF_SIZE; j++) {
72      c[j] = a[j] + b[j];
73   }
74
75    mfc_get(&b, cb.b, BUF_SIZE, 31, 0, 0);
76    mfc_write_tag_mask(1<<31);
77    mfc_read_tag_status_all();
78 }
```

Obviously, to the users, developing an OpenMP program like List 2.1 is much easier than spe_thread program. In a program like List 2.2, the users have to divide the program into two parts and handle the communications very carefully. Thus the single source OpenMP programming model outperforms the spe_thread model on programmability.

From the user point of view, single source compilation looks as if there is only one work flow in the compilation process. On the contrary, from the view of the tool chain, it has to deal with different ISA and the work flow would be divided into two and merge together eventually to generate the final executable file, though these details would be invisible to the user. There are two possible framework designs:

Both designs will somehow compile the single source files into SPU binary and PPU binary respectively. And then the embedder will be used to transfer the

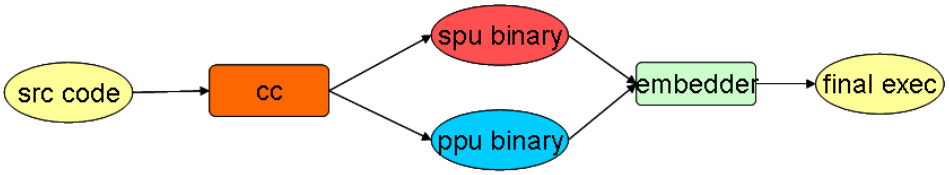

Figure 2.5: Single Source Compilation Design I

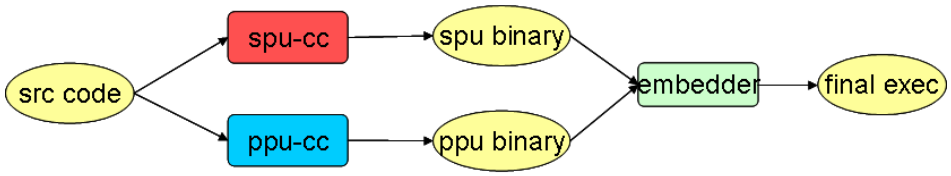

Figure 2.6: Single Source Compilation Design II

SPU binrary to a PPU library. Finally, this library will be linked with PPU binary to the final executable file. The difference is that, in the design I, only one compiler is involved and it will take charge of the generation of both SPU and PPU binary; On the other hand, the design II will use two different compilers to generate two types of binary respectively. The advantage of design I is that the interface is more straight forward. However, we decided to take the second design because these two architectures are so different and it is easier to maintain two compiler back-ends independently. It also makes it possible for us to retarget part of them to other heterogeneous platform backends.

2.3.2 Execution Handler Runtime

In this section, we present the execution model of our system by going though the sample program in List 2.1. Execution handler is the OpenMP runtime we developed as a part of Open Opell project. In our execution model, when encountering a GOMP service call, a SPU thread will send a request as well as the necessary parameters to the PPU and wait. After finishing the service, the PPU will also send the results back to the SPU. Then the SPU can use it to do further computation. The detailed communication process is showed by Figure 2.7.

In this process, each GOMP thread will create a thread on the PPU side and each is "mirror" to a SPU thread. All of the GOMP runtime data structures reside in the PPU and most of the GOMP runtime routines are executed on the PPU. There are two execution modes of these runtime routines; it could be executed either by the master thread directly, or through function-shipping from the SPU thread to the corresponding PPU thread. In the second mode, the SPU thread will store the parameters for a GOMP runtime call into a shared command buffer, which is known by both the PPU and its shadow SPU thread. Then, the SPU thread will send the function-shipping request to an outbound mailbox, on which, the corresponding PPU thread is listening. After the PPU thread executes the requested GOMP runtime function, the results will be passed back to SPU through the same command buffer.

The OpenMP compiler will generate two copies of the code: one for the SPU and one for the PPU. For the outlined parallel function, the code for SPU is the same as the original GOMP generated code. The function code for PPU is different: the original function body is replaced by a call to a service routine, which contains a loop to handle GOMP runtime requests from the SPU thread. On the SPU side, the main function is the execution handler, which is a loop waiting for the next task from the PPU, setting up the execution environment, and executing the tasks. Whenever a GOMP runtime function is called, a wrapper function with the same signature is executed. It will store the parameters into the command buffer, send the function-shipping request to the PPU, and wait for the results. The real GOMP runtime function is executed on the PPU side, while all of the computation work is executed on the SPU side.

In the rest of this section, we will review the example posted in List 2.1 and explain the details in our execution model. List 2.3 is the master thread code generated by the compiler around the parallel region. The parallel region

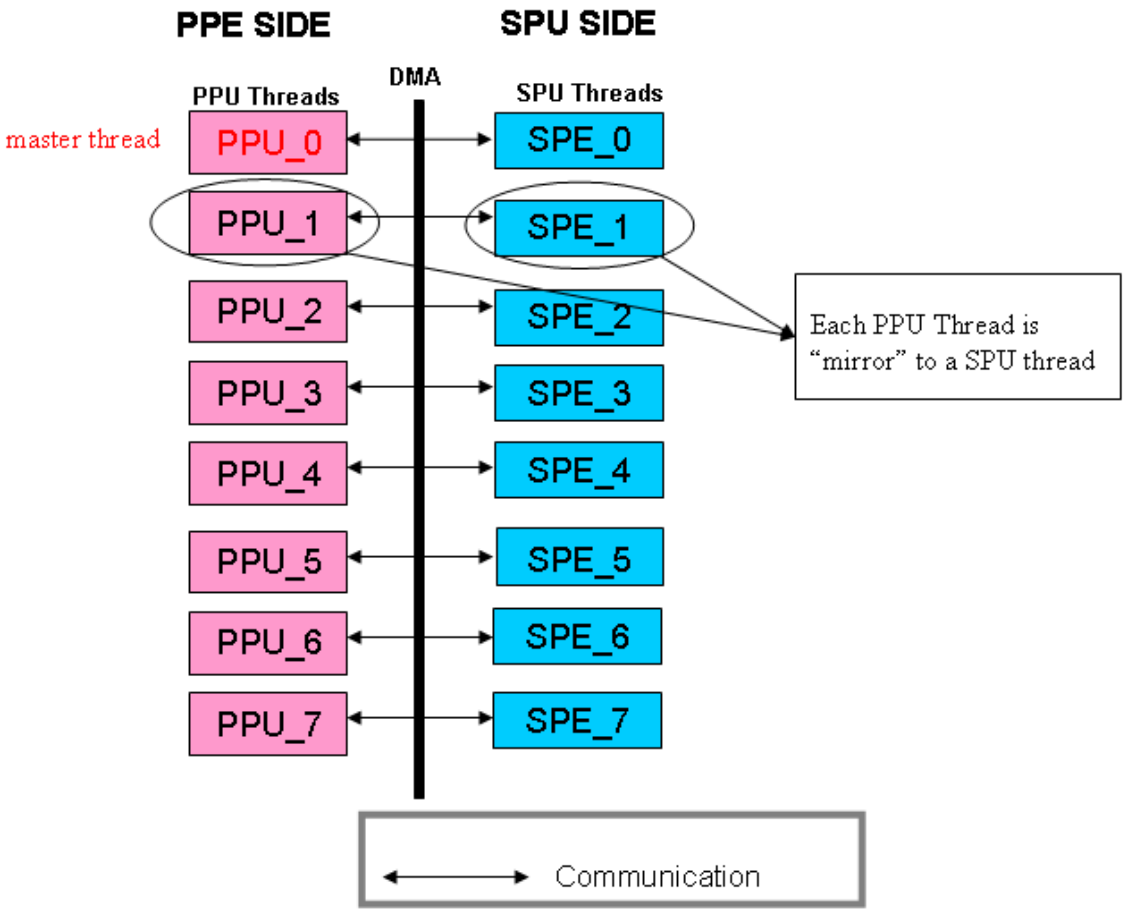

Figure 2.7: Thread Map in Execution Handler

has been outlined to a function named *main.omp_fn.0*. Two GOMP function call, *__builtin_GOMP_parallel_start* and *__builtin_GOMP_parallel_end*, will be inserted before and after the outlined function. They indicate fork and join point respectively. List 2.4 presents the pseudo code generated for SPU core. The GOMP function call *__builtin_GOMP_loop_dynamic_start* and *__builtin_GOMP_loop_dynamic_next* will help to get the lower and upper bound of iterations that this thread are working for. Moreover, the references of shared variables in line 24 26 needs to be transformed by the compiler to software cache function calls or explicit DMA instructions.

Listing 2.3: PPU Thread Code

```
1  main (argc, argv)
2  {
3    ...
4   .omp_data_o.14.chunk = 10;
5   .omp_data_o.14.c = &c;
6   .omp_data_o.14.b = &b;
7   .omp_data_o.14.a = &a;
8   __builtin_GOMP_parallel_start \
9              (main.omp_fn.0, &.omp_data_o.14, 0);
10   main.omp_fn.0 (&.omp_data_o.14);
11  __builtin_GOMP_parallel_end ();
12  return 0;
13   ...
14 }
```

Listing 2.4: SPU Thread Code

