A Tiger Compiler for the Cell Broadband Engine Architecture

A Thesis Submitted
to the College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in the Department of Computer Science
University of Saskatchewan

by

Yiqing Liu

Saskatoon, Saskatchewan, Canada

Permission to Use

Acknowledgments

I would like to take this opportunity to express my deepest appreciation to all the people who helped me accomplish this work.

1. My family, thank you for your invaluable love. I would not have gotten this work done without your support.
2. My friends, Yuan and Chuan, thank you for your encouragement throughout these years, especially for your help in those tough times.
3. SRL members, Jeeva S. Paudel, Andriy Hnativ, Fatima Alawami, Zahrah Alawami, Yosuke Yamamoto, Gadi Tellez Espinosa. Thank you for comments and suggestions on my presentation.
4. Committee members Dr. Nathaniel Osgood and Dr. Chanchal Roy. Thank you for your invaluable suggestions and support for my thesis completion.
5. Last and most important, my supervisor, Dr. Christopher Dutchyn. Thank you for your patience and devoting your precious time in my study. I really appreciate your suggestions and the encouragement you offered when I encountered the stress of the research process. Thank you for teaching me how to think and learn.

Abstract

The modern computing industry tends to build integrated circuits with multiple energy-efficient cores instead of ramping up the clock speed for each single processing unit. While each core may not run as fast as the single core model, such architecture allows more jobs to be handled in parallel and also provides better overall performance. Asymmetric Multiprocessing, also known as Heterogeneous Multiprocessing, involves multiple processors that differ architecturally from one another, especially where each processor has its own memory space. Under power limitations, this design could provide better performance than that attained through symmetric multiprocessing. However, the heterogeneous nature adds difficulty to programming. Each specific architecture requires its own program code. Programmers also need to explicitly transfer code and data between processors.

This study describes the implementation of a compiler of the pedagogic Tiger language for the Cell Broadband Engine, an asymmetric multiprocessing platform jointly developed by Sony, Toshiba and IBM. The problem above is solved by introducing multiple backends for the Tiger language, along with a remote call stub (RCS) generator. Functions are compiled into different architectures, and calls across architectures are linked automatically through the stubs. RCS takes care of the execution context switch and hides details of the argument data/return value transfer. TigC simplifies the programming and building procedures. It also provides a high-level view of the whole program execution for future optimization because all of the source files are processed by a single compiler. As an example of this procedure, the possible optimization of data transfer during remote calls is investigated here.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ABI	Application Binary Interface
AMP	Asymmetric MultiProcessing
BCE	Base Core Equivalent
BP	Base Pointer
Cell B.E.	Cell Broadband Engine
CellSs	Cell Superscalar
DMA	Direct Memory Access
EIB	Element Interconnect Bus
FP	Frame Pointer
GPU	Graphics Processing Unit
ILP	Instruction-level Parallelism
IR	Intermediate Representation
MFC	Memory Flow Controller
NUMA	Non-Uniform Memory Access
OOE	Out-of-Order Execution
OpenMP	Open Multiprocessing
PE	Processing Element
PPE	PowerPC Processor Element
PPSS	PowerPC Processor Storage Subsystem

PPU	PowerPC Processor Unit
RCS	Remote Call Stub
RPC	Remote Procedure Calls
SML/NJ	Standard ML of New Jersey
SMP	Symmetric Multiprocessing
SP	Stack Pointer
SPE	Synergistic Processor Element
SPU	Synergistic Processing Unit
TLP	Thread-level Parallelism
TOC	Table of Contents

1. Introduction

With contemporary advances in circuit integration, the industry has been applying Moore's Law [46] to ramp up the clock of a single processor, or uniprocessor by putting more transistors on a die. However, when the transistor count reaches a certain point, the processor performance will saturate because of the so-called "Power Wall" problem. This physical limit of semiconductor-based microelectronics makes it difficult to improve single processor clock frequency. In addition, heat dissipation and data synchronization will also pose problems in this case. All of these reasons make hardware designers move away from conventional superscalar processors and study various alternatives that may increase the productivity; Instruction-level parallelism (ILP) [42] [43] and thread-level parallelism (TLP) [9] offer such possibilities. Multiprocessing, as one of the TLP methods, refers to the usage of more than one processor in a single computing unit [6] and now is widely used in almost all areas of computing. These applications include general-purpose desktop computers, embedded systems, and networking devices. Instead of focusing on increasing the performance of a single processor, multiprocessing is designed to improve the overall speed for programs that can be run in parallel.

1.1 Motivation

While multiprocessing has emerged in order to overcome challenges stemming from the physical limits of the conventional processor, simply offering multiple processors cannot directly improve performance and reduce energy usage for most of the applications. Developers should take the responsibility for exploiting the parallelism and fully engaging the power of all their available processors. Programming becomes increasingly complex because data in the memory may be accessed by different codes at the same time, which requires some sort of synchronization. Consequently, different programming models are used to make the synchronization easier[15], such as pthread[31], OpenMP[14], and others.

For example, one type of multiprocessing, namely asymmetric multiprocessing, involves different processor architectures within one system. This means a binary targeting on one architecture will not work on other types of processors. The whole procedure of developing on this type of system becomes more complicated because of the following extra work:

- Involves more steps to compile the source files. As mentioned, processors may have incompatible instruction sets and require their own binaries. Programmers need to provide different types of source files for this system and compile them separately.
- Requires the implementation of a program loader. A program designed for other processors is not loaded automatically by the system; the programmer needs to write a piece of code to manually transfer it to the target processor and correctly setup the environment for its execution.
- Requires the programmers to manage data transfer between different processors. Programs running on different processors can communicate in many ways. This communication needs to be implemented in the source files and interleaved efficiently with the actual computation code.
- Difficult to analyze and optimize the program. Source files are now processed by different compilers where each compiler runs independently. Usually there is no data shared between different compilers, so it is difficult for them to exchange information and produce a global view of the whole program.

This thesis explores all of these issues based on one asymmetric multiprocessing platform, Cell Broadband Engine [41]. The pedagogic language known as Tiger[4] serves as the input language for this compilation procedure.

1.2 Thesis Statement

A single compiler that translates the program fragments for different architectures of an asymmetric multiprocessing system offers opportunities for better program modularization and performance.

Further optimization is investigated and described in this thesis, including the following tasks:

1. Analyze and reorder the instructions so that the program can still run some code while calling a function on a different processor.
2. Evaluate different distribution methods for an input source file and minimize the running time of the program.

1.3 Contributions

This thesis implements a compiler to simplify the programming and compiling procedures on an asymmetric multiprocessing platform. The goal is to process functions defined for different architectures automatically and to generate corresponding assembly codes. Furthermore, the compiler hides the details of handling the processor switches during function calls and data transfer between processors. This feature makes programming easier and also provides more information for future optimizations.

The specific contributions include the following items:

1. The Tiger language is extended with some new keywords to support function definitions for different architectures. The frontend is modified to recognize these keywords and generate correct syntax trees.

2. Multiple backends are implemented to emit corresponding system instructions. The compiler will choose the correct backend depending on the function type.

3. Stub functions are generated automatically by the compiler to hide all the details of function calls and argument transfers. Programs can trigger a heterogeneous function execution under simple or parallel mode.

4. The data transfer between processors is optimized to reduce the overhead of processor switching.

5. The linking procedure is automated to hide both the stub functions and different assembly files.

1.4 Thesis Organization

The rest of this paper is organized as follows: Chapter 2 conveys the background knowledge of multiprocessing, the Tiger language, and related work in this area. Chapter 3 gives an overview of the solution to the problem explained above and describe the operating environment and general design of the compiler. Chapter 4 discusses in detail all the components components and explore a number of practical problems that were solved during implementation. Then, it discuss the optimization for data transfer across processors. Chapter 5 demonstrates performance results for different practical applications of the compiler and then describes potential optimization opportunities provided by the compiler. Finally, Chapter 6 concludes and outlines future work.

2. Background

To take advantage of multiprocessing, a recent trend is to implement multi-threaded software[3]. By sharing data across processors, threads can be distributed and executed in parallel. However, from the nature of algorithms, these types of programs inevitably contain a sequential execution inside each thread and require synchronization across threads. This portion of the code serves as the primary factor for any performance improvement to be gained from multiprocessing. This chapter will discuss how multiprocessing impacts the performance, paying particular attention to describing the different flavors of multiprocessing implementations. Then, Amdahl's Law is introduced which characterizes the impact of sequential program fragments on overall performance. It also details the Tiger language and covers some general requirements for the compiler. Finally, this chapter ends with a literature review of related work.

2.1 Symmetric and Asymmetric

The development of multiple processors has spawned various implementations to partition and distribute the code. Two major operating modes [48] employed in the industry are Symmetric Multiprocessing and Asymmetric Multiprocessing:

1. **Symmetric Multiprocessing (SMP)**
   SMP[48] refers to the employment of several identical, general-purpose processors in the computing unit. They may be connected to a single memory through a system bus, yielding a symmetric view of memory where every processor sees all of it and operates on memory in the same way. Alternatively, such proces-

sors may have privileged access to a small amount of memory and limited or no access to all of the memory. This Non-Uniform Memory Access (NUMA) architecture could be constructed so that every processor has its own fast local memory. Figure 2.1 shows a structure for an SMP system.

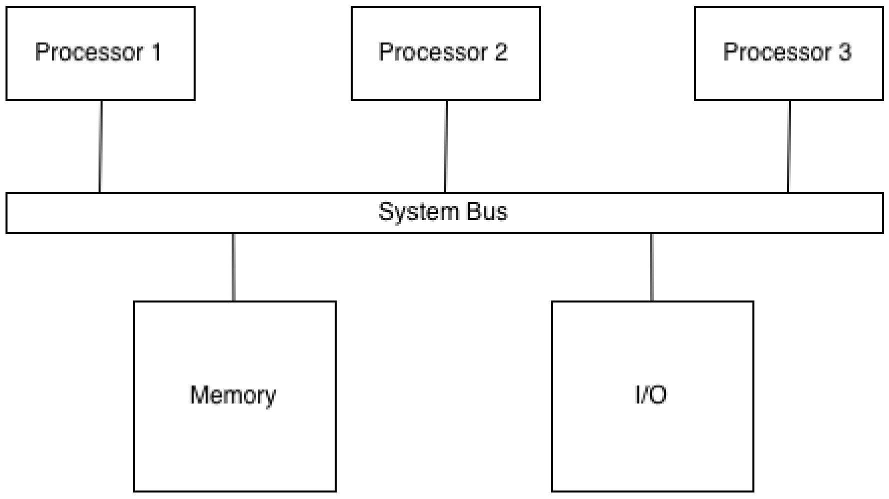


Figure 2.1: Symmetric Multiprocessing System

Most vendors are marketing SMP multicore processors, especially Intel/AMD x86 architectures, which are widely used in science, industry, and business. With carefully designed multi-threaded applications, SMP can bring performance improvements by a factor approaching the number of processors. Even with programs designed for uniprocessors, one may still benefit from SMP because other processes or background tasks can be handled by the additional processors.

However, limitations in both performance and power for the uniprocessor still remain because SMP is a replication of the original single processor. To fully utilize the power of the additional processors, it also requires extra work to synchronize the execution and data during the software development. In addition, all processes are managed by the operating system. It must be built with SMP support so that a process can be scheduled for different processors.

The overhead for synchronization also increases with the number of processors because there could be more contention for system resources. For example, a situation of different processors accessing the same memory location requires protection such as a critical section or a lock to make sure only one can touch it exclusively. Other processors at the same time will either do busy waiting or sleep waiting for the next signal. Finally, if a shared system bus is used and each process has its own cache, an instruction operand may have several copies, one in main memory and one in each cache. If one copy is modified, the change must be propagated to different copies. This discipline, known as cache coherency, becomes a potential performance issue.

2. **Asymmetric Multiprocessing (AMP)**
   AMP[48] is based on the idea that no single processor is ideal for all types of existing applications. It includes different types of processors where each type is designed to solve one specific problem. For example, an AMP system may include a low power-consuming processor to manage resources and to distribute input data, along with several computation-intensive processors so that each works on a different portion of the data.

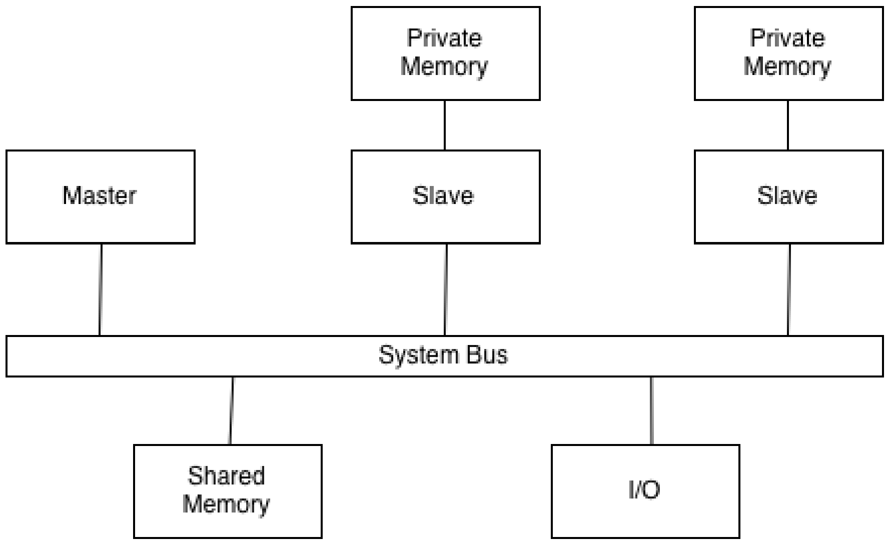


Figure 2.2: Asymmetric Multiprocessing System

Distributing input data among different processors requires resource management; generally, one processor is designated as the master processor to run the operating system and to perform I/O. The master distributes computational threads among other slave processors. An example AMP system is shown in Figure 2.2. It contains one master processor and two slave processors. Each slave processor has its own private memory. All processors are connected with the system bus and can access the main memory, as well as performing I/O. One special type of processor also considered as an AMP system in this study is the Graphics Processing Unit (GPU).

For each computation thread, AMP provides an execution environment that is similar to that of the conventional uniprocessor system. It is simple to migrate the legacy code to take advantage of multiprocessing. Also easier is the implementation of the operating system for AMP as the scheduling and system interrupts happen on the master processor.

AMP also has some disadvantages:

- A single computational unit cannot be split to run on different processors, so some of them may be left idle if the algorithm is not well selected. This circumstance causes underutilization of the processors. For example, if one processor gets busy while many threads are running on it, there is difficulty in most cases to move them to another processor that has free CPU cycles. A complex checkpointing procedure is required to save and restore the running state of a thread, which leads to service interruption during the movement.
- Processes will not be able to take advantage of AMP easily because the operating system is running on the master processor. The task must be designed for some specific processor and explicitly assigned to that type of slave processor in AMP. By contrast, in SMP, all processors are compatible and the scheduling across processors is transparent for programmers.

This research work focuses on the AMP design.

2.2 Amdahl's Law

Amdahl's Law[2] is used to calculate the maximum speedup of an overall system when a fraction of it is enhanced. The speedup is defined as the original execution time divided by the optimized execution time, and it can be calculated with Formula 2.1

$$Speedup_{enhanced}(f, S) = \frac{1}{(1-f)+\frac{f}{S}} \tag{2.1}$$

where f stands for the fraction of the system that is enhanced, and S means the speedup of this fraction. Amdahl's Law implies that if f is small, the overall optimization will have little effect.

This law also has a special case for multiprocessing. If n processors can be used in parallel, and there is no scheduling overhead, the speedup will still depend on the fraction of the program that can not be parallelized. It shows the calculation in Formula 2.2

$$Speedup_{parallel}(f, n) = \frac{1}{(1-f)+\frac{f}{n}} \tag{2.2}$$

Hill and Marty extended Amdahl's Law to multicore processors to demonstrate that Amdahl's Law still plays an important role even in the multicore era[21]. They were interested not only in maximizing performance, but also in taking the power limitation into account. In their paper, they illustrated a simple cost model to simulate three kinds of integrated circuit design and analyzed program performance on these integrated circuits.

Before discussing the different intreated circuit designs, they first introduced the term *base core equivalent* (BCE) to stand for the baseline core, also making some assumptions before the experiment:

- Assume for given size and technology, a multicore integrated circuit can support at most n BCEs.

- Assume, given the resources of r BCEs, that architects are able to rebuild them into one more powerful core with performance of *perf(r)* in sequential execution.
- Assume *perf(r)* is calculated as:

  $$perf(r) = \sqrt{r}$$

  For example, with 4 BCEs, the performance is doubled.
- Assume the non-processor resources such as shared caches, interconnections, memory controllers, etc. are constant in the experiment.

In their experiments, these three different types of integrated circuit designs were:

- *Symmetric Designs*: In symmetric designs, all the cores on the integrated circuit have the same cost, as shown in Figure 2.3. In the symmetric design, only one

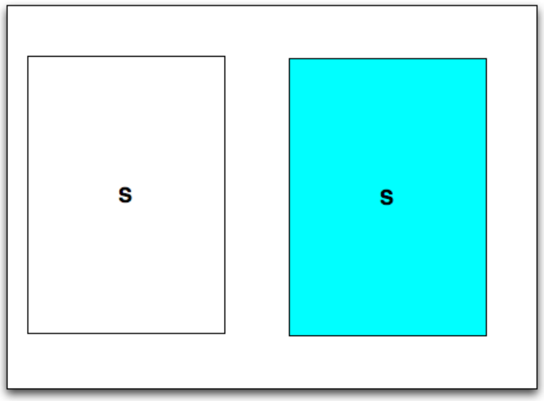


Figure 2.3: Symmetric Designs

core with resources r BCEs will run the sequential part of the program at a performance of *perf(r)*; all the n/r cores will run the parallel part of the program at a performance of *perf(r)*n/r*, so the speedup equation is reformed into:

$$Speedup_{parallel}(f, n, r) = \frac{1}{\frac{(1-f)}{perf(r)} + \frac{f}{perf(r) * \frac{n}{r}}} \tag{2.3}$$

- *Asymmetric Designs*: In asymmetric designs, one or more cores are more powerful than the other cores on the integrated circuit, as shown in Figure 2.4. It is very common to design one more powerful core with r BCEs to execute

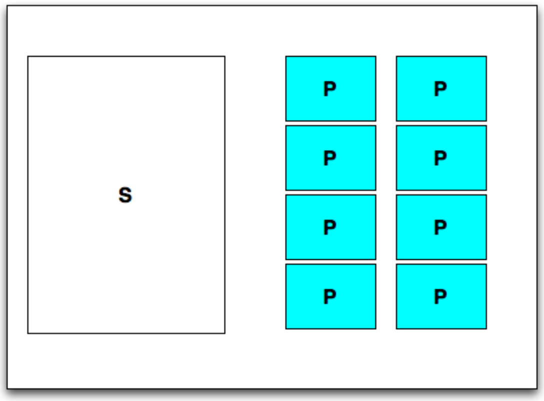


Figure 2.4: Asymmetric Designs

the sequential part of the program at a performance of *perf(r)*, and it will run the parallel part of the program together with all the n-r 1-BCE cores to get the performance of *perf(r)+(n-r)*. The enhanced speedup equation is shown as below:

$$Speedup_{parallel}(f, n, r) = \frac{1}{\frac{1-f}{perf(r)} + \frac{f}{perf(r)+(n-r)}} \tag{2.4}$$

- *Dynamic Designs*: Different from symmetric or asymmetric designs, an integrated circuit with dynamic design switches between the sequential mode and the parallel mode. Figure 2.5 shows how these cores perform during the program execution.

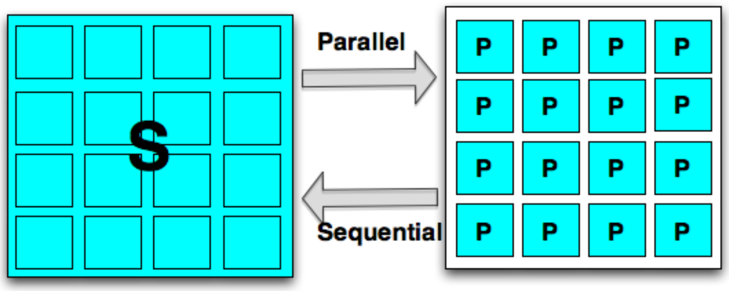


Figure 2.5: Dynamic Designs

This design employs r BCEs cores to be a richer core for the sequential part of the program with a performance of *perf(r)* and works as n 1-BCE cores for the parallel part of the program with a performance of n. So, the speedup equation is modified as shown below:

$$Speedup_{parallel}(f, n, r) = \frac{1}{\frac{1-f}{perf(r)} + \frac{f}{n}} \tag{2.5}$$

The authors applied Amdahl's Law to multicore integrated circuits using these three different techniques. Their simulation results show that asymmetric multicore designs provide better speedup than those enabled by symmetric designs. Dynamic designs have the potential to achieve the best of both worlds, especially under modern power limitations. Results also showed that even in the multicore era, sequential program execution still dominates program performance.

Asymmetric processor code generation is not deeply investigated, but is necessary to effectively employ power-limited processors. A goal of this research is to evaluate these simulation-based claims using existing hardware. The Cell B.E. is used as an asymmetrically designed integrated circuit.

2.3 Cell Broadband Engine (Cell B.E.) Architecture

Few AMP systems exist nowadays, but one has achieved commercial success: Cell B.E. is a novel, heterogeneous, multicore architecture [45], designed by IBM, Sony, and Toshiba, to provide significant performance improvements for computation-intensive tasks at a relatively low power consumption. It is representative of AMP designs and has been widely used by both industry and academia; one familiar example of domestic use is Sony PlayStation 3. The processor is a multi-core integrated circuit consisting of a standard 64-bit Power Architecture processor with eight high-frequency synergistic processor cores offering pipelined SIMD capabilities [29]. To facilitate communication between processors, it also integrates a high-speed memory controller and a high-bandwidth bus interface on the integrated circuit. The structure of Cell

B.E. system [25] is shown in Figure 2.6. The Element Interconnect Bus (EIB) is the system bus of Cell B.E. which connects all other components. A memory controller and an I/O controller provide interfaces to access different hardware resources. The remaining components will be described in this section.

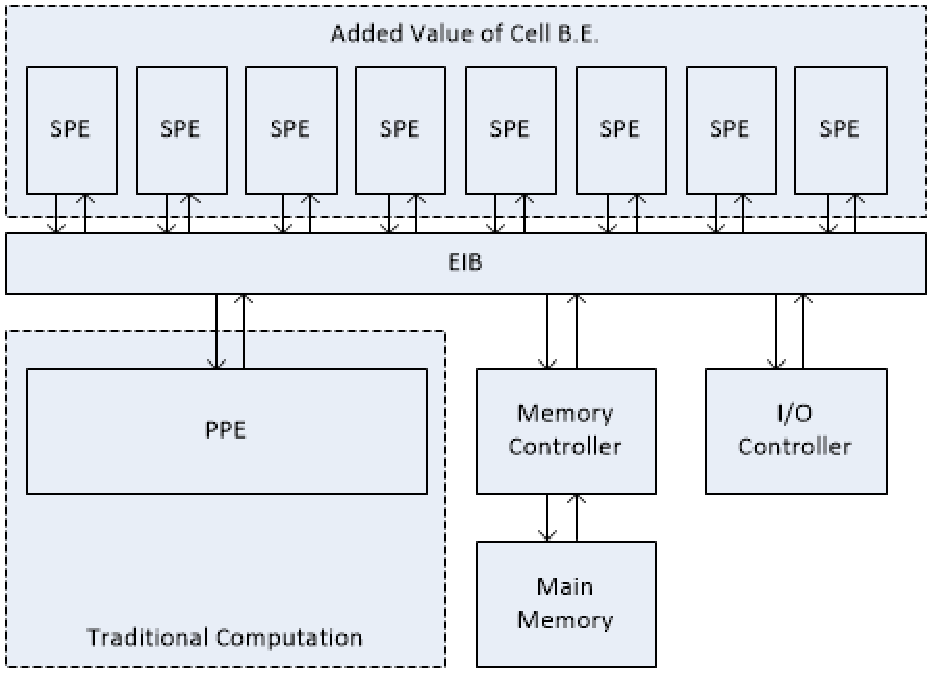


Figure 2.6: Cell Broadband Engine System Structure

2.3.1 PowerPC Processor Element

A PowerPC Processor Element(PPE) is the general purpose processing unit included in Cell B.E. intended for system resources management. It has a 64-bit PowerPC processor unit (PPU) that conforms to standard PowerPC architecture and a PowerPC virtual memory subsystem. The PPU supports dual-thread processing and employs an extended version of the PowerPC instruction set, which includes SIMD[16] instructions. Together with 32 special vector registers, these extra instructions are used to broadcast an instruction to multiple computation units where this instruction is executed on different data simultaneously in order to achieve the data-level parallelism.

Typically, the operating system runs on PPE and has an overall control of the whole system. Parallelizable tasks sent to Cell B.E. are distributed by PPE to the synergistic processors. Figure 2.7 shows the structure of PPE.

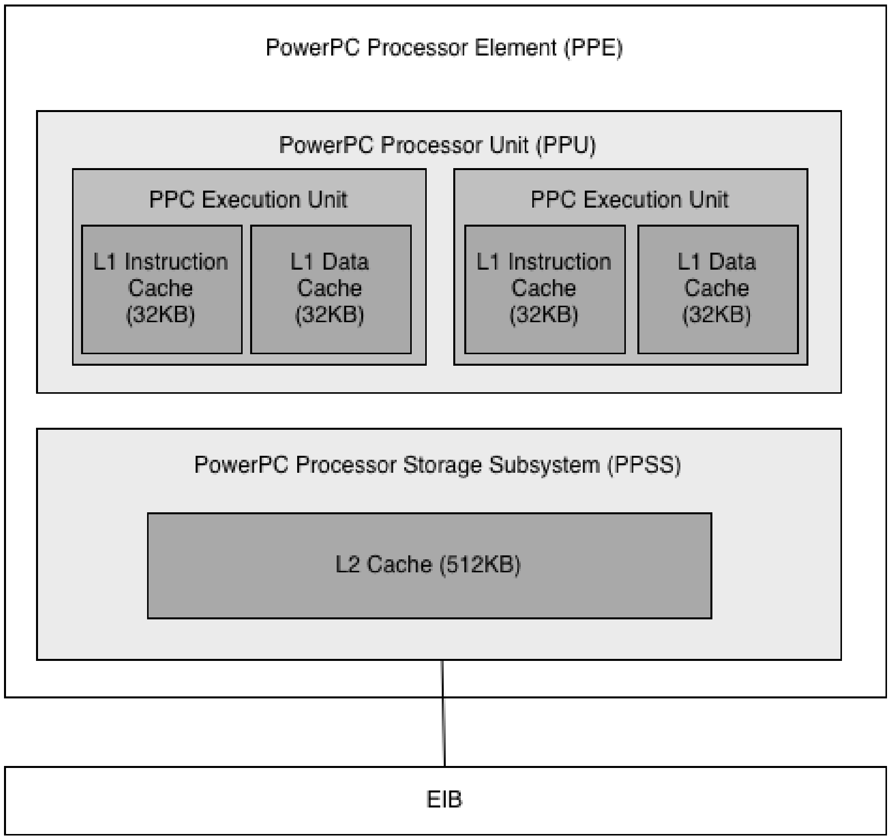

PowerPC Processor Element (PPE)
PowerPC Processor Unit (PPU)
PPC Execution Unit
L1 Instruction Cache (32KB)
L1 Data Cache (32KB)
PPC Execution Unit
L1 Instruction Cache (32KB)
L1 Data Cache (32KB)
PowerPC Processor Storage Subsystem (PPSS)
L2 Cache (512KB)
EIB

Figure 2.7: PowerPC Processor Element

The PPU is a 2-way multiprocessor with shared data flow, and it can support two simultaneous threads of execution. Simplistically, the PPU itself can be viewed as an SMP system with two independent processors. For instruction fetch, it contains a 32KB L1 cache which is 2-way set-associative, reload-on-error and parity protected. The cache line size is 128 bytes. It also includes another 32KB cache for data access. This cache is 4-way set-associative, write-through and parity protected. The cache line size is 128 bytes.

The PowerPC Processor Storage Subsystem (PPSS) handles memory requests from PPE in addition to memory coherence operations from EIB. PPSS includes a

512KB L2 cache that is 8-way set-associative, write-back with ECC support. The cache line size is also 128 bytes.

In a Cell B.E., all hardware resources are mapped to the PPE's linear address space so that PPE can directly access any of these resources with an effective address value. PPE also supports the vector/SIMD multimedia extension to run software that uses this extension, meaning that, with a well designed distribution mechanism, PPE can also serve to do some computation work.

2.3.2 Synergistic Processor Element

The Synergistic Processor Element (SPE) is the novel part of Cell B.E. architecture. It is named as "synergistic" because it works as a coprocessor to the PPE. Optimized for computationally intensive processing instead of general purposes, these eight SPEs are less complex cores able to achieve superior performance on critical workloads such as gaming, media encoding, and similar activities. Figure 2.8 shows the structure of SPE. Each SPE includes a Synergistic Processing Unit (SPU), a Memory Flow Controller (MFC), and in the current generation of Cell B.E., a 256KB embedded local storage for both instruction and data.

Optimized for computation, each SPU processor contains 128 registers, and each register is 128 bits wide. This capacity benefits instruction scheduling and allows some optimization techniques. The SPU instruction set does not match other systems, such as x86, MIPS, but is optimized for this large number of registers in design. It includes the following major features:

- Sequential loading and storing on these 128-bit registers. This means the SPU will always execute the load and store instructions in the order specified by the program. Even if two instructions have no dependency, the SPU does not reorder them.

- SIMD arithmetic on multiple data elements up to 128 bits in total. In contrast to PPE wherein a separate set of registers is used, SIMD registers on SPE

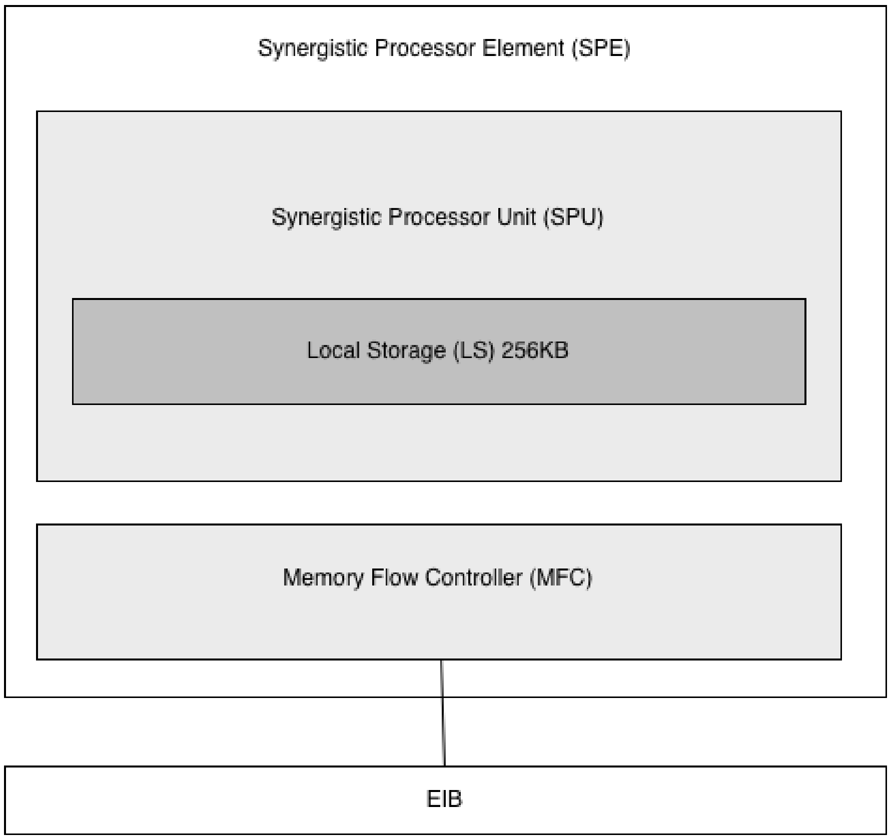

Synergistic Processor Element (SPE)
Synergistic Processor Unit (SPU)
Local Storage (LS) 256KB
Memory Flow Controller (MFC)
EIB

Figure 2.8: Synergistic Processor Element

are also unified. They can act as operands for both integer and floating-point instructions.