```
1  main.omp_fn.0 (.omp_data_i)
2  {
3   ...
4
5  <bb 2>:
```

```
6   D.2260 = __builtin_GOMP_loop_dynamic_start \
7     (0, 100, 1, .omp_data_i->chunk, &.istart0.15, &.iend0.16);
8   if (D.2260) goto <L19>; else goto <L7>;
9
10  <L7>:;
11  __builtin_GOMP_loop_end_nowait ();
12  return;
13
14  <L19>:;
15  i = .istart0.15;
16  D.2254 = .iend0.16;
17  pretmp.92 = .omp_data_i->a;
18  pretmp.93 = .omp_data_i->b;
19  pretmp.78 = .omp_data_i->c;
20  ivtmp.84 = 0;
21
22  <L5>:;
23  D.2358 = (int *) i + (int *) ivtmp.84;
24  MEM[base: (int *) pretmp.78, index: D.2358, step: 4B] = \
25    MEM[base: (int *) pretmp.93, index: D.2358, step: 4B] + \
26    MEM[base: (int *) pretmp.92, index: D.2358, step: 4B];
27  ivtmp.84 = ivtmp.84 + 1;
28  if (D.2254 > (int) (ivtmp.84 + (unsigned int) i)) \
29    goto <L5>; else goto <L6>;
30
31  <L6>:;
32  D.2262 = __builtin_GOMP_loop_dynamic_next \
33                          (&.istart0.15, &.iend0.16);
34  if (D.2262) goto <L19>; else goto <L7>;
35
36  ...
37  }
```

Figure 2.8 presents the whole communication process:

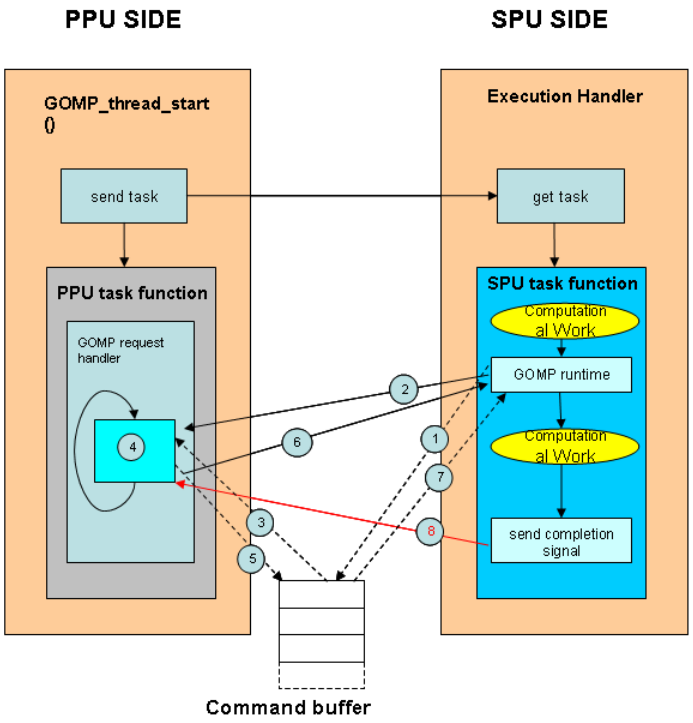

Figure 2.8: PPU and SPU thread communication in a runtime call

(1) When the SPU thread meets a GOMP service call like the one in line 6, instead of running the service routine locally at once, it will put the request in the command buffer with parameters.

(2) SPU thread sends a message including the function-shipping request to PPU thread and wait for the results.

(3) PPU thread receives the message, gets the command from the buffer and then reads the parameters.

(4) PPU thread calls the real GOMP runtime routine and generates the result. In this case, it is the lower and upper bound of the computation. The values refer to the variable .istart0.15 and .iend0.16 in the SPU Thread Code;

(5) PPU thread stores the results into command buffer.

(6) PPU thread sends a message back to SPU thread.

(7) SPU thread reads the result, does the computation and signals PPU thread again when the task processing ends.

This process will repeat until all computation work is done.

This design has several advantages. First of all, we can implement it based on original GOMP runtime library and very few changes need to be made because the PPU side runs a standard Linux kernel for PowerPC architecture. Thus, it should be able to run the original GOMP runtime; Secondly, threads on the SPU are very light-weighted as they have has passed most of the runtime calls to the PPU. Thirdly, SPU threads do not need to maintain runtime data structures as everything can be found on the PPU side. Last, we applied incremental strategy here. According to our design, the system may suffer extra communication overhead. While whenever we find an approach that executes a runtime function on the SPU with certain performance enhancement, we can move it to the SPU runtime, and still ship other runtime calls to the PPU side.

2.3.3 Software Cache

As we know, there is no cache in the SPU core, which presents us with both opportunities and challenges. For the regular memory references like memory accesses within a loop, we have the whole control of the LS and can figure out a static buffering strategy to reuse data. While for reference to shared memory, it would be very annoying for the user to insert DMA operations all the time, especially when he does not know the precise run-time behavior of the program. We can see from the example in List 2.2, the programmer need to insert a lot of DMA instructions with complicated parameters even in a small program. In this case, we will use a compiler-controlled software cache to load/store the data from/to system memory. In Line 24 of the List 2.4, the system memory access will be transformed to the service call to software cache library, which will handle the whole DMA transfer process. The main contributor in this part is Ge Gan and Andrew Russo.

The main specification of our software cache design is like the following. Currently, SPU software cache line is 128 bytes since it is the most efficient size for a DMA transfer. It contains up to 64 sets with 4-way set associative. The total size of SPU software cache is up to 32KB. The user can configure the number of sets at compile time. The cache line write policy is write-back and write-allocate. Currently, we only enable the D-Cache.

If the effective address that a load or store operation request can be found in the cache, it is a cache hit and the value in the cache is used and no DMA operation needed. Otherwise, it is a cache miss. For a miss, we allocate a new entry in the cache either by using an empty line or by replacing an existing line. Then, for a load, we issue a DMA_get operation to read the data from system memory. For a store, we write the data to the cache, and maintain dirty bits to record which byte is actually modified. Later, we write the data back to system memory using a DMA put operation, either when the cache line is evicted to make space for other data, or when a cache flush is invoked in the code based on OpenMP semantics.

The dirty bit vector can also help us to flush cache line. When a cache line is flushed into global memory, only the modified bytes are really written back by checking the dirty bits in the bit vector. An example of this process is shown in Figure 2.9. In this case, The bit vector tells that only the higher half byte of the cache block has been changed. As a result, only that part is written back when flushing. This feature can be also used to handle the false sharing and multiple writer problems.

2.3.4 Overlay and Partition Manager Runtime

Since each SPU only has a 256KB LS but may process big data stream, we rely on the technique of overlays. The overlay feature will allow the different sections to share address space and do not occupy the local storage at the same time. Thus, the pressure on local storage due to SPE code is greatly reduced. In chapter 3, we

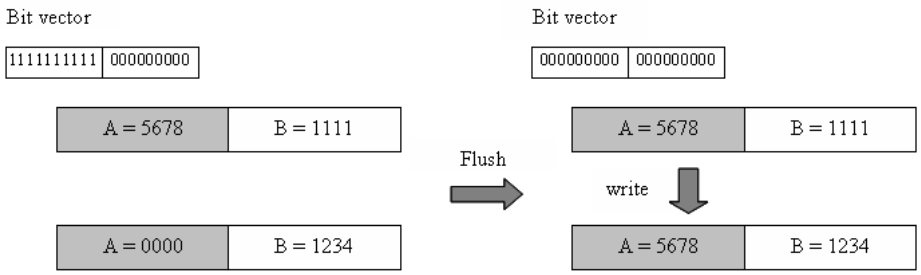

Figure 2.9: Flush in the Software Cache

will discuss how we may support overlay in single source compilation framework. On the other hand, traditionally the user has the responsibility to configure overlays in the link script, which appears to be a complicated job for them. Consequently we would also try to hide these details and design a new mechanism to invoke it. In chapter 4, we will present all the designs that target these issues.

Another problem is automatic code/data overlay loading and inter-partition calls. In our project, the partition manager runtime library will handle it with the help of assembler and linker. If the assembler gets a hint that a function call could be an inter-partition one, it will pass necessary partition information to the partition manager. Then at run time, the partition manager can maintain the call stack and decide whether to launch loading and complete the call finally. This part is mainly designed and implemented by Joseph B. Manzano of University of Delaware. Since it works so closely with the linker and assembler, more details will be explained in chapter 3 and chapter 4.

Chapter 3

SINGLE SOURCE COMPILATION

We have mentioned in the previous chapter that multiple ISA in heterogeneous cores are considered as a key challenge in programming for this type of architecture. To support OpenMP, we created a single source compilation framework to tackle this challenge. This chapter discusses the integration of this single source framework into the current tool chain as well as its interactions with the other Open Opell components such as compiler and partition manager.

This chapter is organized as follows: the overview of the design is presented in section 3.1 . We will abstract and formulate the problem of the integration into the tool chain part in this section. Then in section 3.2 we present the programmers' visible aspects of our framework. After that, section 3.3 presents our design of single source compilation and overlay framework in Open Opell system.

3.1 Design Overview

In the last chapter, we mentioned that the compiler will outline the parallel regions from the source code and run them on an SPU. However, these outlined functions that will run on SPU cores could be too big to fit into the limited SPU local storage. In this case, we have to split the whole SPU code section into partitions and overlay them. Each partition may contain one or several functions running on an SPU. To support this feature, our single source compilation framework need to solve two major problems:

The first problem is how to divide a program into multiple partitions. The compiler will assign the functions into different partitions based on user assigned pragmas (manual mode) or due to internal call-graph-based algorithms (automatic mode). For both modes, the goal is just to minimize the number of expensive inter-partition calls. Moreover, other components of the Open Opell system need to provide a clear and safe mechanism to get the partition information from the compiler, maintain partitions along the building process and finally, overlay them. In this chapter, we will focus on the first two phases. Specifically, we will present the "partition"' concept in section 3.1.2. In section 3.3.1, we will explain how we design a series of interfaces to attack this problem.

The second problem is how to handle different kinds of inter-partition calls. Since we introduce the code partitions, the handling of function calls on this platform is more complicated than on a conventional general purpose architecture. Two kinds of partitions could be involved in a function call. The first one is the "default" partition, which contains the library functions or the functions that always reside in SPU local storage. The second one is the "mobile" partition, which are assigned by compiler and share the run time address space with other "mobile" partitions. For a "mobile" partition, the address of a symbol cannot be determined at build time and it will be fixed at runtime (by the partition manager) with the support of tool chain.

Inside the tool chain, we use different relocation types to take care of different types of function calls. Since the caller and callee of a function call in our system could reside in either the default partition or a mobile partition, we need more types of relocation to handle each case. Another case arises when the function is called via pointer, then we do not even know the type of the callee in the case. In section 3.3.2, we will explain how we classify these scenarios and how we handle them.

3.1.1 Design Criteria

In our design, we will try to meet these criteria:

(1) Easy to use

Since the motivation for this work is to improve the programmability, usability is one of the key design criteria. To guide the tool chain behavior, user might use the new pragmas in the source file and new command line options for the compiler. However, we will show in this chapter that this control is very limited and all other complicated interactions between different components will be handled by the tool chain and invisible to the end user.

(2) Simple interface to maintain partitions

Just as it was mentioned in section 3.1.2, the partition plays a critical role in the Open Opell system, from the compiler to the linker. Moreover, the users are given the power to create partitions by themselves. Thus, we need to provide a simple interface so that all necessary information is easy to access whenever it is needed.