- Memory access instructions are restricted to the local storage. There is no direct access to other resources. The program needs to issue DMA operations to transfer data between the local storage and main memory.
- Unlike a traditional system cache, this local storage is not transparent to software and does not provide prediction for loading data.

In the typical usage scenario, the system will load the program binary and data to the local storage of SPEs and chain them together to process each step of the operation. The registers, local storage and main memory offer a three-level memory structure for SPE, which is designed to avoid the false-sharing problems[10] and other cache inefficiencies frequently found in conventional processors.

2.3.3 Memory Flow Controller

A crucial component of an SPU is the memory flow controller. All data transfers between different stores on Cell B.E. platform, including main memory to local store, local store to main memory, and between local stores on different SPU. SPE entirely depends on MFC to transfer data between its local storage and main storage. An MFC command is translated into a DMA transfer, and MFC can support multiple commands at the same time. As shown in Figure 2.8, each SPE contains its own MFC. MFC can be instructed by both PPE and SPE to perform DMA operations. Requests are queued at MFC through two types of interfaces:

1. *SPU Channel*: As mentioned above, the SPU is decoupled from the system with limited access to other components, and MFC is required to access the main storage domain and other hardware resources. The MFC hardware provides the channel interface used by the SPUs to control MFC. Each channel is unidirectional, so one can either read from, or write data to, the MFC through it. An SPU interrupt mechanism detects data availability and avoids polling. Multiple channels are supported for each SPU.

2. *Memory-Mapped Register*: This register is used by PPE and all other devices. A request can be issued on behalf of an SPU through this interface.

Each MFC command is tagged with a 5-bit identifier. This identifier can be shared by multiple MFC commands, and these commands are called a tag group. By using this identifier, a program can check whether commands in a tag group have completed.

Based on the above data transfer mechanism, MFC also offers methods for data protection and synchronization. Several mechanisms are employed for communication between PPE and SPEs and between the SPEs, including the following items:

1. *Mailboxes*: This mechanism allows programs to exchange 32-bit messages between processors. The MFC of each SPU contains both the inbound mailboxe, where the local SPE reads and PPE and other SPEs write, and outbound mailboxes where a local SPE writes and PPE and other SPEs read.

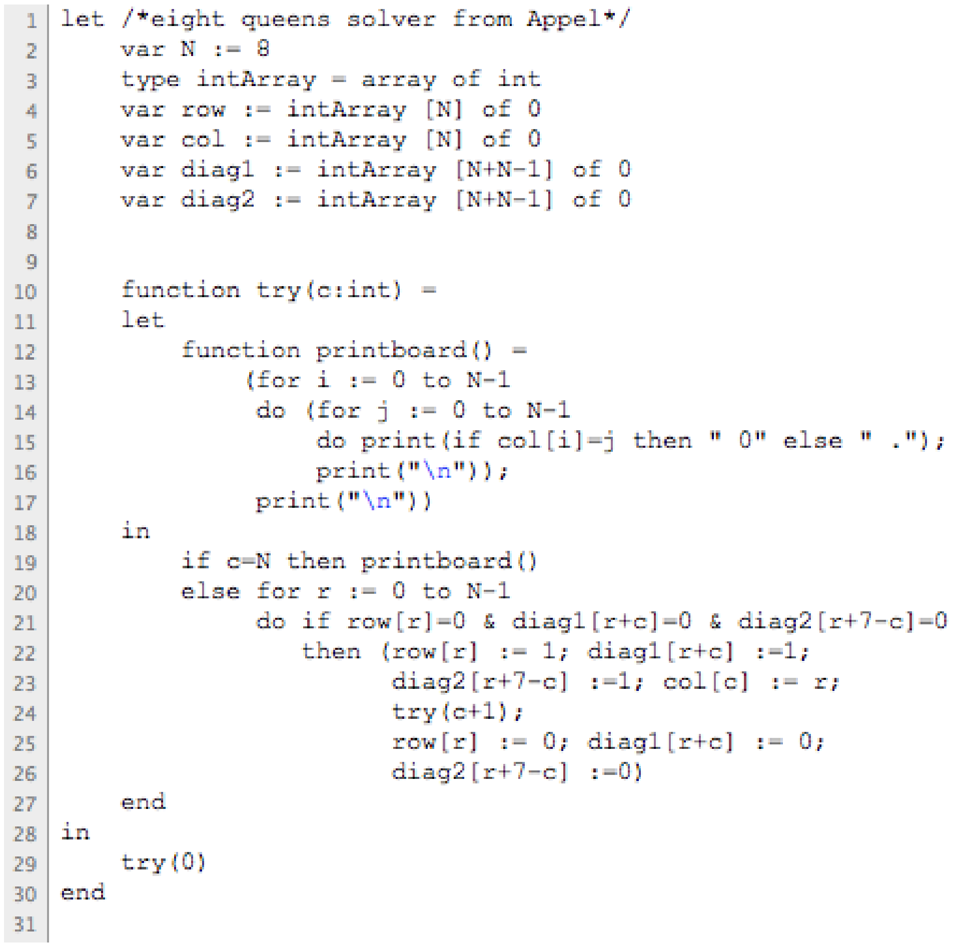
2. *Signal notification*: This allows a PPU program or an SPU program to signal an SPE with 32-bit signal registers. This is a one-direction operation with information being sent only toward the SPU which resides in the same SPE as the signal registers.

3. *SPE events*: This mechanism allows an SPU program to trace external events set by either hardware or the Mailboxes and Signal notification. The PPU program can also include an event handler to track the events which happen on SPEs.

2.4 Tiger Language

Tiger is a functional programming language derived from an example language introduced in Andrew Appel's *Modern Compiler Implementation* book [4]. It has built-in types, such as *int*, *string*, *record*, and *array*. Tiger allows flow control by supporting *for*, *while*, and *if*. It also supports statically nested functions with variables defined in outer functions available to inner functions. In addition, some built-in functions exist in Tiger for string manipulations and system interactions. An example of Tiger language is shown in Figure 2.9, listing a program that solves the classic 8-queens problem.

Four variables of type `intArray` are defined, named `row`, `col`, `diag1` and `diag2`. Function `try` is a recursive function which takes one parameter to describe the current column. It tries each different placement on the current column with a loop, and if it is valid, moves to the next column. When it finishes the last column, a nested function `printboard` defined as part of `try` will be called to print the current placement of the chessboard. Each variable in this program has its scope. For example, those



```
let /*eight queens solver from Appel*/
    var N := 8
    type intArray = array of int
    var row := intArray [N] of 0
    var col := intArray [N] of 0
    var diag1 := intArray [N+N-1] of 0
    var diag2 := intArray [N+N-1] of 0


    function try(c:int) =
    let
        function printboard() =
            (for i := 0 to N-1
             do (for j := 0 to N-1
                 do print(if col[i]=j then " 0" else " .");
                 print("\n"));
             print("\n"))
    in
        if c=N then printboard()
        else for r := 0 to N-1
             do if row[r]=0 & diag1[r+c]=0 & diag2[r+7-c]=0
                then (row[r] := 1; diag1[r+c] :=1;
                      diag2[r+7-c] :=1; col[c] := r;
                      try(c+1);
                      row[r] := 0; diag1[r+c] := 0;
                      diag2[r+7-c] :=0)
    end
in
    try(0)
end
```

Figure 2.9: Tiger Example Source

variables defined in the `let...in` can be accessed by all codes in this `let...in...end` structure, but `i` defined in `printboard` can be accessed only by this function.

In this thesis, this study will extend the standard Tiger language by adding keywords and flow control logic to build an executable program run across the PPU and the SPU processors on the Cell B.E. platform. The Tiger language in this thesis supports all the above features. But it has limitations on function calls across processors. Nested functions will not be able to access parent's variables. Besides, with current implementation, only array is supported as complicated parameter type.

2.5 Overview of TigC

This thesis provides a Tiger compiler called TigC for Cell B.E. platform, which is able to accept mixed source code for both PPE and SPE processors and generate a single executable file to run across the processor. All details about transferring data between main memory and local storage and the heterogeneous function calls should be hidden by stub functions generated automatically by the compiler. This study will focus solely on the function calls from PPE to SPE, as this is the primary usage on Cell B.E. platform.

TigC supports two different backends, one for PPE and one for SPE. The frontend tags each function for its target platform. With these tags, function calls will be analyzed and all remote calls (across processors) will be translated into the execution of the wrapper code. Therefore, TigC requires another component to generate the wrapper code in order to transfer the input/output data between processors, to manage the usage of SPEs, and to synchronize the execution contexts. The compiler also needs to support distribution of a function onto different SPEs, each with a subset of the input data, thereby employing the full power of Cell B.E. platform.

Lastly, to make the compiler easily extensible to support other AMP platforms, or even programming with GPU, there should be only a few changes involved with the general compiler frontend. A general backend interface is proposed so that it can be extended to describe the features of all architectures.

2.6 Related Work

This section describes previous research that is closely related to the TigC implementation. Different solutions are proposed to offer convenient techniques for writing programs on heterogeneous systems. These works are categorized into different groups based on their target platform: Generic Multiprocessing Systems, GPU, and Cell B.E. system. These groups are described below.

2.6.1 Generic Multiprocessing System

Early implementations of simultaneous executions include array processors, which handle multiple data elements at the same time in a parallel fashion. They use special instructions optimized for array operations. Differing from multiprocessors, all computation units in array processors can work only on a single set of data elements. In [7], G.H.Barnes describes the structure of a parallel-array computer, ILLIAC IV, which includes 256 processing elements (PEs) for arithmetic and logic operations. Each PE has its own 2048-word 64-bit per word random-access memory, named PEM. Such PEs are used for multi-array processing. W. J. Watson in [50] describes the TI-ASC system developed by Texas Instruments. This system implements a single instruction to read streams of operands from memory into a vector unit and processes them. The result stream will then be sent back to the memory.

In [11], John R. Callahan and James M. Purtilo propose a solution to integrate heterogeneous, distributed software components. These components are implemented with different programming languages, different data representation formats, and perhaps different runtime environments. To link all these components together, the authors propose a tool that analyzes the program and generates a wrapper function as a bridge to connect the call between two components. Phillip B. Gibbons in [19] describes a stub generator that is both language and machine independent. Based on a set of language and machine specifications, the stub generator generates stub functions that marshal the arguments and return values, also doing the remote procedure calls (RPC)[37]. These research examples provide the ideas to do the heterogeneous function calls between processors.

Mary W. Hall et al., in addition, describes the SUIF Compiler in [20], targeting on shared memory multiprocessors. The compiler locates the coarse-grain parallelism by analyzing the scalar and array usages in the program. These parts of the program will be distributed to different processors.

Open Multiprocessing (OpenMP)[12] is library support for multiprocessing programming on different platforms, such as AIX, HP-UX, Linux, Mac OS X, and Windows. Different languages can be used with OpenMP, including C, C++, and Fortran. It has evolved towards the parallelization of a wide range of applications ranging from desktops to supercomputers. Preprocessor directives are used to create threads upon entering a section. For example, the following pragma is utilized to fork multiple threads for executing the next adjacent code section:

```
#pragma omp parallel
```

where `omp` represents the OpenMP preprocessor directive, and `parallel` tells the compiler that the following code block is a parallel region with a default number of threads. Eduard Ayguade et al. in [5] explores the approach to employ OpenMP on heterogeneous systems, including GPU and Cell B.E. processors. They provide a new pragma to precede the existing pragma task:

```
#pragma omp target device(device-name-list) [clause-list]
```

where the `target` construct specifies that the function declared right after is designed to be executed on all the devices which appear in the `device-name-list`. A device could be *cell*, *cuda*, or *fpga*. In this way, users are allowed to specify the target processor to run a section in the program.

2.6.2 Cell B.E. Platform

Mao and Shen [34] explores the capability of Cell B.E. platform for scientific computing. This paper presents the experience of implementing the LU decomposition to utilize fully the power of a Cell B.E. system. Various performance factors are considered for occasions while the tasks and data are distributed to different SPUs, including:

1. *Locality*: A ready queue is maintained on SPE. When an SPU is about to process a task, it checks this queue and issues the prefetch instructions for the data that the newly entered program may need. This is a non-blocking operation, so the current task immediately starts. Double buffering is used to overlap the

computation and prefetch. In addition, a software cache is implemented as an index array to record what data exists in the local storage.

2. *Load balancing*: Four different distribution schemes are evaluated. The dynamic distribution puts a task into the ready queue that has the fewest tasks in order to produce a good load balance among SPEs. It is locality oblivious because it assumes that no data is shared between tasks. The other three distribution schemes are static but have different partitions. A 1D-unbalance scheme splits the matrix evenly. Because LU decomposition involves different levels of computation complexity on different parts of the matrix, this scheme leads to the worst balance. 1D-interleaving improves the load balancing by interleaving the distribution. 2D-interleaving does the same thing but also on the second dimension, and it has the best load balance. However, the first two schemes have better locality than the third one because the whole column is processed on the same SPU.

Evaluating different distribution schemes provides the compiler a baseline of how to generate the parallel code.

Cell superscalar (CellSs) [8] is a C compiler which automatically exploits the functional parallelism through different processing elements on Cell B.E. system. Users are required to provide some annotations in the source code (a feature that is similar to OpenMP) to be used by a source-to-source compiler for generating necessary helper code. An example of the annotation is shown below:

```
#pragma css task input(a{}, index) output(b{})
void array_op(float a[N], float b[N], int index);
```

where `css task` specifies that the following function should be scheduled to run on the SPU. It also describes the types of each parameter and specifies whether they are input or output parameters. Based on this information, a runtime library takes care of the task scheduling and data transfer between different processors.

Alexandre E. Eichenberger et al. in [17] discusses different optimizing techniques for a compiler for a CELL processor. The compiler is first optimized for branch execution because a branch misprediction penalty on the SPU is an 18-cycle high penalty. Then a simdization framework is introduced to generate the SIMD instructions automatically. In the end, the compiler employs the OpenMP pragmas to guide parallelization decisions and to distribute threads to different SPUs. The authors propose a single shared memory abstraction for data access. The compiler creates a software cache for the SPU, then it analyzes the program to replace the load and store instructions with instructions that search the operand in the cache. For the data that appears in the cache, the address will be calculated and then passed to the load and store operations. Otherwise, a subroutine is called to transfer the data from the main memory.

An extension to OpenMP was introduced by Kevin O'Brien et al. in [39] to support the Cell B.E. platform. Based on IBM's XL compiler which already supports OpenMP for AIX system, a new runtime library was developed to utilize the Cell B.E. SDK libraries to target the new platform. It implements the OpenMP memory model on the Cell B.E. memory system and generates thread code targeting different architectures. Wei and Yu in [51] does similar work by translating the program with OpenMP directives into codes targeting on both the PPU and the SPU.

2.6.3 GPU

CUDA is developed by NVIDA as a parallel computing platform and also a programming model [38]. It is employed to utilize the computation power of GPU, which is a specialized processor to address real-time, high-resolution graphics tasks. Current GPUs actually do more than just render graphics, instead having a substantial floating pointer performance that makes them perfect for applications from finance to medicine [33]. With CUDA, users are allowed to write applications with *kernel functions* in C, C++, and Fortran, as well as to execute code directly on GPU. There is no need to use an assembly language for GPU programs. CUDA can be considered as a special AMP system as GPU is different from the main processor. There are several approaches available for using CUDA in parallel computing:

1. *GPU-accelerated libraries* One example of the accelerated library is CULA[23], which is designed to solve Linear Algebra problems. It employs a hybrid processing model, meaning the code utilizes both CPU and GPU for its computation.

2. *OpenACC directives* OpenACC[40] shares the same idea as OpenMP by using directives to specify the parallel execution on the GPU. The code below shows one example of using OpenACC in C or C++:

   ```
   #pragma acc kernels copyin(a[0:n], b[0:n]), copyout(c[0:n])
   for (i = 0; i < n; i++) {
       c[i] = a[i] + b[i];
   }
   ```

   where a kernel is defined which runs on GPU and accepts two arrays, `a` and `b`. The output is stored in `c` and should be copied out from GPU after execution.

3. *Standard languages with extensions* For example, in [13], Ludovic Courtes developed the StarPU, an extension to the C language, to allow programmers to select a target for each function. A `starpu_codelet` is defined to describe the implementation and parameters of a task:

   ```
   struct starpu_codelet scale_vector_codelet =
   {
       .cpu_funcs = { scale_vector_cpu, NULL },
       .opencl_funcs = { scale_vector_opencl, NULL },
       .nbuffers = 1,
       .modes = { STARPU_RW },
       .name = "scale_vector"
   };
   ```

   where `.cpu_funcs` and `.opencl_funcs` describe the functions that run on the CPU and GPU. `nbuffers` denotes a possible transfer of large data back and forth between main memory and GPU. `modes` describes whether the buffer is accessed read-only, write-only, or read-write by the task. Then, C language is extended to process this structure specially and to generate a wrapper function to call the GPU version.

The authors in [30] offer a compiler frame which translates OpenMP program to use CUDA-based GPUs. The translator interprets the OpenMP semantics and identifies the *kernel regions* (code sections to be executed on the GPU). These *kernel regions* will then be transformed into CUDA kernel functions. Shared data will be analyzed to insert required memory transfer calls.

To improve the performance of OpenMP on heterogeneous systems, Thomas R. W. Scogland et al. in [47] discusses the task scheduling for accelerators, (e.g., GPU). Similar to the actions of OpenACC, an extra directive is introduced to define an accelerator region representing a code block that should run as a thread on an accelerator. New clauses are used to describe whether some values should be copied in and out, just in, or just out from the accelerator. Then, the authors evaluate both static splitting and dynamic splitting mechanisms to execute the accelerator region on both CPU and the accelerator.

2.7 Summary

This chapter described different types of multiprocessing systems: Symmetric and Asymmetric. The differences between them are discussed, such as their advantages and drawbacks, and several ways that their characteristics impact the programming models. Then, Amdahl's Law was introduced. It can be used to estimate the performance improvements with multiple processors. After that, the Cell B.E. system was considered in detail, including the important components, the way different processors communicate, and how data is transferred between main memory and each SPE's local storage. This chapter also examined the Tiger language, which is used as the input for the compiler. Finally, a literature review of existing research on multiprocessing systems and compilers optimized for systems is included.

3. Approach

This chapter summarizes the general design for the Tiger compiler, TigC. It starts by describing the system environment that contains all required components to support this compiler. Thereafter, the general steps of the compilation procedure and the structure of the compiler are discussed at a high level. Finally, it explores the architecture of the compiler.

3.1 Operating Environment

The core component of TigC is implemented in Standard ML of New Jersey (SML/NJ) language [35]. An implementation of SML/NJ is required in the system in order to build the compiler itself. TigC should focus on code generation, and to abstract the heterogeneous call implementation from the compilation procedure, a separate component in the compiler is designed to manage all the function definitions for different architectures and to generate the necessary stub code. This component is implemented in a Python script and works as an automation tool, so Python is a required tool in the system. Finally, this compiler generates assembly code and relies on the system assembler for the final binary.

Two different compilers are provided by IBM for development on Cell B.E., one for each of the two types of processors. The GNU toolchain contains the GCC compiler for both the PPU and the SPU, called *ppu-gcc* and *spu-gcc*. In addition, a separate command *ppu-embedspu* is provided to enable an SPU binary to be embedded with the PPU binary and loaded to the SPU processor at runtime. In TigC, these gcc commands are used to access the assembler and generate binaries.

To run the resulting executable file, if the program contains functions only for the PPU, TigC will generate the binary which is compatible with standard PowerPC 64-bit architecture. However, a Cell B.E. platform will be required when the program has SPU functions.

3.2 Steps of Compilation

While assembly language programs must be modified or rewritten execution on different hardware architectures, a compiler enables development with higher-level languages that are machine-independent. It accepts high-level source programs and translates them into machine code for the target hardware platform. The compilation procedure usually include several operations:

1. **Lexical Analysis**
   The compiler must understand the structure and meaning of the input source code to do the translation correctly. A scanner is used to break the input source code into individual words that are called "tokens". Based on given lexical rules, the scanner reads the source program represented as a sequence of accepted characters from the input stream (usually a source file) and produces a list of tokens.

2. **Syntax Parsing**
   Tokens from Lexical Analysis are passed here for further processing into abstract syntax. They will be put together and parsed to form the syntactic structure of the language. The syntax parser identifies the sentences that belong to the input language and then translates them into abstract syntax representation.

3. **Semantic Analysis**
   In this step, the semantic information will be added to the abstract-syntax representation. This means the compiler will associate variable and function definitions with their references and perform type-checking. Incorrect programs will be rejected in this step.

4. **Translation into IR**

   The abstract-syntax representation is transformed into IR, which describes the machine-level operations while not involving too much details Most important, IR is also independent of the source language so that, in future, the compiler can be easily extended to support different languages.

5. **Code Generation**

   Nodes of IR will be grouped and translated into clumps that correspond to the native system instructions in this step. It also does the register allocation where symbolic variable names used in IR will be replaced with actual registers.

In addition, the compiler usually also includes steps to optimize the size and speed of the resulting binary. The above operations are organized into three categories: the frontend that does lexical analysis and syntax parsing to generate the abstract syntax representation, the middleend to generate the IR, and the backend to translate the IR into assembly code. Optimizations on different levels could be involved in all these three phases to improve the performance and also the binary size.

As stated before, the goal of this thesis is to support different architectures and also to provide opportunities for better program modularization and performance. For each architecture, all the compilation steps should be applied to the corresponding source code to generate the resulting binary. This condition means that, logically, a program designed for an AMP system will have more than one resulting binary. To achieve this situation, the compiler could be designed with one of the following two methods:

1. **Separate compilations**

   The compiler itself is also a program that translates source language into machine code. To support different architectures, a simple solution is to have multiple compilers. The input source code should be categorized into groups of files where each group targets at one architecture. Then, the corresponding compiler program will be used to process each group in order to generate result binaries. A simple example is shown in Figure 3.1.

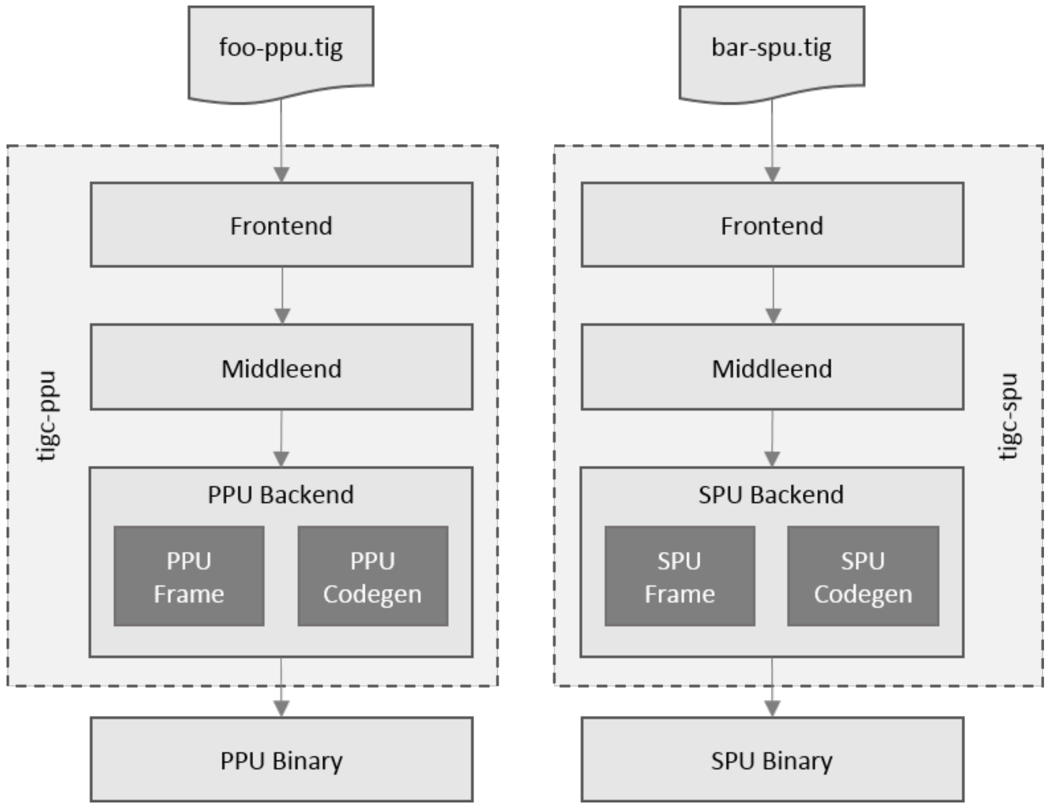


Figure 3.1: Example of Separate Compilations

In this example, the source code contains two different groups: one includes all the function definitions for the PPU architecture and the other for the SPU architecture. Each group has its own compilation procedure, which involves all the above steps from lexical analysis to the final code generation. These two compilations run as separate processes where little or no information is shared between them. In the end, two different binaries will be generated.

In this case, where only a single source language is used, it becomes obvious that although these two compilers belong to different programs, they could share the same frontend and middleend. Only the code generators are needed to be implemented for different architectures. This step may still involve significant investment, as modern processors are usually equipped with dedicated performance improvement technology, which then requires extensive compiler support for generating high quality code; the difference between processors makes it difficult to retarget the compiler to another architecture. So generic backend, such as MLRISC, could be used to solve these problems. Based on the machine description file, which includes information such as register file and instruction encoding, MLRISC generates the actual backend implementation and does the optimizations.

With this compilation model, the compilers do not handle any interactions between different architectures, a circumstance which simplifies the implementation of the compiler. However, programmers need to explicitly use the platform SDK to make function calls across processors.

2. **Unified compilation**

   The second way is based on the fact that the compiler works only on one source language. Because the frontend depends solely on the input language, the compiler could be designed as a single program that includes a unified frontend. Source code targeting at different architectures could then be processed into abstract syntax representation. However, to generate IR and the final machine code, each architecture requires its own backend implementation. One example of this compilation procedure is shown in Figure 3.2.

   In this example, there exists only one compiler program, but it has two backends. Each function definition in the source code is first processed by the frontend. Then depending on the information, provided either by programmers or by some decision logic inside the compiler, the function will be labeled with a tag indicating its target architecture; the corresponding backend will then be used for the code generation. Note that even with a single compilation procedure, the compiler will still generate two different binaries.

   As each function definition is tagged with its target architecture, by checking the caller and callee, the compiler can easily identify function calls across processors, handling and automatically generating some wrapper code to hide the implementation details. For programmers, calling a function on a different

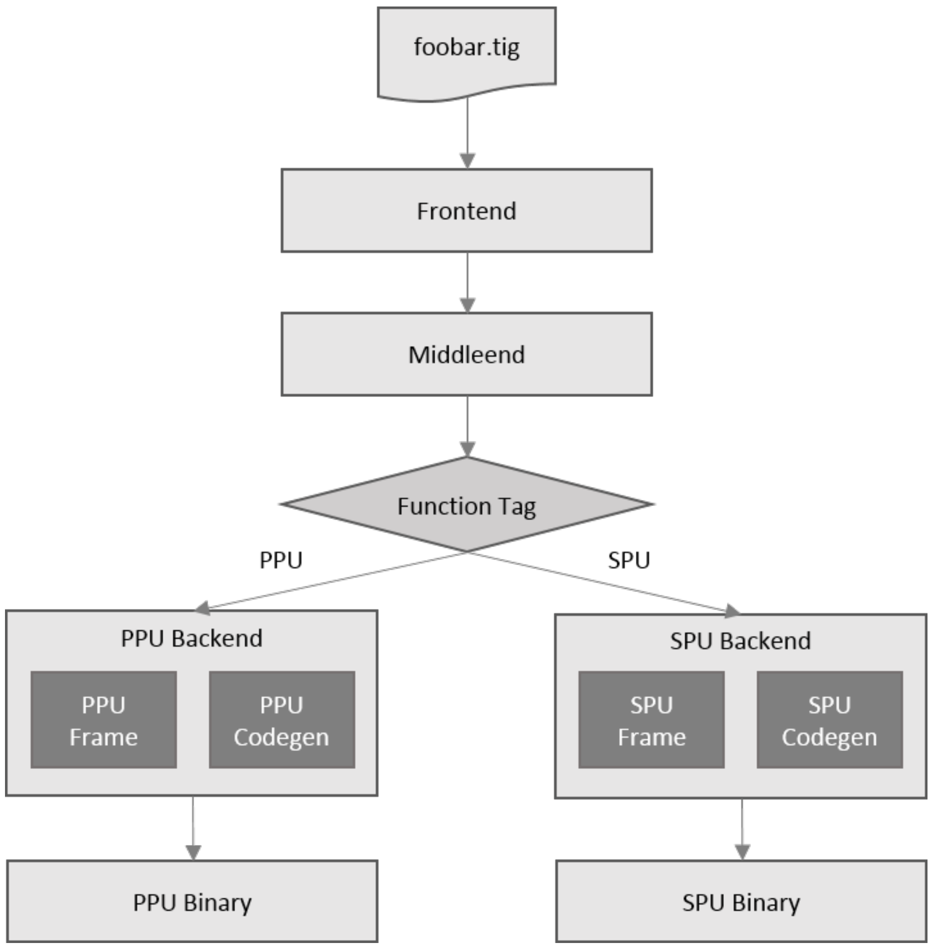


Figure 3.2: Example of Unified Compilations

architecture has no difference from a local function call.

To illustrate the difference between these two designs, the following Tiger program is used as an example: considering a piece of code targeting at the SPU platform and implemented as a function `bar`, it will be called by a function `foo` which is running on the PPU platform. The first design requires the programmer writing code as shown in Figure 3.3. File `foo.tig` includes the definition of function foo, and `bar.tig` includes the definition of the SPU function bar. To call an SPU function on the PPU, the programmer needs to implement the code that loads the SPU program, to manage the argument and return value passing, and to control processor switch. Compiling this program also requires two different procedures, which in the end gives us two resulting binaries.

Source Files:

```
foo.tig:
ppu_function foo() :=
  (ctx := spe_context_create();
   spe_program_load(
     ctx,
     bar_program);
   spe_program_run(ctx);
   spe_context_destroy(ctx)
  )

bar.tig:
function bar() :=
    ;; implementations of bar
function main() :=
    bar()
```

Compilation Procedure:

```
ppu-tigc foo.tig
spu-tigc bar.tig
```

Figure 3.3: Example Program of the First Design

With the second design, the compiler knows if a function call involves any processor switch. It could generate all necessary code to hide all the details. The programmer then could write the program as shown in Figure 3.4. It has the following differences from the first design:

1. Only one source file, `foobar.tig`, is needed. It includes definitions of both the PPU function and the SPU function. In this case, the programmer is required to provide the target architecture tag for the function.

2. No implementation of program loader, data transfer management and processor switch code is needed. Based on the tags of functions, the compiler has enough information to generate these codes automatically.

3. A single compilation is required to build the whole program, which will generate two different binaries.

Comparing these two designs, it is obvious that the second one not only allows better modularization because function definitions with different target platforms can

Source Files:

```
foobar.tig:
ppu_function foo() :=
  bar()

spu_function bar() :=
  ;; implementations of bar
```

Compilation Procedure:

```
tigc foobar.tig
```

Figure 3.4: Example Program of the Second Design

appear in the same source file, but also provides more opportunities for optimization. In the first design, `bar.tig` that contains the SPU code is not processed by the PPU compiler, so it will be difficult to analyze the SPU functions when processing the PPU code. In addition, function `foo` does not call the SPU function `bar` directly. Instead, it calls some SDK functions (such as `spe_program_load` and `spe_context_run`) to start the SPE program. For the PPU compiler, all the SPU calls end up with calling these functions. Detecting the actual target function will require a significant amount of work.

However, in the second design, all source codes (in this case `foobar.tig`) will be processed by a single compiler. It knows all the function definitions for code analysis. The SPU function is also called directly from the PPU function. After the frontend translates the source code into IR, the SPU call could still be represented as a simple function call node (although it may have some special flag). The compiler can easily identify the target function for optimization use. Finally, the compiler that generates the code for the processor switch could analyze both the PPU and the SPU code to do more optimizations. More detail will be discussed in Chapter 5.

Because of the above reason, TigC uses the second design where a single compiler program is created to process all the input source code. As a variation of the standard Tiger compiler, TigC implements all of the common steps to generate the target executable file [1]. It also has some special operations to support dynamic architecture selection:

1. **Function Labeling**

   To support both PPE and SPE architectures, each function defined in the source files needs to be labeled. This is done through extended keywords which specify the function type. When parsing the function definitions, the function-type token will be read as a tag and attached in the abstract-syntax representation.