(3) Compatible to the GNU standard

All our work is based on the GNU-CC framework, so all extensions, including source code and binary file format, should be compatible to the standard GNU conventions. Therefore, maintenance, future update and extensions would be easier.

3.1.2 Partitions Handling: From the View of System Software

In the last chapter, we present our overall design on the single source compilation. In this chapter, we will focus on the toolchain's design, which is based on the concept of "partition". We will define "partition" and raise the related challenges in this section. Actually, the partition concept exists in most OpenMP systems. However, we need to design the new features to attack the new challenges in Open Opell system.

In a typical work flow of a OpenMP toolchain, the compiler separates out each code region in the source code that corresponds to an OpenMP parallel construct (including OpenMP *parallel regions*, *work-share loops* or *work-share sections*, and *single constructs*), and outlines it into a separate function. The outlined function in partition may take additional parameters such as the lower and the upper bounds of the loop iteration for parallel loops. The compiler inserts an OpenMP runtime library call into the parent function, and passes a pointer to the outlined function code into this runtime library function. During execution, the runtime function will elaborately invoke the outlined function. The compiler also inserts synchronization operations such as barriers when necessary. We would like to define "partition" as the basic unit that the framework can operate on. It will contain one or multiple outlined functions. At the same time, the concept of "partition" can also be extended to data. For example, in the mpeg2 benchmark, we can divide a picture into several data partitions. Each of these data partition can be run in parallel achieving an extra performance boost. A source file can hold multiple partitions, however, a partition can only reside in one source file.

Due to the heterogeneity of the CBE architecture, the partitions containing parallel tasks may execute on both the PPE and the SPEs. In this case, there could be one copy of the partition for the PPE architecture, and another one compiled for the SPE architecture. In Figure 3.1, we illustrate that the way that the threads access the partitions in Open Opell system are different from a traditional shared memory system. in the traditional share memory model like Figure 3.1 (a), these partitions reside in the same memory space and can access data directly. As a result, the tool chain usually put all code in the same section to make the tool chain implementation easier. However, in Open Opell as Figure 3.1 (b), just as we mentioned in chapter 2, each SPU only has limited local storage and the real application could be very huge. Therefore, code and data in one file should be

partitioned carefully so that all partitions created in SPU can fit the local storage. On the other hand, partitions may have to share the local storage with each other. Thus they may not be able to reside in local storage all at the same time. Figure 3.1 represents the difference.

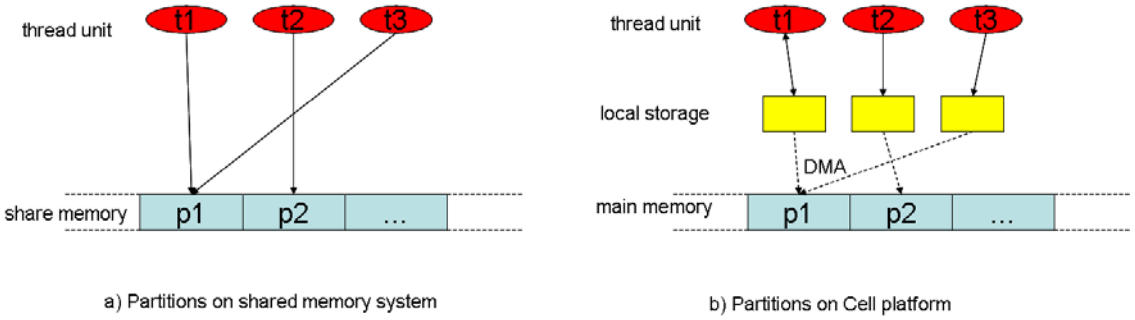

Figure 3.1: Difference of Partition Access

Based on these new challenges, three problems are raised:

(1) How to identify and maintain a partition?

(2) How to handle a call between partitions?

(3) How to guarantee that a partition which is needed at run time can be always found in the local storage?

To address the first problem, we designed a series of interfaces between toolchain components from the compiler to the linker and we extended the standard ELF file format to make sure that all necessary information is safe and easy to access in each phase. On the other hand, our partition overlay solution is proposed to address the second and third problem. For our design, code and data are partitioned. These partitions may have run time memory addresses which overlap. In this case, the partition manager library is responsible for loading the correct partitions during run time. Though it is an effective way, the configuration of overlay is difficult. To this end, we provide a "ghost" mechanism to invoke this feature. We will discuss this mechanism in detail in chapter 4 and in this chapter we will focus

on our tool chain support and assume that overlay partition configuration has been accomplished.

3.1.3 Single Source Compilation Framework

We have explained our design decisions in section 2.3.1. In this section we would like to provide a more detailed picture of the whole framework, which can be seen in detail by Figure 3.2.

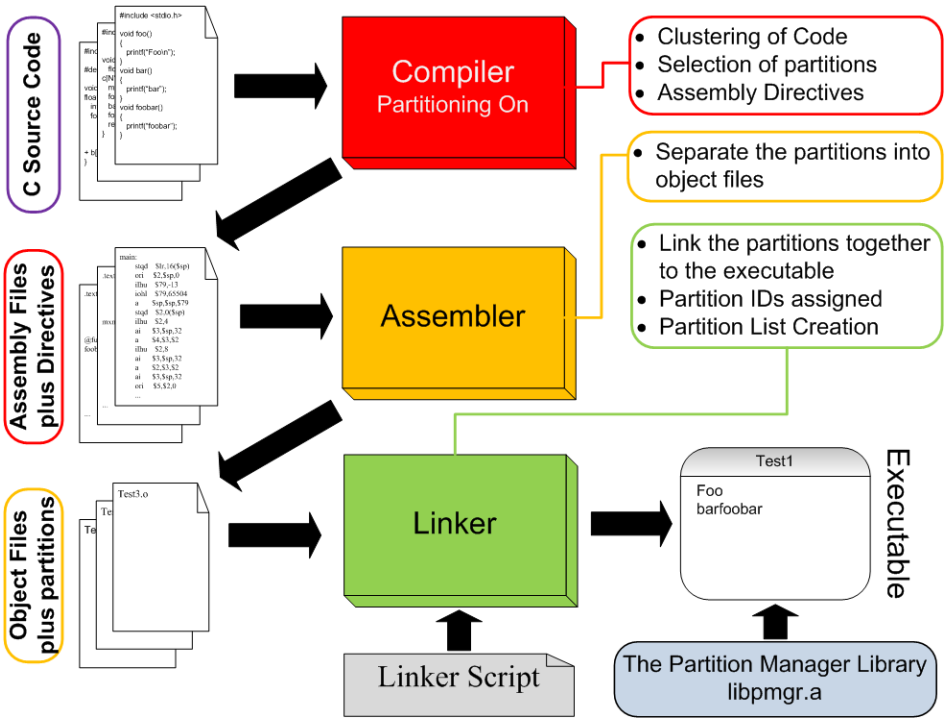

Figure 3.2: Overview of Single Source Compilation Framework

As represented by this figure, we can see that both the SPU part and PPU toolchains will share the same source code base. However, this code will be compiled by different compilers due to distinct ISA. Then, the generated object file will be linked with the necessary libraries and produce the executable file. The libraries that need for the framework include partition manager run time library and software cache library which will be linked against the SPU executable file and GOMP &

SPE library which will be linked against the PPU executable file. After that, the SPU executable file will be generated as a PPU library. Finally, the embedder will combine this library into final PPU executable file. From the viewpoint of the user, the whole process just looks like a single source compilation since both source and final executable file have only single copy. Since there was lack of support on SPU partition handling, our work will focus on the tool chain for the SPUs.

3.2 Compiler Design

This section will focus on explaining the interfaces and how they integrate all the tool chain components as a whole.

In general, the users would interact with the OpenMP compiler via two ways. First, the users would like to interact with the compiler via a couple of options, which will instruct the toolchain to produce the. At the same time, compared to a sequential programming model, OpenMP provides a set of hints to the compiler by pragmas to help parallelization. In our framework, we need to design and implement the new features to address the new challenges presents by this platform. Thus, a couple of new options and pragmas are designed. Although we should give the users the flexibility to control the whole compiling process, we must not overwhelm them with too many options. We will actually provide three options to the users. By using these options, the users can control whether the users want to partition code by themselves and the possible maximum size they need for automatic partitions.

- **-fcode-partition-auto:**

  For a benchmark running on the parallel platform, one of the most important problems is how to partition code and data. This could affect on the performance or even the correctness of parallel programs. In our design, we will let the users decide who will take charge of the partitioning. If they specify this option, it tells that it is the compiler that does this job. Compiler decides the

partition for a function based on a set of parameters or heuristics, such as estimated execution time and the size of each function. The goal is to minimize the number of inter-partition calls while still satisfying the size limit of code partition and the requirements of the programmers.

- **-fcode-partition-manu:**

  This is the complement option of -fcode-partition-auto. If the user make sure that he can produce even more efficient code than the compiler can without losing correctness, he may want to specify this option. In this case, programmers assign a code partition index to each function by using partition pragmas or partition clauses (in OpenMP parallel pragma). Functions that are not specified will be put into the default partition.

- **-fmax-insns=n:**

  As we know, in the explicit memory hierarchy architecture, we have complete control over local memory. Therefore, more flexible buffer strategies are allowed. Specifically, you can have one buffer with big size or more buffers with smaller size. The user can use this option to specify the size of the partition that can fit on the proposed buffer strategy.

Another way to interact with compiler is a new set of pragmas. Usually users will use them to guide the compiler optimization. In our platform, the compiler provides the pragmas to the users. With these, the users can implement their partition strategy freely.

- **#pragma partitions x func_1 [, func_2, ..., func_n]**

  Traditionally, A compilation unit would be a source file. However, in our framework, the unit is reduced to a partition. Intuitively, we need a mechanism to identify partitions. By using this pragma, the user will tell the compiler

what functions a partition consists of. In the parameter list, x is the id of this partition that allows it to differentiate from others, and the function name list just indicates the functions belong to this partition. Unsurprisingly, the partition id should be unique in this file to avoid confuse. If a function is not assigned to any partitions, we just put it into the default partition which implies that it will always reside in the local memory. Just as mentioned before, the option -fcode-partition-auto will disable all these pragmas.

- **#pragma keep_function func_1 [, func_2, ..., func_n]**

  In the compilation process, the scope of the compiler will be limited to the source file, so the compiler have no idea about the partition status of callee functions which do not reside in this file. On the other hand, this kind of information is very important for the compiler to generate assembly code. Consequently, we design this pragma to help the compiler to know somehow "global" information. Specifically, this pragma tells us the functions in the list may reside in a different partition or file.

3.3 Assembler

Assembler design is one of the key points in the whole framework. On the one hand, it needs to accept information about all partitions from the compiler. On the other hand, appropriate object file and relocations should be generated for the following linking process.

In our platform the assembler's job is even more complicated because of the two problems raised at the beginning of the section 3.1. In this section, we will present how the assembler interacts with the other components in the tool chain to solve these two problems. In the section 3.3.1, we will explain how to create a partition from the directives generated by the compiler. In the section 3.3.2, we will classify three types of special function call related to "mobile" partitions and give

the solution to each of them. Specifically, since the tool chain needs to interact with the run time library "partition manager" to solve the problems of partition loading and inter-partition calls. we need to figure out what kind of information partition manager needs and how we can transfer this information smoothly and efficiently.

3.3.1 .partition directive: Identifying a Partition