2. **Stub Generation**

   With the tags of functions in the abstract-syntax representation, the compiler can analyze and build the control flow graph for the program. For all function calls across different processors, it will generate stub functions to hide all the details of context switch and arguments/return-value transfer. A special stub function will also be created if the target function is required in parallel mode. In addition, the entry function will be generated with a map of all SPE functions, so that all of them can be grouped into the same SPE binary.

3. **Automated Link**

   The code generator in TigC produces assembly language that is a textual representation of the target machine code. In this step, the assembler and linker will group the code and translate it into the binary representation for both PPE and SPE. Addresses of variables and functions are determined in this step. All required libraries, such as stdc and Cell B.E. SDK will be linked into the binary. Then, the SPE executable file will be embedded into the PPE binary to produce the final output.

3.3 TigC Architecture

The general structure of TigC is shown in Figure 3.5.

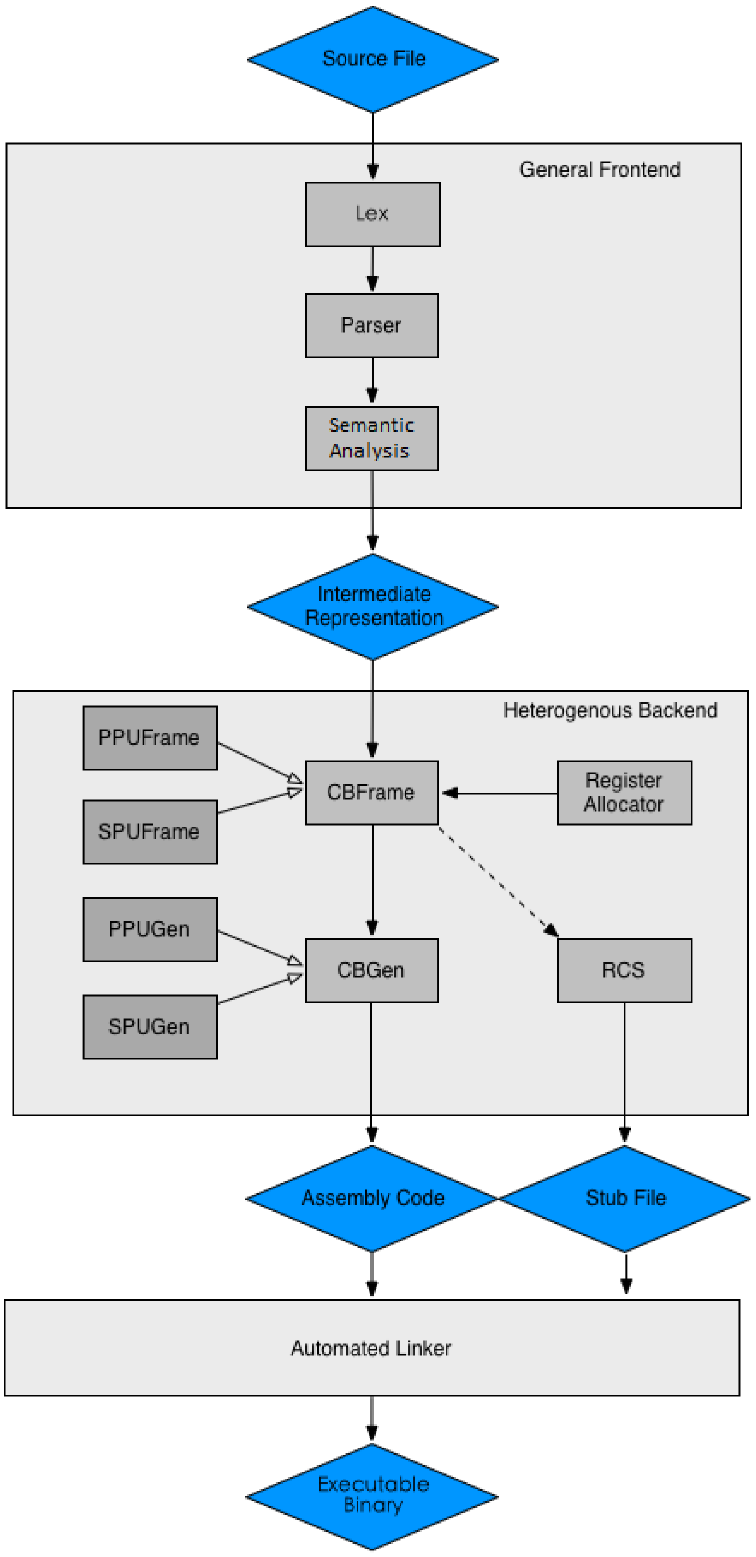


Figure 3.5: TigC Structure

Based on the compilation steps described in the previous section, it contains four subsystems: The General Frontend, Heterogeneous Backend, RCS Generator and Automated Linker. The General Frontend accepts source files as input and translates them into intermediate representation. Then, based on the tag of each function, a different backend is chosen to emit the machine code. With different configurations, they can share the same sets of utility classes, such as the register allocator. An RCS generator reads the function definitions and their targeting architecture types that are generated by the backend, creating all the needed stub functions for hiding all the context switch details. Finally, the automated linker combines everything together and produces the executable binary. Each of these components will be characterized in detail in the next section.

Usually, a compiler is built with the separate compilation design, where the backend is fixed to work on one specific hardware platform. For example, *spu-gcc* mentioned earlier, can be used only to generate binaries for the SPU architecture. To support different backends during compilation and to dynamically switch between them based on given function type, TigC employs the unified compilation design and creates a group of abstract interfaces for the Cell B.E. frame and code generator. Different implementations of these interfaces describe the characteristics of target architectures, such as word size, system instructions, and stack layout. The frontend will pass the tag to the backend in order to employ the corresponding architecture.

The compiler is structured in such way that it can be easily extended later to support different types of platforms. First, some keywords are required to specify the function types. The frame and code generator must be rewritten to have the stack layout and the instruction set for each new platform. RCS needs to be extended as the stub function varies in different systems. In other words, this structure provides a unified programming interface for different AMP architectures.

3.4 Summary

This chapter first discussed the operating environment for the TigC compiler, because TigC has dependencies on various tools and libraries. The resulting binary also requires a Cell B.E. to run correctly. Then it described the steps of compilation used in TigC. In addition to the standard procedure, compilation involves several extensions to support different architectures. Finally, this chapter illustrated the general structure of TigC and how it can be extended to support multiple platforms.

4. Implementation Details

The following chapter discusses the implementation details of TigC. First it describes the general frontend of the compiler which accepts the mixed source code and translates it into an abstract syntax tree with proper function tags. Then, different backends are presented for both PPE and SPE architectures to generate the assembly code. After that, there follows a detailed description of the RCS generator that populates the stub functions and other necessary code for heterogeneous function calls. Then, the automated linker script is introduced that is used to generate the final binary. Finally, the optimization for data transfer between the main storage domain and a local storage domain is presented.

4.1 General Front-end

As described in the previous section, a frontend includes three major steps to validate the input source code and to check whether it is correct in terms of the language syntax and semantics. TigC accepts Tiger source codes for two architectures, but operations in this step do not involve any architecture-specific information, so the frontend is designed as a general component.

TigC uses ML-lex and ML-yacc to simplify the scanner and parser for lexical analysis and syntax parsing. This practice makes it possible to use regular expressions and a context-free grammar to describe the Tiger language without focusing on the implementation details. Figure 4.1 shows the rules governing how the source code is parsed into tokens.

Lexical rules are specified with regular expressions. In this example, some defini-

```
digit = [0-9];
alpha=[a-zA-Z];
alnum=[a-zA-Z0-9_];

"+"        => (PLUS(yypos,yypos+size yytext));
"-"        => (MINUS(yypos,yypos+ size yytext));
","        => (COMMA(yypos,yypos+ size yytext));
":"        => (COLON(yypos,yypos+ size yytext));
"("        => (LPAREN(yypos,yypos+ size yytext));
")"        => (RPAREN(yypos,yypos+ size yytext));

var        => (VAR(yypos,yypos+ size yytext));
while      => (WHILE(yypos,yypos+ size yytext));
for        => (FOR(yypos,yypos+ size yytext));
function   => (FUNCTION(yypos,yypos+ size yytext));

({alpha}{alnum}*) => (ID(yytext,yypos,yypos+size yytext));
({digit}{digit}*) => (case Int.fromString yytext of
                      NONE => (ErrorMsg.error yypos
                              ;continue())
                    | SOME z = >INT(z,yypos,yypos+size yytext));
```

Figure 4.1: TigC Lexical Analysis

tions such as `digit`, `alpha` and `alnum` are first created to make the rules clear; then, rules describe how a given input string is translated into a token. For example, a single character "+" is marked as token type `PLUS`, and word `var` is marked as token type `VAR`. Some rules are more than simple matches. Strings starting with `alpha` and following with any number of `alphanum` will be treated as `ID`, while one or more digits represent a token of type `INT`. The location of each token is also noted together with the token type in the rule so that any error may be reported with friendly information, such as the line number in the source file.

Following the above step, BNF-style rules are applied to describe the grammar. Figure 4.2 shows the rules that form the result tokens into the abstract syntax tree. First, all names required by these rules are listed. They could be either terminal symbols, which are tokens recognized by the lexical analyzer, or nonterminal symbols, which could be expanded and recognized by the parser. The rules are then defined to describe an allowable structure in TigC and to associate it with a nonterminal symbol. The right parts of these rules are called actions, and these actions describe how the abstract syntax tree is formed. For example, a token of type `ID` matches the rule and could be recognized as a var and then matched as an exp. An exp could be expanded in many ways. It could be a single var, a token of type INT, or some other type of recursive definition. For example, given a token sequence ID PLUS ID MINUS INT, ID matches the rule, so it represents an exp, so ID PLUS ID becomes exp PLUS exp; it then forms a new exp. After this step, exp MINUS INT could be transformed into exp MINUS exp and thus form another exp. This outcome means that the given tokens form a correct construct consistent with the grammar; it will be accepted by TigC.

```
%term ID of string
    | INT of int
    | PLUS | MINUS | WHILE | FOR | FUNCTION | VAR
%nonterm program of A.exp
       | exp of A.exp
program : exp               (exp)
var : ID                    (A.SimpleVar(S.symbol ID,IDleft))
exp : var                   (A.VarExp var)
    | INT                   (A.IntExp INT)
    | exp ASSIGN exp        (A.AssignExp { var = var, exp = exp,
                                     pos = ASSIGNleft })
    | exp PLUS exp          (A.OpExp { left = exp1,
                                       oper = A.PlusOp,
                                       right = exp2,
                                       pos = PLUSleft })
    | exp MINUS exp         (A.OpExp { left = exp1,
                                       oper = A.MinusOp,
                                       right = exp2,
                                       pos = MINUSleft })
    | ID LPAREN args RPAREN (A.CallExp { func = S.symbol ID,
                                         args = args,
                                         pos = LPARENleft })
    | WHILE exp DO exp (A.WhileExp { test = exp1, body = exp2,
                                     pos = WHILEleft})
```

Figure 4.2: TigC Syntax Parsing

In TigC, function is reserved as a keyword and translated into a token of type FUNCTION. The grammar shown in Figure 4.3 is used to define a function where fields describes the parameters for this function, which could be empty, a single ID, or multiple IDs separated with commas. optty designates the return type that is potentially omitted, and exp will be expanded as expressions to form the function body.

```
tann : COLON ID               (S.symbol ID)
field : ID tann               ({ name = S.symbol ID,
                                 escape = ref true,
                                 typ = tann,
                                 pos = tannleft })
fields : (* empty *)           ([])
       | field'                (field')
field' : field                 ([field])
       | field COMMA field'    (field :: field')
optty : (* empty *)            (NONE)
      | tann                   (SOME(tann, tannleft))

fdec : FUNCTION ID LPAREN fields RPAREN optty EQ exp (
                             { name = S.symbol ID,
                               params = fields,
                               result = optty,
                               body = exp,
                               pos = FUNCTIONleft})
```

Figure 4.3: Function Definition in Tiger Language

To support different architectures in TigC, the definition of each function should be labeled with a tag to specify its target platform, either by the programmer or by TigC. This condition means there exist two ways to achieve the goal:

1. Rely on the programmer to explicitly specify the target platform and also the execution mode (simple or parallel call) for each function, with the compiler remembering this information and using different backends to generate the assembly code.

2. Implement some logic to automatically decide the target platform for each function. This action requires the analysis of the calls to decide which function would be better moved to SPE context. The compiler also needs to decide how the execution and data should be distributed to multiple SPUs.

With the first approach, the compiler hides the details of function calls across architectures. Programmers need only to decide how the execution should be distributed and to assign correct tags to each function. The second approach improves this situation by analyzing the whole program to find the optimal result binary. By optimal, it means the compiler generates a binary where maximum parts of the computation could be distributed to different SPUs and run in parallel. As described before, this approach usually comes with loop optimization [32, 53, 22, 54]. The implementation involves the following steps:

1. Locate all the loops in a function by analyzing its control flow.
2. Check all variables used in this loop to see if it carries across iteration dependency. Cross iteration dependency means that during each iteration, the value of a variable relies on the output of the previous iteration.
3. Consider the current loop as a candidate for SPU function if no cross iteration dependency is found.
4. Evaluate the loop body and estimate if any performance improvement can be gained if it is implemented as an SPU function.
5. Transform the loop body to an SPU function, and change the original function to make a function call instead of entering a loop.

The implementation of the second approach in TigC requires added work because calling SPU functions also needs some time. Distributing a simple loop may take longer than running it on the PPU. This circumstance means that in step 4, the compiler should make the decision either based on some knowledge of how long an instruction usually takes to execute or by running a sample code on the platform to get the actual time cost.

As this research focuses on evaluating the performance gained from the Cell B.E. system, not on finding the optimal binary for a given work, the first approach is employed in the current version; choosing such an approach relieves us of implementing overly complex logic. The Tiger language is extended by adding a few more keywords: `ppu_function`, `spu_function`, and the original keyword `function` is removed so that users cannot use it again. A function now in the input source file must be defined either as a PPU function or as an SPU function. The rules for the new grammar to define a function are shown in Figure 4.4. Both of these definitions share the same abstract syntax tree node, and the only difference is the additional tag added, which describes the target platform. This measure ensures that the same code can be shared to parse both types of functions. The tags will be used later in the backend.

```
fdec : SPU_FUNCTION ID LPAREN fields RPAREN optty EQ exp (
            { name = S.symbol ID,
              params = fields,
              result = optty,
              body = exp,
              pos = SPU_FUNCTIONleft,
              tag = A.SPU_Func })
     | PPU_FUNCTION ID LPAREN fields RPAREN optty EQ exp (
            { name = S.symbol ID,
              params = fields,
              result = optty,
              body = exp,
              pos = PPU_FUNCTIONleft,
              tag = A.PPU_Func })
```

Figure 4.4: Extended Function Definition

Another extension for Tiger is accomplished by supporting the parallel execution mode with the new keyword `spu_for` and the grammar defined in Figure 4.5. It requires the name of the function that should run on the SPU, along with an input array as its arguments. To split and distribute the input array to different SPUs, some additional parameters are required: programmers need to provide the total length of the array and the amount of data to be processed in the course of each iteration of the loop on the SPE. Differing from ordinary function calls, this extension will be translated into a separate abstract syntax tree node. One example of using `spu_for` is shown below:

```
spufor spufunc of (data, 128, 1024)
```

where a given array `data` of size `1024` will be split into small groups that have a size of `128`. These groups will be distributed to different SPUs and processed by the SPU function `spufunc`. For example, it could be a function which calculates the multiplication results of two 8×8 matrices. The arguments could be dynamically determined

- for example, with an expression of a variable. With this current implementation, `spufor` can be employed only in a PPU function.

```
  SPUFOR ID OF LPAREN args RPAREN (A.SpuForExp { func = S.symbol ID,
                                                 args = args,
                                                 pos = LPARENleft })
```

Figure 4.5: Parallel Call Grammar

With these extensions, the general frontend will generate an abstract syntax tree with enough information about SPE usage.

4.2 Heterogeneous Backend

This section describes the facility to translate the abstract-syntax tree produced by the general frontend into machine code for the PPE and the SPE architectures. As mentioned earlier, IR constitutes another layer before the actual system instruction is generated. In TigC, IR is another type of tree code that is at a lower level compared with abstract-syntax trees. It contains basic data movement, branch operations, binary operations, and comparisons. For example, during the translation from abstract syntax tree to IR, the rules used for an add operation of two variables in the memory is shown in Figure 4.6. It defines a binary operation PLUS for addition and then an expression standing for a memory access with the given offset. The figure also shows the result IR after the translation.

```
      exp = BINOP of binop * exp * exp
      | MEM of exp
      binop = PLUS

      R.BINOP (R.MEM off1, R.MEM off2)
```

Figure 4.6: Example of IR

One atypical action during this step is function-call translation because two different types of functions are expected in the source code, but not all types of function calls are supported. As mentioned, the primary usage of Cell B.E. is to distribute computation work to different SPUs, so it will be rare for an SPU function to call a PPU function. Table 4.1 describes all possible function calls in TigC. When the function call is within the same processor, R.CALL will be emitted, and calling from the PPU to an SPU will emit R.SPUCALL. Note that function calls between different SPUs are not supported in this implementation. A parallel-mode call will be translated into a separate IR node named R.PARCALL.

Source	Target	Call Type
PPU	PPU	R.CALL
PPU	SPU	R.SPUCALL
SPU	SPU	R.CALL
SPU	PPU	N/A

Table 4.1: Function Call Types in Tiger

Although IR is not strictly bound to the architecture and both the PPU and the SPU backends share the same format, it does requires specific information to express correct operations. Accessing variables in a function requires a memory read at an offset in the frame. It means that this step needs to apply the Application Binary Interface (ABI) for each of the architectures. All other components are still required to have less knowledge about the heterogeneous implementations, so an abstract interface is created to represent all of the properties and operations shared between the two architectures.

The translator will query the function tag and instantiate an actual frame. The primary information included in the frame component includes:

1. *Word Size*

   Word is the natural unit used in the system instructions and its size varies on different platforms. Usually, the `int` type in C language has the same size of a word on a platform. In addition, most of the processor registers have the same size as a word. The SPE uses 16 bytes as a basic word, while the PPE uses 8 bytes. This value is important for memory operations, especially when calculating the variable locations in memory and in transferring data between different architectures. For example, an array is represented as a collection of `int` values in memory, so an object on the PPU cannot be copied to local storage and processed by the SPU directly because each time the SPU accesses the i^{th} element, the memory address is calculated as $(i + 1) \times 16$; on the PPU, the correct address should be $(i + 1) \times 8$.

2. *Argument Passing*

   This step defines how each argument is passed to the target function. Each time a function is called, its arguments must be assigned to the corresponding parameters. The arguments can be passed through a stack or, to improve the performance, passed in registers. Then, it follows how TigC differentiates and generates correct load/save instructions for these two styles in the implementation.

3. *Stack Frame Layout*

   This is another definition in the ABI. When creating a new function frame, it describes the locations of all data required for the execution of a function, such as the return address, arguments, and local variables. The compiler needs to generate the same format so that the executable file is compatible with results from other languages.

4. *Register Usage*

   ABI describes conventions for the ways registers are used during execution, including rules about changes of register values, argument passing, and some system registers for frame setup. For example, a platform may include several dedicated registers, such as a *stack pointer (SP)* which points to the top of the stack associated with the current call. Sometimes, there is another register named *base pointer (BP)* or *frame pointer (FP)*, which is used to make the relative addressing easier for arguments and local variables.

Note that even when all the source files are compiled by TigC and there is no plan to interact with programs written in other languages, it is still necessary for the result executable file to be compatible with the ABI because TigC has dependencies on certain libraries implemented in other languages and generated by other compilers. For example, it has a library written in the C language to provide the ability to allocate array objects and to implement built-in functions, such as converting between strings and integers. So, even though all the users' source codes are processed by TigC, employing an array or calling the built-in functions still need to interact with binaries generated by other compilers.

After translation, assembly code is generated from IR. The compiler selects proper instructions for each IR node and assigns each temporary to a register. These steps are examined in detail.

4.2.1 Argument Passing

A parameter refers to the special variable that represents one of the pieces of data provided as the input of the target function. These pieces of data are called arguments. Passing arguments means assigning one argument with data to the corresponding parameter. This section will describe how this action is implemented in TigC.

In most modern architectures, including both the PPU and the SPU, an argument may be passed to the called function by one of the following two methods:

1. *Passing on Stack*

   Most systems designed in the 1970s have all of their function arguments passed on the stack. It reserves a space, either statically or dynamically, and saves the values of the arguments in that area. The address of this area is saved somewhere, usually in a dedicated register, such as *SP*, so that the callee can read it and then access the arguments.

2. *Passing in Registers*

   Research shows that few functions in practice have more than four arguments. Therefore, modern architectures have the calling conventions that the first k

arguments of a function are passed in registers, while any remaining ones are still passed on the stack. This practice saves the unnecessary memory access for the arguments under some cases, such as a leaf function that makes no calls.

Both the PPU and the SPU architectures utilize a stack area and some registers for argument passing, which will be described in detail in the following sections. However, there are some parameters, for which, although their indices are less than k, the arguments cannot be passed in registers. For example, the callee needs to get the memory address of the argument, so the argument must be stored in the memory because variables in registers do not have addresses. In TigC, a function-specific list called `formals` is maintained where each element represents whether the corresponding parameter should be passed on stack or in register. Arguments are passed by registers by marking the corresponding flag as false whenever it is possible and if there is still a register available. TigC later generates instructions based on this list in order to save the arguments to the correct places.

4.2.2 Stack Frame Layout

A stack frame represents a function call during runtime. It keeps track of the point to which each function should return and also records the values used during the execution of the callee function. As the name states, these data are organized in an area of the process stack memory, which is pointed by a special register *SP*. The calling function saves the execution state to the stack and then jumps to the target function. When finished, the callee function recovers the original state from the stack and jumps back to the calling function. *SP* will be adjusted before and after the function is executed so that the stack is always balanced. The stack frame usually includes the following data to describe the function call:

- *BackChain*

  This specifies the address of the previous stack frame.

- *Link Register*

This is the value of the return address which is used when function finishes.

- *Saved Registers*

  Values of some registers cannot be changed during the execution. This area is used to store their values so they can be recovered after the function finishes.

- *Local Variables*

  This area represents all the local variables that cannot fit into the register file in the function.

The way such data is organized in the memory is defined as the stack frame layout, which is described in the ABI specification of an architecture. The standard layout for a PPU is shown in Figure 4.7 [24]. The *backchain* and *link register* are stored as the first two words, and the function parameters and local variables are stored right behind them. The stack grows downward from a high address, and the stack pointer must maintain quadword alignment, so padding may be required in a PPU stack frame. Note that the *backchain* at the top belongs to the parent's stack frame.

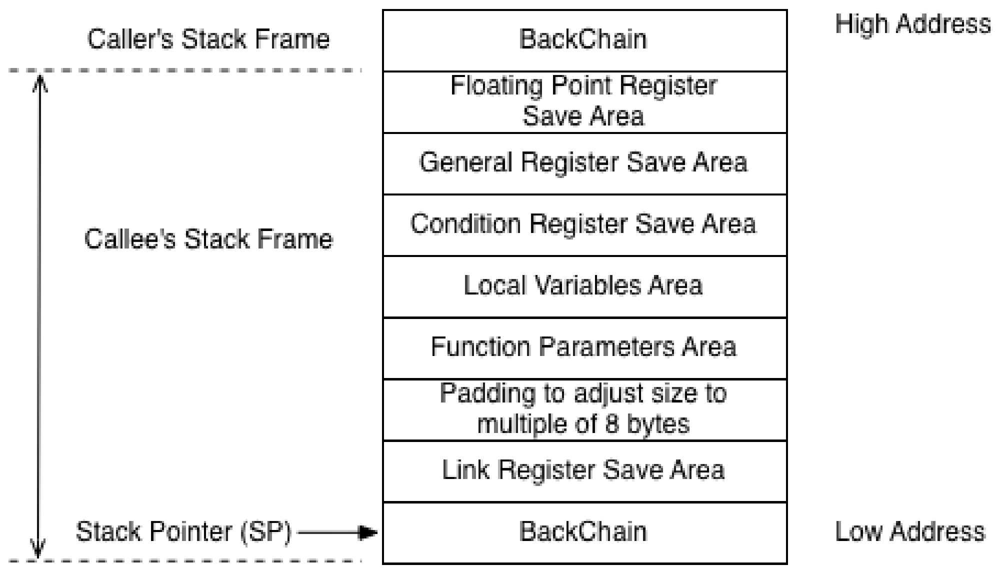


Figure 4.7: PPU Stack Frame Layout

Figure 4.8 illustrates the stack frame layout of the SPU architecture [26]. The first two words are used to store the backchain and the link register. The *arguments* *save area* section starts from offset 32 bytes from the stack pointer, right above the link register, and is followed by the local variable space and the register save area. The stack pointer must maintain 16-byte alignment and must always point to the backchain word. The backchain word, appearing as the first data in the frame, must always point to the previous stack frame, except for the first stack frame (which must have value NULL).

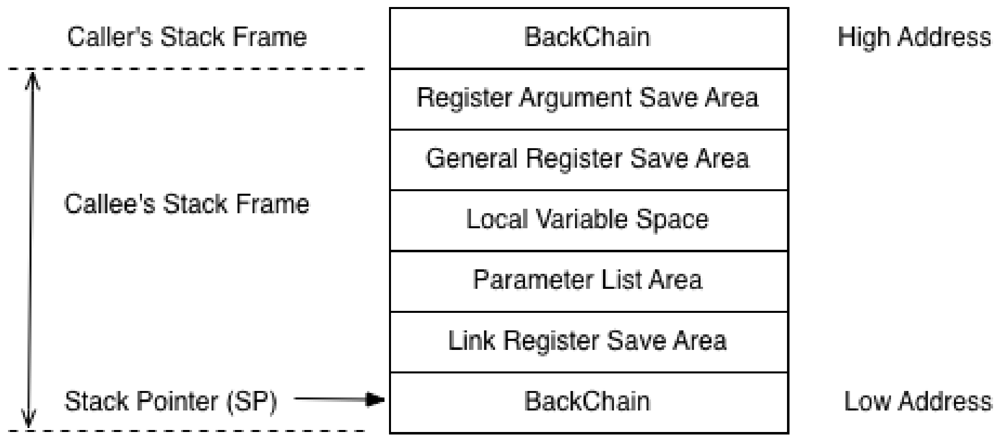


Figure 4.8: SPU Stack Frame Layout

As mentioned, the stack frame layout describes the data organization, and the compiler should follow it for generating the necessary instructions to correctly set up the stack frame during runtime. In TigC, this procedure is implemented as a series of `procEntryExit` functions. These functions generate small pieces of code named `prolog` and `epilog`, which are used to set up the new stack frame at the beginning and to recover the calling function's state at the end. When a new frame is created, escape analysis is used to determine the scope of all the variables. A variable that is passed by reference escapes and it should be allocated in stack. This analysis generates the list `formals`. which will be applied to check whether a parameter needs to be stored in the stack instead of being passed through the register. It then generates a list of all the values that need to be saved before the call for argument passing and runtime recovery. The `procEntryExit` functions translate this list to instructions that store the values in proper locations. This means that, based on the *SP* of a current frame, the compiler needs to calculate the correct memory address for each argument that is passed in stack to generate the correct store instruction. Depending on the architecture type, this calculation can represent one of the following cases:

1. The architecture includes a register to store the *FP*, and each parameter could be accessed via *FP*. One example is MIPS, and its stack frame layout[36] is shown in Figure 4.9. The parameter build area exists in the caller's stack frame while *FP* is set to the start address of the callee's stack frame. Consequently, the calling function pushes each parameter into the stack in its own frame at the address to which SP points. No extra calculation is needed to generate these instructions. Later, the callee can access each parameter by using $FP + \mathit{offset}$.

2. On some platforms, such as the PPU and the SPU, FP is not available, and the parameters are stored in the callee's stack frame. When passing escaped parameters, the calling function must know the memory address where the value should be stored, which is represented by an offset to the caller's SP. However, at this time, the callee's stack frame remains not fully defined, and its size is still unknown. For example, the compiler does not know the size of the local variables area. Only the offset of each parameter to the callee's SP is saved.

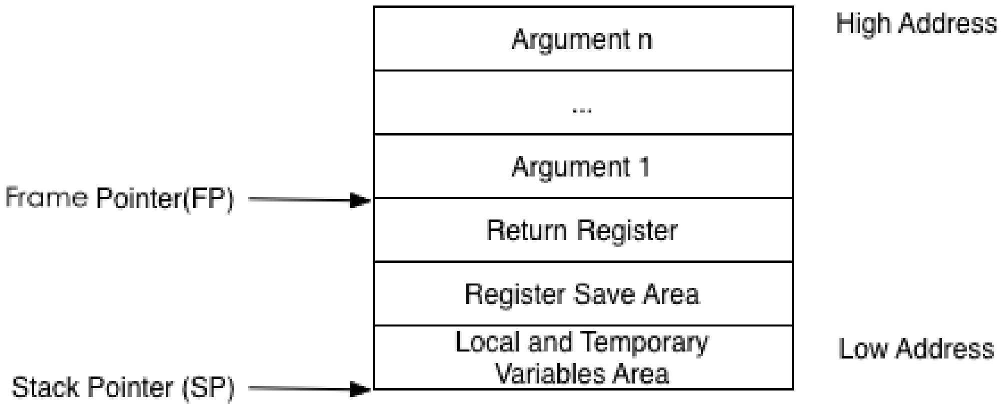


Figure 4.9: MIPS Stack Frame Layout

Because the PPU and the SPU frames do not provide a *FP* for data access, the calling function needs to store the parameters' value to the callee's stack frame without knowing its stack pointer because data are still being pushed to the stack. To calculate the correct locations for each parameter, during the compilation, a variable *fsize* is used to represent the callee's frame size; this dimension is incremented by a word size every time an instruction is generated to push an item into the stack for the callee. Meanwhile, whenever a parameter is pushed to the stack, the value of *fsize* refers its offset to the callee's *SP* because the stack frame is built bottom-up. Finally, the compiler calculates the offset of a parameter to the caller's *SP* by applying the following equation:

$$offset_{parameter} = fsize - offset'_{parameter}$$

where *offset* refers to the parameter offset to caller's *SP*, and *offset'* refers to the parameter offset to callee's *SP*.

Tiger also supports nested function declarations, which means that the nested function can access local variables defined in its parent function. This process requires a *static link*. However, the frame module should be independent of the specific source language being compiled. Suppose a function f has k ordinary parameters: let l be the formal list specifying whether the parameter is in stack or register whose size will be k. Then during the translation, a new list l' will be created with an extra *true* in front of l to represent the *static link*; thus its value can be retrieved in the same way as other parameters.

Finally, there is a piece of code in Tiger language that does not belong to any function. This piece will serve as the entry code and will be executed at the very beginning. In TigC, this piece of code is considered to be defined in a special function called `tigermain`, so a stack frame should also be created before the execution. Because this is the first function to be called, there is no `backchain`, and no register values need to be saved. When generating the `procEntryExit` functions, the function name is checked for generating different instructions for `tigermain`.

4.2.3 Register Usage

A register is a small amount of the storage unit in the processor and can be accessed quickly. Almost all architectures will first load the data from memory to registers for later manipulation; they then store the result value back to memory. However, registers are limited resources, and each frame will need to share the same register space, so it is important to define the consistent usage of each register.

Certain registers have special usage, and they are named dedicated registers. Their values must remain unchanged by the callee. Depending on how the register values are saved and restored during a function call, non-dedicated registers can be divided into two categories:

1. *Volatile Registers* (also named as caller-save registers) contain values that can be lost when sub-routines are called. Therefore, if the calling function wants to maintain these values, it must save the registers explicitly in the stack frame or in another temporary unused register.

2. *Non-volatile Registers* (also named as callee-save registers) contain values that must remain unchanged if subroutines are called. This stipulation means that the callee must save the values to the stack before touching them. If this action does not alter that register, no save/restore instructions will be generated.

TigC depends on the Frame module to provide the list of volatile registers so that when calling a function, these registers will be stored together with the creation of the new frame. The frame also describes the ABI conventions for registers in the platform, such as those governing which registers are used to pass parameters to functions and to return values back to the caller.