In this section, we present how the assembler interacts with the compiler to create partitions by handling the specific directive. Usually, all code belonging to one source file will be compiled into the same section named ".text". The advantage of maintaining a single code section is that it is easier to handle and debug. This is specially true for the DWARF debug format, which is used in GNU-CC , all debug information in a source file will be generated in one tree. Moreover, the section boundary would be impossible to determinate if we have multiple sections in one file. However, the pre-condition to use this design is that we assume all code will reside in a continuous memory block. Obviously, this is not suitable for our partitions situation since every partition will be operated independently at run time.

In general, our design treats these partitions as special sections. On the compiler side, compiler will parse the pragmas and generate .partition directives. Then, the assembler will process it like a traditional section. Afterwards, a flag named SHF_PARTITION is set. This flag will be used in the future when interacting with partition manager and linker. We will explain it in the following subsections. An example of generating partitions is presented in Figure 3.3.

3.3.2 .pmcall, .libcallee, .funcpointer directive: Inter-partitioned call

We will discuss the function call in this section. The traditional function call handling in the tool chain would be straight forward; in general, the assembler will find all function calls in which the callee's position is unknown and put a relocation

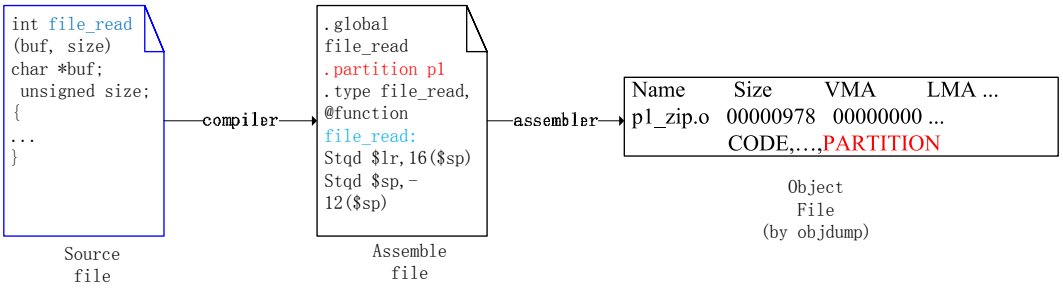

Figure 3.3: Process of Partition Generation

on that. Then the linker, who has global scope view, will figure out the callee's address and resolve the relocation. However, the partitions in our platform make things more complicated.

As we mentioned in section 3.1, the function could reside in default partition or mobile partition. If the second case, it is impossible for the linker and assembler to compute the actual address of a partition. Instead, the address will be decided during the runtime when the partition is loaded into SPE local storage. Thus when we find a inter-partition call, we have to invoke the partition manager and pass it enough information about this function call. Then a problem is raised: what kind of information is necessary and how could we pass it efficiently?

If a inter-partition call is initiated, partition manager will check whether the partition in which the callee resides is available in local memory. An efficient way to do this is to assign a unique partition id to all partitions in this program and then the partition manager can just compare the requested partition id with its history records it maintained. The linker will assign these id since it has the global scope. This whole process will be revisited in Chapter 4. Another issue is to locate the functions at run time. Though the run-time position of a function can not necessarily be decided in link time, the relative offset to the partition start point is fixed. Then we will use the current partition address plus this offset to locate a function. Therefore, the partition offset will replace the symbol's address. So far, we

need a symbol's value represents both partition id and partition offset. Fortunately, the local memory is only 256K big while SPU is 32-bit system, so we can simply encode partition id information into the high bits of a symbol. Figure 3.4 shows the design of symbol format.

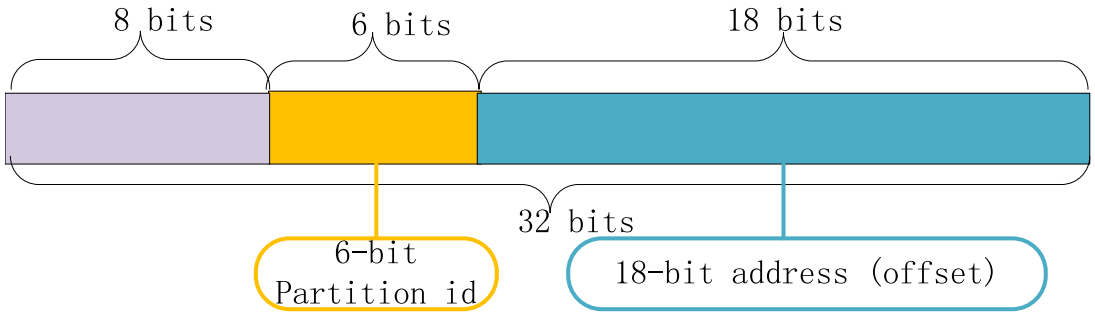

Figure 3.4: Format of Symbol in Partitions

According to the location of the caller and callee, we can classify function calls into four types:

1) It is a direct call. The callee could reside in mobile partitions.

2) It is a direct call. The callee reside in the default partition and the caller could reside in mobile partitions.

3) It is a indirect call via pointer. The callee could reside in mobile partitions.

4) Other normal calls.

For the fourth call, we can just use the traditional handling routines. In the rest of this section, we will explain our solutions to the other three types of inter-partition call.

- **.pmcall func: Inter-partition Call Class I**

  Compiler will generate this directive before the first kind of function calls and the "func" indicate the caller. In this case, both the caller and the callee could reside in mobile partitions. So we have to ask the partition manager to get the actual address of both and accomplish the function call at run time. At

the same time, the tool chain needs to provide necessary information to the partition manager at build time.

According to the agreement, three parameters, as caller partition id, the callee partition id and the offset of the callee symbol from the beginning of the partition, will be passed to the partition manager. The passing process will be via three registers which will be spilled to the stack and restored by the partition manager. Actually, we will generate relocations for these information references and the linker will resolved them with the appropriate values. Following this, the assembler will generate a specific instruction sequence. Finally, we will replace the current call instruction with a jump to the partition manager. This process can be represented in Figure 3.5.

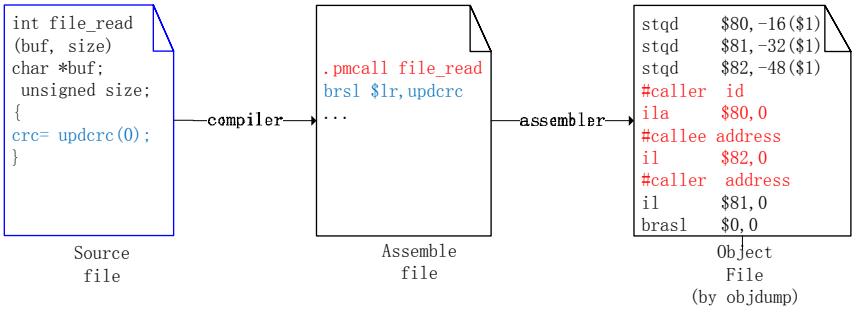

Figure 3.5: Processing .pmcall directive

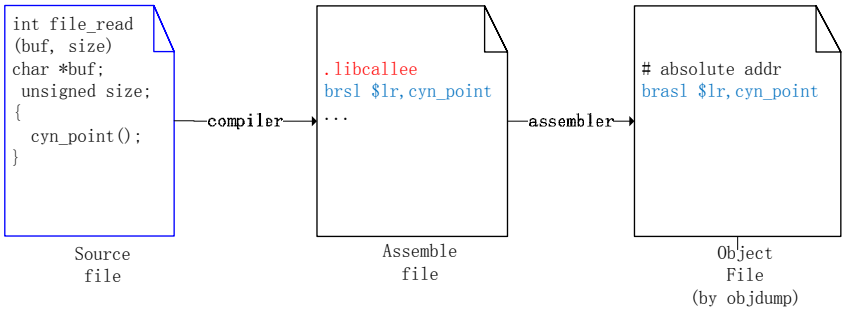

Figure 3.6: Processing .libcallee directive

- **.libcallee: Inter-partition Call Class II**

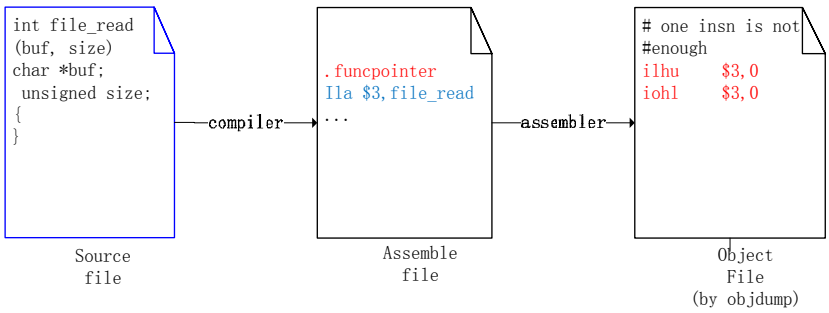

Figure 3.7: Processing .funcpoint directive

This directive is generated to help identifying the second kind of inter-partition calls. Though we know the callee resides in the default partition (or partition zero) and the address is fixed at build time, we still need to apply the specific handling.

Because the caller might reside in a "mobile" partition, the relative offset between the caller and callee is not a constant and the traditional pc-relative call instruction can not work in this scenario. On the other hand, a inter-partition call via partition manager is rather expensive, just like the first scenario, so we will try to avoid it. Our strategy is straight forward, we just process function call using absolute addressing instructions (e.g. brasl) instead of relative ones (e.g. brsl). An example is given in Figure 3.6.

- **.funcpoint: Inter-partition Call Class III**

  This directive can help the assembler to figure out the third kind of inter-partition calls by the compiler. It will be put just before a function pointer assignment. Now the inter-partition callee is a function pointer. In this scenario, we do not even know who the callee is, we have to invoke the partition manager to accomplish the call.

  The handling is similar to the first scenario. In this case, we will also put a .pmcall before the function call that use this function pointer. Again we

will pass three parameters described before. From the view of the assembler, the callee function address will be stored in a register and the call is via this register. Correspondingly, we will decode the fields of the value in this register and distribute them into the dedicated registers mentioned above. Be aware that one instruction is not enough to load both address and partition id so we have to use a pair of load instruction to do so. Figure 3.7 shows the handling of the function pointer assignment.

Chapter 4

GHOST OVERLAY

In the previous chapter, we have mentioned the overlay support for the Open Opell system. Although this is an effective technology that could be used to overcome the local storage limitation of SPUs, it does not completely free the users from the configuration tasks. In this chapter, we would like to propose our "ghost" mechanism which represents a novel way to use the overlay feature and address the problem mentioned above.

This chapter is organized as follows: In section 4.1, we review the overlay technology. The motivation of our "ghost" design will also be discussed. Then we report our design in section 4.2. Finally, the interaction with the run-time library partition manager has been described in section 4.3.

4.1 Overlay Overview

Actually, the overlay itself is not a new idea [35]. In order to reduce system costs, many applications use DSPs with limited on-chip memory has considered placing part of the program code and data off-chip. The overlay idea was raised in order to run these applications more efficiently by smartly making good use of internal memory. In the other words, if internal memory was not a constraint, overlays would become unnecessary.

In these systems, applications usually are desirable to run from on-chip memory. However, under this strategy, most applications are too big to fit the available on-chip memory. In this case, memory overlays provide a solution for this problem.

All program instructions and data are partitioned and stored in external memory. They will be loaded when they are required for program execution. In this case, two separate partitions of code or data can share the same internal memory space while occupying different external memory space. At the same time, it implies that only one of these sections can reside in internal memory at a time. We can take overlay as a many-to-one memory mapping system. The partitions are referred as memory overlays and the routines that load and execute them are overlay managers.

There are two kinds of overlay managers: manual and automatic. In the first class, it is the user's responsibility to figure out statically when and which memory overlays will be loaded. Moreover, it is the users' responsibility to write the code such that the memory is properly initialized. And in the second class, the user just point out where he might want to use overlays and then the system software tools, like a library, will take care of it. In both cases, the user needs to know the overlay structure to invoke the overlay manager at the right time. That is a non-trivial effort even for an experienced programmer because the structure could be very complicated. The overlay manager can also handle more advanced functionality, such as checking if the requested overlay is already available in local memory, overlapping function execution and overlay load, and tracking recursive overlay function calls.

The relationship between overlays is more difficult than sequential sections since we need to specify dependence and sharing between them. Figure 4.1 provides an example of a complicated overlay. In this example, the section ovl1, ovl2 and ovl3 will share the same memory address. Therefore, the start address of section ovl4 depends on the biggest section of ovl1, ovl2 and ovl3. On the other hand, the section ovl5 and ovl6 also depends on ovl4 and they share the same address, too. To handle this complex case, the linker designers have to provide complex commands in the linker scripts. Furthermore, the users have to be very careful when using them.

At the same time, the linking time could also increase dramatically to process them.

Under Open Opell system, we observe that these overlay partitions actually only maintain rather simple relationships. Based on this, we designed the "ghost" mechanism to invoke overlays and hide these details to users.

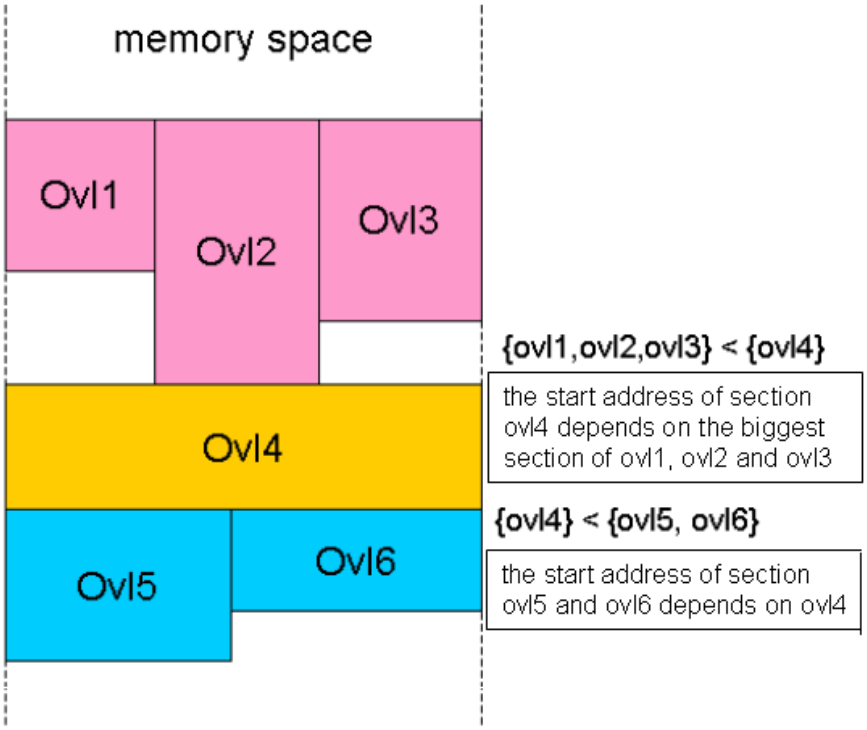

Figure 4.1: An example of complicated Overlay

4.2 Ghost Design

We can see from the last section that compared to normal sections, it is more difficult to describe the relationship between overlays. Usually, users have to be familiar with the language in the linker script file (LDF) and write down a lot of code. While according to the design criteria mentioned in section 3.1.1, in our design

we want to hide all these details to the user. Furthermore, we do not even want the user to see the overlay structure in the tool chain flow. From the programmer's view, they only know that the code and data can be partitioned into parts and they might run in parallel – it has nothing to do with low-level design like overlays.

The first design point of this feature is that, instead of bothering the users to specify all overlays, we just create a general overlay rules. In this case, we invoke the overlay partitions via the "ghost" mechanism and the users do not even know the overlay partitions. Again, it is based on the observation we mentioned in the last section: these overlay partitions actually only maintain rather simple relationships and our general rule can cover it. We will explain this rule in details in section 4.2.2. The second point is to create a runtime data structure to hold all partitions' information. By doing this, the partition manager library can access them and accomplish overlay partition load and replacement in an efficient way. The design will be presented in section 4.2.3 Specifically, our design will consist of the following parts:

4.2.1 Overlay Preparation

In our design, all partitions are potentially flexible and could be loaded and replaced by the partition manager. It implies that all of them have the possibility to "overlay" with others. Therefore we do not have to configure the link script to specify them. Instead, just as we mentioned in the last chapter, all these partitions, which are identified with the .partition directive will have a special flag, SHF_PARTITION, set. Thanks to this, the linker can identify and handle these "implicit" overlay sections.

4.2.2 Overlay Creating

The mapping from sections in each object file to segments in executable file is done by the linker. According to the general description in section 4.1, linker need

to figure out two addresses: the runtime address and the loadtime address.

In reality, the relationship between partitions is not as complicated as in traditional overlay sections. We call the memory buffer that is designated to be shared by overlay partitions as "partition buffer". We also enable multiple buffers strategy, which allows different partition buffer and the partition could be put in any one of them. For the runtime address side, the partition manager could only figure out the actual location of function at run time. According to Figure 4.2, the actual partition runtime address will start from 0x80, which is named by "Partition Buffer"'. On the other hand, since we can not figure out it during linker time, we will just look all partitions as if their runtime address start from zero. This refers to the partition virtual runtime address in Figure 4.2. From the view of linker, we can consider that all partitions share the same run time address.

For the loadtime address side, at the beginning of program run, all partitions just reside in PPU's main memory. To avoid tricky memory address transformation, we will not specify loadtime address at link time. The partition manager can compute it by offsetting from image start at run time. Thus, in this field, we just assign them the address greater than 256K to avoid confusion with other useful information. This refers to the partition virtual load time address in Figure 4.2.

4.2.3 Overlay Maintenance

Overlay Maintenance

Since the responsibility of computing runtime and loadtime addresses for partitions has been assigned to the partition manager, we need maintain a table, called partition list, to store partition information and aid the partition manager to do address computation. In this partition list, each partition will occupy an entry with 128 bits and indexed by a unique partition id assigned by linker. Just like the address field of symbols in partitions, these partition ids would also be encoded in the section's address field. For each entry, the first 64 bit would indicate the

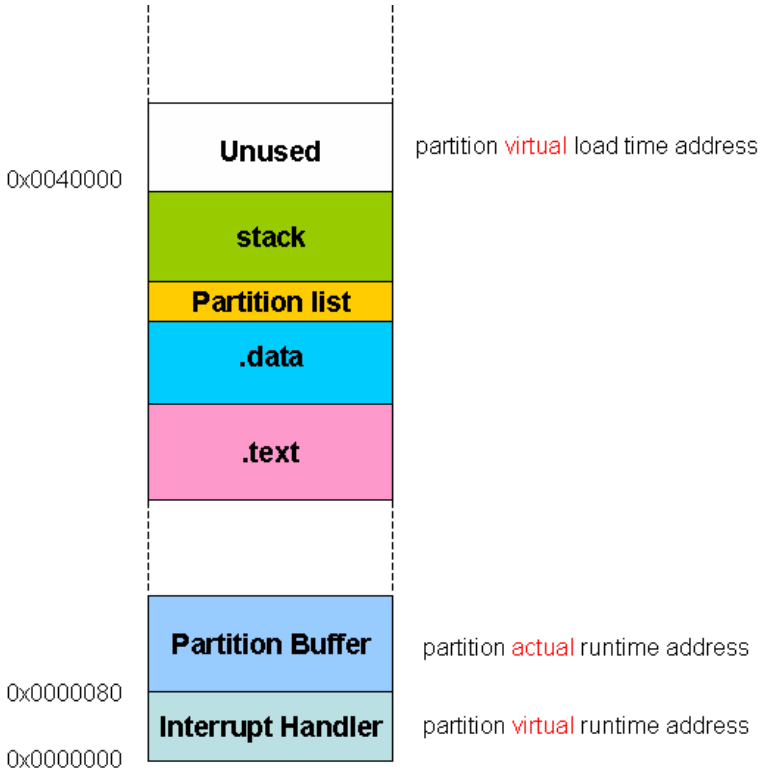

Figure 4.2: Memory Space in SPU binary

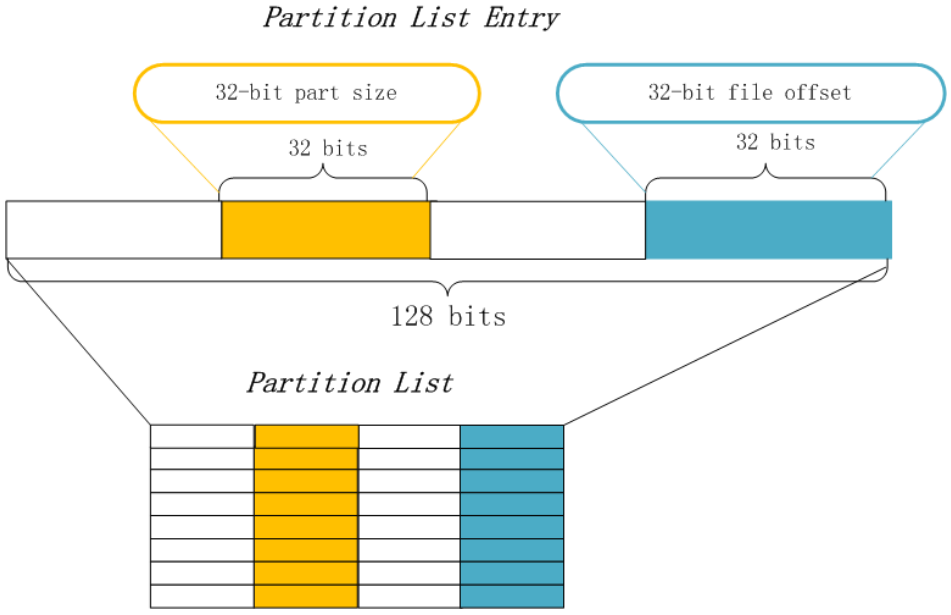

Figure 4.3: Partition List Format

partition's offset from the image starting point and the second 64 bit would be the size of the given partition block. The design is shown in Figure 4.3. Each partition will occupy an entry in this list, which include the partition size in high 64 bit and image file offset in low 64 bit. The partitions are assigned in the order of partition id. Therefore, the partition manager library can look up and access the appropriate partition easily. In the next section we can go though how the partition manager can use this information to locate the desired section in the PPU's main memory.

Our "ghost" overlay section design is supposed to be highly flexible so a set of state variables are also provided to the partition manager. For example, partition manager might want to inquiry the number of the partitions. Using these information, partition manager can try different buffer strategies or boundary checks. All these special variables will be discussed in the next chapter.

4.3 Interaction with the Partition Manager

In this section we will briefly introduce how overlay manager deal with a inter-partition call with the help of the tool chain. First of all, assembler will spill registers that will be used as argument register and then prepare all the parameters like the caller partition id, the callee partition id and the offset of the function to the beginning of the associated section. The actual values would be resolved by the linker. It is partition manager's responsibility to restore the argument register from stack. The reason is that we want to leave a clean stack context before we start the real call. Secondly, partition manager will check whether loading from main memory is necessary. If callee happens to reside in the partition zero or active ones according to the history record, we do not bother issuing the time-consuming DMA operation. Thirdly, if the loading is really necessary, the partition manager will have to use a DMA transaction. In this DMA transaction, the source would be located by the image start point in main memory plus the file offset of the partition and the destination will be decided by the state variable _buffer which is specified by linker

and NUM_BUFFERS which is specified by partition manager itself. Once we have it, it is rather easy to use function offset to finally find the exact address of callee. Finally, the partition manager will make a normal call using the symbol address as a pointer.

Chapter 5

IMPLEMENTATION AND EVALUATION

In this chapter, we report the implementation and evaluation of our design in the CBE platform. This chapter is organized as follows: all implementation details has been presented in section 5.1. Then a brief introduction of the experiment platform that we used is given in section 5.2. Detailed evaluation of both robustness and performance will also be presented in this section.

5.1 Implementation

Our implementation is based on IBM Cell SDK version 1.1 [36]. This SDK provides a complete Cell Broadband Engine development environment, including Linux kernel for Cell Broadband Engine processors, Linux support libraries, tool chains, XL C/C++ compiler, system simulator, source code for libraries and samples, and a new, fully-integrated installation. As it is released by the official designer of Cell processor, it has good reputation on both robustness and efficiency. However, the single source compilation feature is not released so we developed new Open Opell features based on it. In this thesis, we will focus on the implementation of assembler and linker.

5.1.1 Assembler Implementation

There are three key functionalities we need to implement in the assembler part: accepting new directives, generating extra instruction sequences and generating new relocation types. Details will be presented as follows:

(1) Accepting New Directives

To add a new directive that can be recognized by the assembler, we need to add these directive names to the "pseudo_table" list. Specifically, if this directive is about an extension of the ELF format, it should be added to the list named "elf_pseudo_table". Otherwise, it should be put into the one named "md_pseudo_table". In this table, each new directive will be mapped to a specific function to accomplish the actual action.

- **.partition: obj_elf_partition()**

  When a .partition directive is encountered, the assembler will call this function to generate a partition in the final object file. In general, it is just a special ".text" section. Besides the default attribute like SHT_PROGBITS, SHF_ALLOC, SHF_EXECINSTR, we will also set SHF_PARTITION to indicate that it is a partition and this flag will be kept through the whole compilation process. In this way, other phases or linker can identify a partition easily. Just like a section, it implies the end of the previous section or partition and its scope will include everything from this directive to the next .section or .partition directive.

- **.pmcall: cross_partition_call()**

  In GNU assembler code base, the method that passes information from lexical analysis to syntax action is to create and maintain a pool of global variables. It is simple and effective. In this case, what we need to do is to figure out the place that we need to insert instruction sequence and give a hint to the object code generator, a global variable "cross_partition_caller" is created and work in this way: each time we get a new .pmcall directive, we will record the caller's information, such as its name, in this variable. On the other hand, once this directive is processed successfully, this variable needs to be reset.

Then, it also plays a role that indicates whether we have this kind of directive currently.

- **.funcpoint: function_pointer()**

  The global variable "function_pointer_call" will be used as an indicator to communicate between the lexical analysis and syntax action phase.

- **.libcallee: library_callee()**

  The global variable "lib_callee" will also be used as an indicator to communicate between the lexical analysis and syntax action phase.

(2) Generating Extra Instruction Sequence

For each directive that is related to the inter-partition call, we need to generate an instruction sequence to prepare the parameters and make the jump to the partition manager. The actual sequence for each directive has been presented in section 3.3.2. In our implementation, we just watch on the indicators generated by lexical analyzer and make these instructions in function "md_assemble". In general, the caller information, if needed, will come from the indicators and all callee information is in the assemble instruction we are processing.

(3) Generating New Relocation Types