As a standard 64-bit PowerPC architecture, the PPU provides 32 general-purpose registers, each 64-bits wide. It also contains 32 floating-point registers of the same size. As TigC supports only integer operations, these floating-point registers will be ignored here. The processor also includes some special registers to control the program execution, such as the *program counter*, which points to the current instruction to be executed. The condition code register is also not described because in current implementation, only *cr0* is used for branch operation. All the other registers in the PPU are listed in Table 4.2. Some special registers are described here:

- *r0* used in the function prolog. In 64-bit PowerPC, the prolog usually includes an instruction to move the return address from the *link register* into *r0*. This will be used to restore the return address at the end.

- *r2* for Table of Contents (TOC) pointer. The code could be put anywhere in memory, so it must be position-independent while still read-only. This state is implemented by using a relative address in instructions. However, this has some drawbacks. Accessing a local variable works fine, but the compiler cannot calculate the offset of a global variable or a library function because they may exist in other modules or be loaded dynamically during runtime. TOC solves this issue by creating a table for the application and each import library. It includes pointers the code uses to locate the static data or external functions.

- *r11* for environment pointer. Some languages require a pointer for local variables, and *r11* should be used for this purpose. TigC does not use environment pointer, so *r11* appears as a normal volatile register.

- *r12* for exception handling. In languages that support exceptions, if an exception occurs, the execution stops, and the control passes to the exception handler. Meanwhile, all objects allocated on the stack will be destroyed in the reverse order of allocation. *r12* is employed to support this process.

- *ctr* for loop counter. This can hold the loop count, which is decremented automatically to stop the loop. The other usage is to provide branch target address for a branch conditional to count register instructions.

As mentioned earlier, the SPU has a large register file in order to achieve the greatest performance. It has 128 general-purpose registers, each having a 128-bit

Abbreviation	Type	Purpose
r0	Volatile	Used in function prologs
r1	Dedicated	Stack frame pointer
r2	Dedicated	TOC pointer
r3	Volatile	Parameter and return value
r4 - r10	Volatile	For parameters
r11	Volatile	Environment pointer if required
r12	Volatile	For exception handling
r13	Dedicated	Reserved to system thread ID
r14-r31	Non-volatile	For local variables
lr	volatile	Link register
ctr	volatile	Loop counter register

Table 4.2: PPU Registers

width. Although an SPU treats all these registers in the same way, the ABI still specifies conventions. Table 4.3 shows these conventions.

Abbreviation	Type	Purpose
r0	Dedicated	Link register
r1	Dedicated	Stack frame pointer
r2	Volatile	Environment pointer if required
r3	Volatile	Parameter and return value
r4 - r74	Volatile	For parameters
r75 - r79	Volatile	
r80 - r127	Non-volatile	For local variables

Table 4.3: SPU Registers

4.2.4 Instruction Selection

Although relatively low-level, IR is still an intermediate level representation that abstracts over processors and must be transformed into machine-specific instructions. In this step, IR is translated into proper assembly code with the least number of instructions or the lowest total execution time.

The implementation of instruction selection is based on the *Maximal Munch*[1] algorithm. It works by converting IR with tiles containing one or more assembly language instructions that perform that IR operation. Optimal tiling is applied to make sure that there are no two adjacent tiles which can be used to combine into a single tile with lower cost. The implementation starts at the root of the tree and tries to find the largest tile that fits, then continuing recursively to apply the algorithm to the IR subtree. Here, the largest tile means the one that covers the largest number of nodes in the tree. Note that the optimal tiling is only local and does not guarantee the lowest cost. This situation is fine because a compiler usually has optimizations such as constant propagation, which introduces more chances for instruction selection. A global optimal result at this step may still require changes later.

Here is an example for this algorithm, given the following source code:

```
a := exp
```

where `exp` represents a certain expression and its result is stored in `t2`, `t1` represents the stack pointer, and `a` has an offset `i` from the stack pointer, then the following IR node will be generated:

```
R.MOVE(R.MEM(R.BINOP(R.PLUS, t1, R.CONST i)), t2)}
```

When generating the assembly code for the SPU architecture, it can be a quad-word (128-bit) **store** instruction that covers **MOVE**, then a quad-word **load** instruction to cover the node **MEM**, followed by an **add** instruction which covers the remaining binary operation subtree:

```
addi t', t1, i
lqd t'', t'
stqd t2, (t'')
```

Alternatively, the compiler could emit it as a single quad-word **store** instruction **stqd**, which will cover the whole IR node[27] (the largest tile).

```
stqd t2, i(t1)
```

According to the Maximal Munch algorithm, the second option would be the better choice. The implementation of this algorithm in TigC is straightforward. Two recursive matching functions are provided: `munchStm` for the statements and `munchExp` for all the expressions, wherein each clause will match one tile. Since SML pattern-matching always chooses the first rule that matches, the rules are ordered so that the biggest tile always comes first. TigC implements two assembly code generators: one for the PPU and one for the SPU.

4.3 RCS Generator

The output from the compiler backend contains valid assembly code that can be used to build the executable program. However, until this time, one important piece of information is missing about the function calls. As described, for calls within the same architecture, an R.CALL node will be used to generate the assembly code, the PPU can emit proper branch code based on the function tag. However, for R.SPUCALL and R.PARCALL nodes, no single instruction is available on the Cell B.E. system in order to switch the execution context from the PPU to the SPU. In addition, all the arguments for the function call are stored on the stack (which exists in the main memory) or in registers of the PPU. Programs running on the SPU are restricted to the SPU's local storage and register file and cannot directly access these values. The MFC must transfer the data into the SPU local storage first before the operation can be executed.

First described is the normal procedure on Cell B.E. to run an SPU binary, then the stub functions which help to transfer the execution from the PPU to the SPU. These stubs also prepare all the required parameters in the local storage and retrieve the return values back to main memory. For the PPU functions, calls to an SPU function will be redirected into a local PPU stub function, which makes it similar to calling a local subroutine. The details will be transparent for the programmers, meaning that they do not need to worry about the SPE context initialization and data transfer when writing the code. It follows by one important situation which describes how complicated data structure is passed as parameters to the SPU functions and explains how it is returned to the PPU. Finally, the implementation of the parallel loop call is discussed.

4.3.1 Simple Call Mode

Simple Call Mode refers to the SPU function call from a PPU function that is not to be executed concurrently. This means the program running on the PPU will help load all the parameter data to one SPU's local storage and then trigger the target function. During the execution, the PPU program will stay blocked until the SPU function returns. A simple call is represented as R.SPUCALL in the IR tree.

In a Cell B.E. system, the following steps are required to run a program on the SPU:

1. Create an SPE context. A context represents an SPE execution environment, so a simple call only needs one context.
2. Load the SPU program with the context created. The program will be loaded into the local storage of the corresponding SPU.
3. Run the SPU code by specifying the context and wait until it finishes. This interface allows us to provide some parameters from the PPE program.
4. Destroy the SPE context. This releases the usage of the SPE.

Creating and destroying the SPE context requires time. Repeating these steps for every SPU function call will be inordinately expensive, and the performance will be degraded greatly. The solution is based on the fact that TigC will compile all the SPU functions into a single binary, and during execution, all the SPEs are employed exclusively by this program. Consequently, all the SPE contexts are pre-allocated and reused as necessary. As mentioned, TigC creates a special function `tigermain` for the entry code that gets executed at the beginning of the program. Some extra IR nodes will be injected to initialize the SPE contexts and to load the binary in the entry function, then to clean up when the program exits. Two C functions are defined as `spu_init` and `spu_exit` inside the TigC standard library. Because the `Frame` module includes `procEntryExit` functions emitting code to initialize the stack frame, the

PPUFrame is simply extended by adding the execution of these two functions in the `tigermain` of the PPU program. The assembly code after this change is shown in Figure 4.10. This procedure guarantees that the context is initialized and that the SPU is ready to run.

```
bl spu_init
...
main body
...
bl spu_exit
```

Figure 4.10: Initialization of SPE Contexts

With mixed function types in the source code, any function could be defined to run on the SPU and then get called from a PPU function. One solution occurs during the code generation: the call instruction such as `bl` is replaced with correct instructions to load the program on the SPU and to run the program while waiting for completion. However, this process makes the code generation complicated because all the work needs to be implemented with assembly code. It should not corrupt the stack frame, and the register allocation must be reconsidered. In addition, if the same SPU function gets executed more than once, the same piece of code will be generated multiple times, which increases the resulting binary file size. Thus, all of this work is encapsulated into a function implemented with the C language, which is named a stub function. Calling the SPU function will be simply redirected into this stub function. The code generation is simplified as it still emits a call instruction, only with a different function name. The stub function has the same interface as the target SPU function, allowing it to correctly receive the argument values. To make sure that all of the stubs required are generated, TigC needs to note down every SPU function defined in the program source code and to assign it a unique ID. This unique ID will be used to interact with the SPU program in locating the target function. This ID assignment is completed by remembering the function definitions when parsing all the functions into the abstract-syntax tree. The result includes a list of the functions defined in the source file in the following format:

$$\#FUNC < arch > name(id_1, type_1|id_2, type_2, |...|id_n, type_n)|type_{return}$$

where *arch* is either *PPU* or *SPU*. The $(id, type)$ pair describes the corresponding parameter name and its type; $type_{return}$ represents the return type of this function. This list is stored in a temporary file, and later the RCS system scans the list to build a function map which includes all function names, architecture types, parameters, and return types. For each SPU function in this map, RCS will generate the stub function that has the same parameters and return type. To simplify the implementation, the prefix *spucall_* is added to the original function to form the stub function name. When translating IR trees into assembly code, for any `R.SPUCALL`, the target function is replaced with the stub in the instruction as the destination; therefore, the call is redirected to the stub. Figure 4.11 shows the comparison between R.CALL and R.SPUCALL translations. Note that only the PPU code generator allows R.SPUCALL because, in current TigC implementation, a function running on an SPU cannot make a remote call on another SPU.

```
munchExp (R.CALL (R.NAME f, es)) = emit (
A.OPER { assem="bl " ^ S.name f
   ...
})
munchExp (R.SPUCALL (R.NAME f, es)) = emit (
A.OPER { assem="bl spucall_" ^ S.name f
   ...
})
```

Figure 4.11: Translations of R.CALL and R.SPUCALL

Calling a function requires more than just setting the context and loading the program. All function parameters' data must be moved to the SPU's local storage. In the main memory, for each SPU, 16KB buffer is reserved for input data along with another 16KB buffer to receive the output from the program. Considering that the input data is partitioned into small pieces for distribution, this buffer should be adequate. The size is also defined as a parameter of the compiler and may be expanded easily. Following the initial steps, the address of the input buffer in the main memory is passed as an argument to the SPU program. All function parameters are stored in the input buffer, which can be accessed by the program on the SPU with MFC as its address, since this address is now known. Because the input buffer is operated upon by two different programs, some information is necessary to describe its structure. The format of the input buffer is defined as a structure shown in Figure 4.12 and shared on both platforms.

```
#define PARAM_DATA_SIZE (16 * 1024 - 7 * sizeof(uint64_t))

struct input_buffer_t
{
    /* The index of target function in the map */
    uint64_t index;

    /* The address of the output buffer in main memory */
    uint64_t out_ea;

    /* param_flags and param_size, used for array data */
    uint64_t param_flags;
    uint64_t param_size;

    /* arr_ea and arr_size, used for data transfer optimization */
    uint64_t arr_ea;
    uint64_t arr_size;

    /* Current SPU ID, assigned by the stub function, also used
       for data transfer optimization */
    uint64_t spuid;

    /* Buffer to hold the values for all parameters */
    uint64_t param_data[PARAM_DATA_SIZE];
};
```

Figure 4.12: SPU Program Input Buffer

For a simple SPU call, `index` represents the target function ID from the function map built by the RCS. `out_ea` is the address of the output buffer in the main memory. `param_size` describes how many parameters exist in the input buffer. `param_data` occupies all of the remaining space in the input buffer. RCS queries the function information in the map and copies its parameters to this array. The stub function consists of these steps to prepare all required data and then to run the SPU program. Once the program starts, the entry function will be activated with the address of the input buffer passed from the PPU caller. An MFC read request will be issued to read the input buffer from main memory to the local storage.

As mentioned, in the Cell B.E., an MFC tag must be reserved to represent a channel for the DMA operation. The program can then issue asynchronous read requests with the channel id and the main memory address. In TigC, the following interfaces [28] are used on the SPU to transfer the data:

- *mfc_tag_reserve* reserves a channel for data transfer. All MFC operations will require this tag ID.
- *mfc_get* issues the asynchronous read request on the given address.
- *mfc_write_tag_mask* selects tag groups to be included in query operation.
- *mfc_read_tag_status_all* queries the status tag groups that are selected in the above operation. This will block execution until the data transfer is finished.

MFC supported transfer sizes are 1, 2, 4, 8, 16, or multiples of 16 bytes, so the input/output buffers are initialized with 16-byte alignment to ensure that data will transferred correctly.

Once the input buffer is loaded into an SPU's local storage, the program is ready to execute the target function on the SPU. However, different SPU functions exist in the program, and their interfaces vary since they may have different parameters and return types. The entry function should be able to call all of these functions with the same code path. It is difficult, though, to implement a static entry function able to correctly pass the arguments. For instance, with the sample code shown in Figure 4.13, there are two SPU functions named `spufunc_a` and `spufunc_b`. Given `index` representing the target function, the function pointer can be retrieved from the map. However, because these two functions have different parameters, if only two arguments are passed while the target function is `spufunc_b`, parameter `z` will not be initialized correctly.

```
int spufunc_a(int x, int y) {}

int spufunc_b(int x, int y, int z) {}

int main()
{
    int index = /* read the target function index */;
    spufunc_t funcmaps[] = { spufunc_a, spufunc_b };

    int result = funcmaps[index]( /* parameters */);

    /* rest of the code */
}
```

Figure 4.13: Implementation of SPU Entry Function

To solve this problem, stack frame should be built where the callee function expects the same environment as that called locally. Three different approaches are evaluated to achieve this end:

1. The problem comes from the fact that only one entry function exists but there are different target functions. So, the easiest solution is to have the same number of function call statements in the entry function, one for each SPU function. Instead of having a map and an index, function calls are implemented as a switch statement (as shown in Figure 4.14). This implementation has the advantage of the function call still being a valid C statement, and there is no extra operation to be performed during each call. The only problem occurs if there are too many SPU functions: this switch statement then grows rapidly and makes the entry function very large. Sometimes, this consequence matters because the local storage has limited size.

2. This is actually a variation of the first approach. When calling a function, although it has different definitions than others, it is always converted into the same function pointer type, `spufunc_t`, which has a fixed number of parameters. The values of the these parameters exist in the *param_data* array, and if the function has fewer parameters than `spufunc_t`, some garbage data will be passed to the callee. However, this problem can be ignored, as the target

```
int main()
{
    int index = /* read the target function index */;
    switch (index) {
    case 0:
        spufunc_a(param_data[0], param_data[1]);
        break;
    case 1:
        spufunc_b(param_data[0], param_data[1], param_data[2]);
        break;
    default: error();
    }
    /* rest of the code */
}
```

Figure 4.14: Using Switch in SPU Entry Function

function will never access that data. For example, if `spufunc_t` has the same definition of `spufunc_b`, but the actual target function is `spufunc_a`, the input buffer will contain only the valid values for x and y, so z will be the next value in `param_data` which is never initialized. `spufunc_a` has no knowledge about the third parameter and will simply ignore its value. This situation also means that the definition of `spufunc_t` limits the number of parameters an SPU function can accept at most. If it has the same definition as `spufunc_a`, `spufunc_b` cannot be called with correct arguments. In addition, parameter that cannot fit an integer will not be correctly passed.

This method has the advantage of the entry function being very small because it contains only one function call statement. Only one map is included to remember all the function addresses. Calling different SPU functions will share the same code to initialize the parameters from the input buffer and to handle the return value. As mentioned, the downside of this approach is the that maximum parameter count is limited by the definition of `spufunc_t`.

3. In the SPU, registers $r3$ to $r74$ are volatile and are used to pass arguments, which means that another stub function can be added to assign the parameters to these registers. This function accepts `param_size` and `param_data` as its parameters. Then, starting from $r3$, whenever `param_size` is not 0, it will get the next value

from param_data and assign it to the next available register. After this, it will set up the callee stack frame and branch to its code. Figure 4.15 demonstrates how the parameters are set up. The naive implementation of this approach

```
if (argsize && argsize--)
    asm ("ai $3, %0, 0" :: "r"(param_data[0]));
if (argsize && argsize--)
    asm ("ai $4, %0, 0" :: "r"(param_data[1]));
/* more checks until r74 is assigned */
```

Figure 4.15: Using Assembly in SPU Entry Function

also limits the parameter count to be under 72. Functions requiring additional parameters get the extra values in the stack frame instead of registers, so this approach cannot be used in such cases. It cannot support data structures that are larger than the register size. In addition, for each parameter, two checks are required, which slightly slows down the program. This approach also uses assembly code directly, so more code change is needed when trying to support another type of architecture.