When preparing the arguments, we need to get the caller's partition id and assign it to one argument register, so we need a new relocation type to indicate the linker to fill appropriate id value instead of address value. At the same time, the flexibility of the standard ELF format makes this possible. The first step is to add a new entry "BFD_RELOC_SPU_IMM6PID" in relocation type list. Then we need to attach the necessary attributes for it in "reloc_howto_struct" list. In this list, each relocation type will occupy an entry which includes information about the type, size, position in the instruction and the function that handles this relocation. These

attributes should agree with the format in Figure 3.4. Finally, when resolving this relocation, linker will follow this "howto" to fill valid values in valid positions.

5.1.2 Linker Implementation

We also need to provide two features in the linker to support our "ghost" overlay design: creating the "ghost" overlay and generating the partition list and the state variables. We will present them as follows:

(1) Creating "Ghost" Overlay

Originally, the linker maintains a section list to help mapping from input sections to output segments in an executable file. The mapping rules are described by the linker description file (LDF). However, in our design, these partitions are just "ghosts" so there are no rules in the LDF about them. Then, as the first step we need to maintain a partition list for the linker to do partition mapping. We implement a pointer field in each section and build a local link list to cover all partitions. Finally, in our design, each partitions will be mapped to a segment with the same name.

Unlike traditional overlay sections, runtime addresses and loadtime addresses for partitions are not that important. However, to align with original error check mechanism in GNU-base code, we should notify that all partitions can share runtime address but they must reside in different loadtime address. According to the design, we just assign runtime address from 0x0 and loadtime address from 0x400000.

(2) Generating Partition List and State Variables

All these helper structures will be used by partition manager at run time. The partition list will reside in the end of .data section to make sure that it will not be kicked out of local storage. Our strategy is to generate them in the last phase of linking where all sections are mapped to the segments in executable files, and then these extra data will not affect original data section. For the partition list, since we have maintained all partition information through linking, we just go though all

partitions and generate the table in the format as Figure 4.3. We will also generate state variables as:

- **_buffer:** This variable is defined as the beginning of buffer in local storage. As we know, it is the base value for partition manager to compute DMA transfer destination and locate function in active partition.
- **_buffer_size:** This variable is defined as the size of the buffer. Usually it will be used to do boundary check.
- **__partition_list:** This variable is defined as the beginning of partition list, partition manager can index each partition based on it.
- **_table_size:** This variable is defined as the size of the partition list. It also implies the maximum number of partitions in the whole program.

5.2 Experiment Result

5.2.1 Experiment Platform

In this study, we use a Playstation3 (trademark of Sony Inc.) as our experimental platform since it is easy for us to find and afford it. It contains a typical Cell processor with 256MB memory. Though Playstation3 is well known as a popular gaming console, Sony proclaimed that it would be more than that. Currently a Linux distribution and various developer tools are available for the Playstation3. Yellow Dog Linux is the first officially supported flavor of the operation system to run on the Playstation3. We installed it as the operation system on our experiment platform. All our experiments used one Cell chip: one PPE and six SPEs [1]. Our tool chain will run in the PPU processor. This dual issue processor runs in 3.2 Ghz. It can support two threads via SMT and 512 KB second-level cache.

[1] Two SPEs are reserved by the operation system

5.2.2 Experiment Summary

In our experiment, we will measure our system by two criteria: robustness and efficiency. First of all, we want to prove that the applications that are built by the Open Opell system can produce the correct results so that programmers will be confident to develop their real world applications on it. Specifically, we have a test case pool proposed by the potential customer of our system. All of them are non-trivial test cases from real world application. On the other hand, the build time is also very important since it will affect the programmers' productivity directly. We will also use the same benchmarks to measure whether the overhead introduced by our new features is acceptable. We will not present the application performance in this thesis. The reason is that in the modern software development system, most of the optimizations are applied in the compiler or runtime. Therefore, the tool chain part like the assembler and linker are not the critical factor to affect the generated code performance.

5.2.3 Robustness Evaluation

One of the Open Opell system's goal is to provide tool chain of high quality. As a result, besides the unit test cases covering each type of pragma and intra-partition call, we also put a group of real applications proposed by our potential customer in our test case pool. They are ranging from DSP applcations to scientific computation. Table 5.1 lists these test cases and brief description.

In the experimental platform, all these application pass our test under our framework. This proves that our system is quite robust.

5.2.4 Performance Evaluation

In this section, we will test the efficiency of our system. We will use our code base IBM Cell SDK 1.1 as reference. Our experiment methodology is to compare the assembler and linker time of the application in Table 5.1 under IBM SDK and

Application Name	Description
DSP	DSP application kernels
Jacobi	Jacobi method
Laplace	Laplace transform
MD	simple molecular dynamic
MGRID	Multi-Grid solve
GZIP	File Compression and DeCompression

Table 5.1: Test Case Pool

Application Name	assemble time	reference time	slowdown
DSP	0.02	0.013	20.00%
Jacobi	0.016	0.016	0.00%
Laplace	0.013	0.013	0.00%
MD	0.013	0.013	0.00%
MGRID	0.016	0.013	18.75%
GZIP	0.153	0.13	15.03%

Table 5.2: Assemble Time Evaluation

Open Opell. These applications built by the two system have exactly the same functionality and input, except we use a couple of directives in Open Opell one so that we can measure the overhead of our new features, which is indicated as "slowdown" in the table. We run all test cases three times and record the average time in the second and third column for Open Opell and our reference respectively. The experiment results for the assembler and linker performance are presented in Table 5.2 and Table 5.3:

According to the result, for assemble time, the slowdown is no more than 20% and 8.96% on average, for link time, the maximum slowdown is 28.13% and 14.41% on average. Compared to the advantage of using OpenMP programming model, this modest performance degradation is rather acceptable to the users[37], especially considering that we have the extra libraries like partition manager. Actually, in

Application Name	link time	reference time	slowdown
DSP	0.04	0.03	25.00%
Jacobi	0.05	0.05	0.00%
Laplace	0.06	0.05	16.67%
MD	0.06	0.05	16.67%
MGRID	0.06	0.06	0.00%
GZIP	0.32	0.23	28.13%

Table 5.3: Link Time Evaluation

our design and implementation, the extra overhead will only be produced when the programmer apply the new pragmas in the source code. Another observation is that though our test cases are from the real world, the assembler and linker time is very short. In the future work, we will do the experiments on bigger applications to prove the efficiency of our system. Overall, we can conclude that our design and implementation is an efficient solution to support OpenMP on Cell platform.

Chapter 6

RELATED WORK

We will briefly summarize published research on Cell. The first aspect is about the processor and architecture research. Kistler et. al [16] studied the performance of the Cell's on-chip interconnection network. Moreover, the latencies of DMA operations for different workload characteristics are determined using a series of microbenchmarks. They also investigate the system behavior under different patterns of communication between local storage and main memory. A framework to analyze and predict performance on Cell platform was proposed by Williams et. al [38] and several application kernels including matrix multiplication, stencil computations and FFTs are used to validate this model. Based on that, some suggestions for the micro-architecture are also proposed.

The second aspect is about the compiler/runtime environment. Eichenberger et. al [22] presented several optimization techniques on the compiler to extract high performance out of the SPE. They covered both thread-level parallelism and SIMD parallelization. Specifically, they discussed the techniques like compiler-assisted memory alignment, SIMD code generation, branch prediction and compiler-controlled software caching. All these techniques are also implemented in IBM XL compiler and the performance improvement is impressive.

Filip Blagojevic et. al [39] presented a runtime system and scheduling policies that exploit polymorphic (task and loop-level) parallelism on Cell and a event-driven

multithreading execution engine. A feedback-guided scheduling policy for dynamically triggering and throttling loop-level parallelism across SPEs is also proposed in this paper.

The third aspect is about the benchmarks on the Cell platform. Benthin et. al [40] present an implementation of ray-tracing algorithms on Cell. Hjelte [17] presents an implementation of a smooth particle hydrodynamics simulation. D.Bader et. al [41] study the case list ranking to analyze of irregular algorithms on the Cell processor. This group [42] also tries to apply the fastest fourier transform on Cell. O. Villa [43] raises several challenges in mapping graph exploration algorithms on Cell platform. F. Blagojevic [44] presents the RAxML, a provably efficient, hill climbing algorithm for computing phylogenetic trees, on the CBE processor. F. Petrini et. al [45] explained their experience on optimizing sweep3d benchmark. S. Alam et. al [46] applied their strategy on covariance matrix creation routine in hyperspectral imaging.

The fourth aspect is about the programming model.

J. A. Kahle et. al. [20] review the new challenges introduced by SPE design and describe a number of proposed programming models, such as function offload model, device extension model and etc. Michael Gschwind [21] discuss exploiting multi-level parallelism using a variety of programming models. They also describe new programming models for heterogeneous cores based on a heterogeneous threading model. L. Cico et. al. [18] briefly describe the performance of signal processing algorithms they have optimized for the CBE and discusses their preferred programming model for achieving high performance. P. Bellens et. al. [47] present Cell Superscalar (CellSs) framework to addresses the automatic exploitation of the functional parallelism of a sequential program through the different processing elements of the Cell BE architecture. This programming model allows the programmers to

write sequential applications and the framework is able to exploit the existing concurrency and to use the different components of the Cell BE (PPE and SPEs) by means of a automatic parallelization at execution time. Kevin O'Brien et. al [48] also present how to support OpenMP on the Cell processor. They support OpenMP by orchestrating compiler transformations with a runtime library that is tailored to the Cell processor. A series of test cases demonstrate that their approach can achieve performance similar to that of manually written and optimized code. Wei et. al [49] also presents the OpenMP programming model on Cell and discuss the effective mapping strategies to conduct the thread creating and data handling. Jun Sung Park et. al [50] have implemented the OpenMP workshares on Cell Broadband Engine. Joseph et. al [51] present part of our work in supporting OpenMP and discuss the code layout optimization in SPU.

Chapter 7

CONCLUSION AND FUTURE WORK

In this thesis, we present the Open Opell system, a co-design of all system software components like compiler, assembler, linker and run-time library to support OpenMP single source compilation on Cell platform. We also propose a "ghost" overlay feature that can help users to address local storage limitation and DMA transfer problem. We have implemented our design based on GNU tool chain and a series of experiments are done on the real Cell processor. According to the experiment result, we can make the conclusion:

- We can develop a toolchain framework to support OpenMP single source compilation on a heterogeneous system like Cell. While OpenMP is target for SMP architecture, users who program on heterogeneous system can still use this API without the extra effort. Though we only implement it on the Cell platform, we believe these designs can be generalized and apply in other heterogeneous systems.

- Our design and implementation is quite robust and efficient. A series of non-trivial benchmark have been used in our test bed. All test cases are proved to run correctly in our framework. On the other hand, both assembler and linker suffer modest performance loss. According to the result, for assemble time, the slowdown is no more than 20% and 8.96% average, for link time, they are 28.13% and 14.41% for maximum and average respectively.

In future work, we intend to incorporate more optimization in our framework to improve benchmark's performance. Specifically, we will optimize automatic partition generation in the compiler to extract more parallelism from certain applications. We also intend to study the code/data overlay buffer strategy to alleviate the expensive DMA operation overhead.

BIBLIOGRAPHY

[1] J. Kahle, "The cell processor architecture," in *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2005, p. 3.

[2] M. Gschwind, "Chip multiprocessing and the cell broadband engine," in *CF '06: Proceedings of the 3rd conference on Computing frontiers*. New York, NY, USA: ACM, 2006, pp. 1–8.

[3] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *Computer Architecture News*, vol. 23, no. 1, pp. 20–24, 1995.

[4] L. Spracklen and S. G. Abraham, "Chip multithreading: Opportunities and challenges," in *the 11th International Symposium on High-Performance Computer Architecture (HPCA'05)*, 2005.

[5] L. Hammond, B. A. Nayfeh, and K. Olukotun, "A single-chip multiprocessor," *Computer*, vol. 30, no. 9, pp. 79–85, 1997.

[6] IBM, "The CELL project at IBM research," http://www.research.ibm.com/cell/.

[7] G. Almási, c. Calin Cas J. G. C. nos, M. Denneau, D. Lieber, J. E. Moreira, and H. S. Warren, Jr., "Dissecting cyclops: a detailed analysis of a multithreaded architecture," *ACM SIGARCH Computer Architecture News*, vol. 31, no. 1, pp. 26–38, 2003.

[8] Intel News Release, "Intel develops tera-scale research chips," Sept. 2006.

[9] DRC, "DRC product overview," http://www.drccomputer.com/drc/products.html.

[10] D. Stasiak, R. Chaudhry, D. Cox, S. Posluszny, J. Warnock, S. Weitzel, D. Wendel, and M. Wang, "Cell processor low-power design methodology," *IEEE Micro*, vol. 25, no. 6, pp. 71–78, 2005.

[11] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick, "The potential of the cell processor for scientific computing," in *CF '06: Proceedings of the 3rd conference on Computing frontiers*. New York, NY, USA: ACM, 2006, pp. 9–20.

[12] T. Saidani, S. Piskorski, L. Lacassagne, and S. Bouaziz, "Parallelization schemes for memory optimization on the cell processor: a case study of image processing algorithm," in *MEDEA '07: Proceedings of the 2007 workshop on MEmory performance*. New York, NY, USA: ACM, 2007, pp. 9–16.

[13] Thomas Chen and Ram Raghavan and Jason Dale and Eiji Iwata, "Cell broadband engine architecture and its first implementation," *IBM developerWorks*, Nov. 2005.

[14] Barry Minor and Gordon Fossum and Van To, "Terrain renderin engine (tre)," http://www. research.ibm.com/cell/whitepapers/TRE.pdf, May 2005.

[15] "Ibm to build world's first cell broadband engine based supercomputer," http://www-03.ibm.com/press/us/en/pressrelease/20210.wss.

[16] M. Kistler and M. Perrone and and F. Petrini, "Cell multiprocessor communication net-work: Built for speed," *IEEE Micro*, vol. 26(3), no. 3, May/June 2006.

[17] N. Hjelte., "Smoothed particle hydrodynamics on the cellbrodband engine," in *Masters thesis, Department of Computer Science, Umea University, Sweden*, June 2006.

[18] R. C. L. Cico and J. Greene, "Performance and programmability of the ibm/-sony/toshiba cell broadband engine processor, White Paper," 2002.

[19] "Sony/toshiba/ibm workshop on software and applications for the cell/b.e. processor, georgia tech," http://sti.cc.gatech.edu/program.html, Atlanta, GA.

[20] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the cell multiprocessor," *IBM J. Res. Dev.*, vol. 49, no. 4/5, pp. 589–604, 2005.

[21] M. Gschwind, "The cell broadband engine: Exploiting multiple levels of parallelism in a chip multiprocessor," *International Journal of Parallel Programming*, vol. Volume 35, no. 3, pp. 233–262, 2007.

[22] A. E. E. et al., "Optimizing compiler for a cell processor," in *Parallel Architectures and Compilation Techniques (PACT)*, September 2005.

[23] OpenMP Architecture Review Board, "OpenMP FORTRAN application program interface," OpenMP Architecture Review Board, Tech. Rep. 2.0, Nov. 2000, in http://www.openmp.org/specs.

[24] ——, "OpenMP FORTRAN application program interface," OpenMP Architecture Review Board, Tech. Rep. 2.0, Mar. 2002, in http://www.openmp.org/specs.

[25] B. Gaster and C. Bradley, "Exploiting loop-level parallelism for simd arrays using openmp," in *International Workshop on OpenMP (IWOMP), 2007*, 2007.

[26] C. Terboven, D. an Mey, and S. Sarholz, "Openmp on multicore architectures," in *International Workshop on OpenMP (IWOMP), 2007*, 2007.

[27] K. Hoang, J. Tao, and W. Karl, "Cmp cache architecture and the openmp performance," in *International Workshop on OpenMP (IWOMP), 2007*, 2007.

[28] P. E. Hadjidoukas and L. Amsaleg, "Parallelization of a hierarchical data clustering algorithm using openmp," in *International Workshop on OpenMP (IWOMP), 2006*, 2006.

[29] J. P. Hoeflinger, "Programming with cluster openmp," in *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2007, pp. 270–270.

[30] W. Zhu, J. del Cuvillo, , and G. R. Gao, "Performance characteristics of openmp language constructs on a many-core-on-a-chip architecture," in *International Workshop on OpenMP (IWOMP), 2006*, 2006.

[31] A. K. Morris, A. Malony, and S. S. Shende, "Supporting nested openmp parallelism in the tau performance system," in *International Workshop on OpenMP (IWOMP), 2006*, 2006.

[32] J. del Cuvillo, W. Zhu, and G. R. Gao, "Landing openmp on cyclops-64: An efficient mapping of openmp to a many-core system-on-a-chip," in *CF '06: Proceedings of the 3rd conference on Computing frontiers*. ACM, 2006.

[33] S. Karlsson and M. Brorsson, "A free openmp compiler and run-time library infrastructure for research on shared memory parallel computing," in *in Proceedings of The 16th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2004)*, November 2004.

[34] "Omni openmp compiler project," http://phase.hpcc.jp/Omni/home.html.

[35] "Overlays {GNU Linker Manual}," http://www.gnu.org/software/binutils/manual/ld-2.9.1/html_node/ld_22.html.

[36] "Ibm cell broadband engine software development kit," http://www.alphaworks.ibm.com/tech/cellsw.

[37] K. K. O'Brien, "Re: A question..." private message, 3 Jan 2008.

[38] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick, "The potentinal of the cell processor for scientific computing," in *ACM International Conference on Computing Frontiers*, May 3-6 2006.

[39] F. Blagojevic, D. S. Nikolopoulos, A. Stamatakis, and C. D. Antonopoulos, "Dynamic multigrain parallelization on the cell broadband engine," in *Principles and Practice of Parallel Programming (PPoPP), 2007*, 2007.

[40] C. Benthin, I. Wald, M. Scherbaum, and H. Friedrich., "Ray tracing on the cell processor." in *Technical Report,inTrace Realtime Ray Tracing GmbH, No inTrace-2006-001*, 2006.

[41] D. Bader, V. Agarwal, and K. Madduri, "On the design and analysis of irregular algorithms on the cell processor: A case study on list ranking," in *21th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2007.

[42] D. Bader and V. Agarwal, "Fftc: Fastest fourier transform on the ibm cell broadband engine," in *14th IEEE International Conference on High Performance Computing (HiPC 2007)*, 2007.

[43] O. Villa, D. Scarpazza, F. Petrini, and J. Peinador, "Challenges in mapping graph exploration algorithms on advanced multi-core processors," in *21th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2007.

[44] F. Blagojevic, A. Stamatakis, C. Antonopoulos, and D. Nikolopoulos, "Raxml-cell: Parallel phylogenetic tree inference on the cell broadband engine," in *21th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2007.

[45] F. Petrini, G. Fossum, J. Fernandez, A. Varbanescu, M. Kistler, and M. Perrone, "Multicore surprises: Lessons learned from optimizing sweep3d on the cell broadband engine," in *21th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2007.

[46] S. Alam, J. Meredith, and J. Vetter, "Balancing productivity and performance on the cell broadband engine," in *IEEE Annual International Conference on Cluster Computing (Cluster 2007)*, 2007.

[47] P. B. et. al., "Cellsc: a programming model for the cell be architecture," 2006.

[48] K. O'Brien, K. O'Brien, Z. Sura, T. Chen, and T. Zhang, "Supporting openmp on cell," in *International Workshop on OpenMP (IWOMP), 2007*, 2007.

[49] H. Wei and J. Yu, "Mapping openmp to cell: A effective compiler framework for heterogeneous multi-core chip," in *International Workshop on OpenMP (IWOMP), 2007*, 2007.

[50] J. S. Park, J. G. Park, and H. J. Song, "Implementation of openmp workshares on cell broadband engine," in *International Workshop on OpenMP (IWOMP), 2007*, 2007.

[51] J. Manzano, Z. Hu, Y. Jiang, and G. Gan, "Towards an automatic code layout framework," in *International Workshop on OpenMP (IWOMP), 2007*, 2007.

Appendix

MEMO OF SOURCE CODE CHANGE IN ASSEBMLER AND LINKER

In this appendix, we highlight all souce code changes in the assembler and linker of our Open Opell system. In section A.1 we present the changes on the important data structures. We list all modified or created functions in section A.2. Finally, all critical global data involved in our implementation is given.

A.1 Data Structures

```
struct bfd{...};

typedef struct bfd_section{...} asection;

typedef enum bfd_reloc_status {...};
```

A.2 Functions

```
void
obj_elf_section (int push);

void
obj_elf_partition (int push);

void
cross_partition_call(int arg);
```

```
void
function_pointer(int arg);

void
library_callee(int arg);

static enum elf_spu_reloc_type
spu_elf_bfd_to_reloc_type (bfd_reloc_code_real_type code);

void
md_apply_fix3 (fixP, valP, seg)
     fixS *fixP;
     valueT * valP;
     segT seg ATTRIBUTE_UNUSED;

static bfd_boolean
spu_elf_check_relocs (bfd *abfd, struct bfd_link_info *info,
      asection *sec, const Elf_Internal_Rela *relocs);

static bfd_boolean
spu_elf_relocate_section (bfd * output_bfd ATTRIBUTE_UNUSED,
    struct bfd_link_info *info,
    bfd * input_bfd,
    asection * input_section,
    bfd_byte * contents,
    Elf_Internal_Rela * relocs,
    Elf_Internal_Sym * local_syms,
    asection ** local_sections);

static bfd_boolean
allocate_dynrelocs (bfd *abfd, struct bfd_link_info *info,
      asection *sec, const Elf_Internal_Rela *relocs);

START_RELOC_NUMBERS (elf_spu_reloc_type)
RELOC_NUMBER (R_SPU_PID6I, 13)
END_RELOC_NUMBERS (R_SPU_max)

bfd_boolean
_bfd_elf_make_section_from_shdr (bfd *abfd,
 Elf_Internal_Shdr *hdr,
 const char *name);
```

```
static void
elf_fake_sections (bfd *abfd,
   asection *asect, void *failedptrarg);

static bfd_boolean
assign_file_positions_for_segments (bfd *abfd,
  struct bfd_link_info *link_info);

bfd_boolean
bfd_elf_final_link (bfd *abfd,
struct bfd_link_info *info);

bfd_reloc_status_type
bfd_check_overflow (enum complain_overflow how,
    unsigned int bitsize,
    unsigned int rightshift,
    unsigned int addrsize,
    bfd_vma relocation);
```

A.3 Datas

```
static const pseudo_typeS elf_pseudo_table[] =
{...};

static const char *const bfd_reloc_code_real_names[]=
{...};

static struct arg_encode arg_encode[A_MAX] =
{...};

static reloc_howto_type elf_howto_table[] =
{...};

char  function_pointer_call;

char  lib_callee;
```

```
char  cross_partition_caller[20];

#define SHF_PARTITION 0x40000000 /* this section is a partition
   and should resides in overlay */
```