As discussed above, although the second approach has some limitations, it simplifies the implementation and reduces the binary size. So, in this study, TigC employs it to pass the arguments to the target function. When the function returns, an MFC write operation is applied to transfer the return value to the output buffer, and the execution on the SPU is finished, which unblocks the stub function. This approach handles the simple data types as well as arrays which will be described in the following section. The whole procedure is now hidden in the stub function, so the calling function does not need to know anything about the processor switch. For programmers, they can now call SPU functions in the same way as calling other PPU functions.

4.3.2 Passing Array as Parameter

In the implementation of simple call, the function parameters are first copied into the input buffer and then transferred to the SPU local storage via MFC. However, Tiger language supports complicated data structures, such as array, record, and string. Marshalling is required to transform these types into representations that are easier to handle. For example, an array is defined as a pointer to a block of memory with two parts: an array header indicating the count of its elements, followed by the actual array data. Copying the pointer value itself to local storage does not work, for the array data is not moved. The array contents should be copied to local storage and then the pointer value in the parameters should be updated so that it can point to the correct location. This section focuses on the array type, as record and string can be considered as arrays as well.

To differentiate an array from other regular parameters, one bit is used for each parameter in *param_flags*. This appropriate bit will be set if it is an array parameter. Otherwise, it will be cleared. Note that this feature limits the parameter count to be under 64, as*uint64* is used for *param_flags*. The array data is also copied into the same input buffer, so only one MFC transfer is required to pass the parameters during each execution on the SPU. The format of the input buffer before and after the array data get adjusted is shown in Figure 4.16.

In the above example, the first and the third parameters are of the array type; thus, their data get copied into the input buffer right after the parameter values. Meanwhile, the corresponding parameter values are also cleared to zero because the original pointers represent main memory addresses and are no longer applicable. As mentioned earlier, an array has its size as the first element in the buffer, so the second array is stored right after the first array, and the offset of the second array can be calculated easily by adding the first array's offset and its size.

When the entry function of the SPU program is executed, an MFC read will be issued for transferring the input buffer from the main memory to the local storage. Then, the *param_flags* will be inspected. Once a parameter marked as an array type is reached, its value in *param_data* will be ignored. Instead, the actual array address is calculated based on the end address of the parameter values and is incremented after processing each array. This address is selected as the corresponding parameter for the target function.

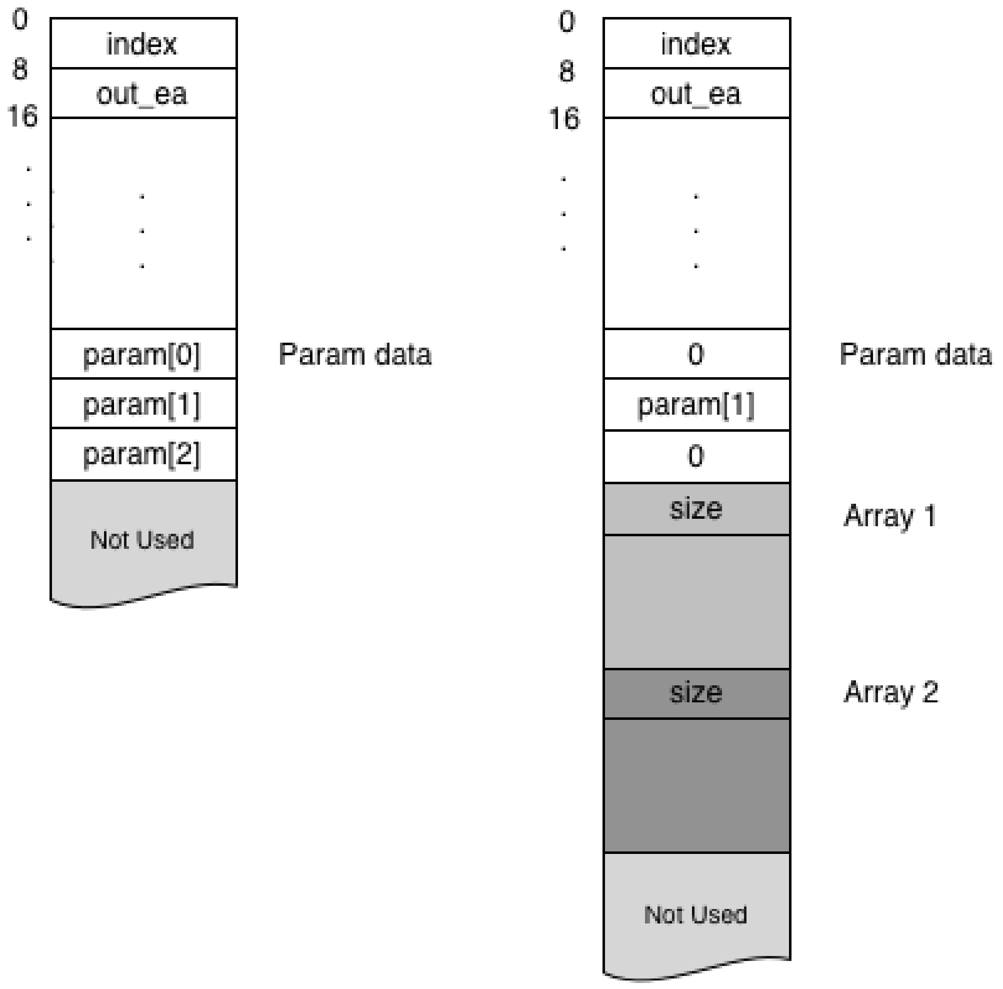


Figure 4.16: Array Data in Input Buffer

4.3.3 Parallel Call Mode

With a simple call, the program will call the target SPU function and then block waiting for its completion. However, this brings extra overhead to prepare the SPE context and transfer the required data. For simple functions, these steps may take longer than the function execution itself. Furthermore, there are eight SPEs in a Cell B.E. system, while the simple call mode utilizes only one of them. The other seven SPEs will stay idle and their power will be wasted. To improve this situation, suppose there is a pseudo code such as

```
map ($p_k \circ p_{k-1} \circ p_{k-2}...p_2 \circ p_1$) [$d_1$, $d_2$, ..., $d_n$]
```

where K sequential operations (p_1, p_2, ..., p_k) are applied on N data elements (d_1, d_2, ..., d_n). Two distribution models are investigated to spread the computational jobs across different SPEs.

1. The first model relies on the assumption that one algorithm usually includes several steps of operations (K in this sample code). Different SPUs will be used to form an execution pipeline where each finishes one exclusive subset of these operations. During each step, the function accepts the resulting data from its previous operation and does its work. Whenever two adjacent steps are running on different SPUs, an MFC transfer is needed to synchronize the result data between their local storages. This model is shown in Figure 4.17.

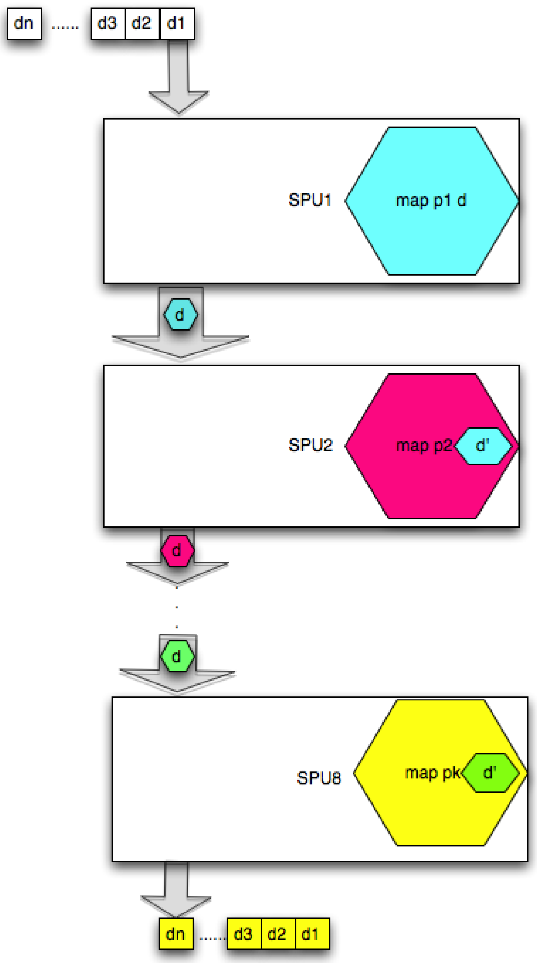


Figure 4.17: Parallel Call by Distributing Tasks

2. For a computation-intensive job, the worker functions usually receive a large amount of input data to process. Therefore, the second model splits the input data into several subsets and then runs all K operations on each SPU with one subset. This requires the input data to be divisible. An example of the second model is shown in Figure 4.18.

   Compared with the first solution, the second one has several advantages. First, it relies on the input data size rather than the number of steps. It may be possible that one task has only one step but still needs to process a large amount of data. Second, the input data is split into independent subsets so that each SPU can run its program without interacting with others, and no communication between the SPUs is required. Third, all the SPUs are always processing the same amount of data, whereas in the first solution, the SPUs may receive steps with different complexity; one may stay idle waiting for the result from the previous step. In TigC, the second approach is employed for parallel processing.

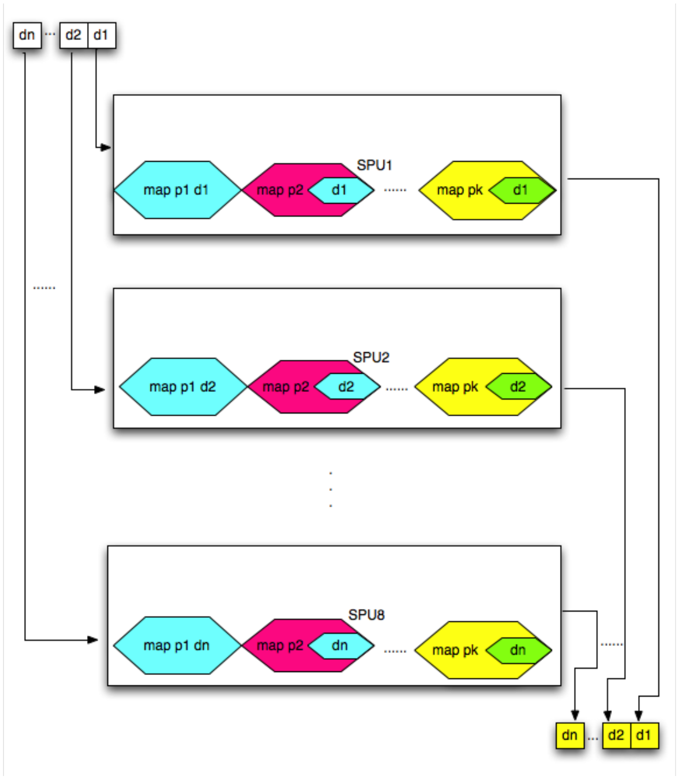


Figure 4.18: Parallel Call by Distributing Data

Recall that simple-call mode is synchronous: after an SPU function is triggered from the program, the stub function will block until the SPU call finishes. To support different SPU calls at the same time, the program needs different threads, one for each SPU execution. The stub function `spu_init` is first modified to support the initialization of multiple SPE contexts. In the main memory, the same amount of input and output buffers are also reserved for these contexts. Each of these threads is then assigned with an index for its target SPE context. When an SPU function is called in parallel mode, it will locate the next unprocessed dataset and pass it to the target SPU. The thread blocks until the SPU call is finished, and the parallel call is considered finished when all threads finish.

When a function is recognized as an `spu_for`, the RCS will generate a new stub function for the above procedure. This new tub function has the same interface as `spu_for` to include the input data, total length and the step size. Similar to the process for a simple-call stub function, the prefix `parcall_` is added to the original function name to form the new stub name. In this implementation, the pthread library is utilized to create the same number of threads as the SPE contexts initialized at the beginning. The input data is divided into datasets with a step size specified by programmer. Each thread uses the following formula to decide the location of the i_{th} dataset:

$$location_i = spu\_id * step\_size + i * spu\_count * step\_size$$

where the *spu_id* is an integer representing the index value assigned for this thread. It should be in the range $[0, spu\_count)$. The thread will loop until the i^{th} dataset goes beyond the total length of the input data. The execution of the SPU function stays the same as in simple call mode, except that each thread uses its own input and output buffer for the SPU to access. The stub function uses *pthread_join* to wait until all threads finish, and then returns.

As illustrated in Figure 4.5, the abstract syntax is extended to include an *SpuForExp* for supporting the parallel call mode. The expression will be translated later into a new IR tree node *R.PARCALL*. Similar to *R.SPUCALL*, it is available only to the PPU code generator. In the last step, Figure 4.19 shows how *spufor* finally is translated to use the new stub function.

```
munchExp (R.PARCALL (R.NAME f, es)) = emit (
A.OPER { assem="bl parcall_" ^ S.name f
  ...
})
```

Figure 4.19: Translations of R.PARCALL

4.3.4 Returning Data from SPU Function

According to the way the stub function works, after the target function finishes its execution, it will return to the entry function of the SPU program. Next, this return value should be transferred back into the stub function generated by RCS, and then returned to the original caller, which is a PPU function. This section describes the implementation of returning the resulting data to the calling function.

When calling an SPU function, either in simple call mode or in parallel call mode, a parameter is passed to the SPU entry function that includes the main memory addresses of the input and output buffers. The goal is to store the return value in the output buffer. This section first describes simple call mode. As mentioned before, in an SPU entry function, all target functions are treated as if they have the same return type, which is a primitive integer. After the target function finishes its execution, this value will be returned to the entry function. It checks the actual return type from the function map, and for scalar types, the value will be saved to the output buffer through an MFC write operation. If the actual return type is an array, this value will be casted into a pointer to integers (`int *`). As mentioned, the first part of the array is a header describing the count of the elements. So the total size of the array could be calculated using the following equation:

$$TotalSize = sizeof(int) + ptr[0] \times sizeof(int)$$

where *ptr* refers to the pointer to the array. An MFC operation will be issued to transfer the whole memory data into the output buffer in the main memory.

For a parallel call, however, returning values from SPU functions are different for two reasons. First, the target function is executed on multiple SPUs with different parts of the input data, so each SPU will return only part of the result. Second, even a single SPU may process different datasets of the input data, and each execution will generate a return value. All of these partial results need to be aggregated to get the final output.

To simplify the implementation, all of the SPU functions employed under parallel call mode must have array as the return type, even if it only returns a single scalar value. Thus, a simple solution utilizes the output buffer reserved in the main memory in order to hold all the return values. An MFC call is issued after processing each dataset. The offset in the output buffer can be calculated based on the SPU ID and the dataset index, so different function calls use exclusive memory regions. However, this process involves more MFC transfers, and although they can interleave with the execution, function calls still take time. In addition, the procedure is based on the assumption that every call on every SPU must return the same size of data, otherwise, program runs on the second SPU need to wait until the first SPU finishes execution to know the start offset in the output buffer; these conditions make the execution run in a sequential way. If the assumption is true, this solution has the advantage that as long as each function call on a single dataset returns smaller data than the MFC transfer limit, an unlimited size of data can be returned to the main memory if the output buffer is large enough. However, the compiler should not rely on this assumption, and should be able to handle all cases.

To solve this problem, the following two changes are made under the parallel call mode:

1. In the main memory, a buffer is reserved for each SPU used in the parallel call mode. This buffer holds only the return values for the function executions on that specific SPU. This exclusivity guarantees that functions running on

different SPUs will not have overlapped buffers.

2. Another temporary buffer is introduced in the local storage to hold all of the results from datasets processed on one SPU. Because datasets on a single SPU are processed sequentially, the result of a function call can be appended to this temporary buffer by using a variable to track the current size of the buffer.

Because the compiler requires the return value as an array, with its length stored as the first value, all the values in the array must be copied to the temporary buffer in the local storage. Note that the length value itself is not copied. During the copy, the first slot is skipped and left unused until after all datasets have been processed. The total length will be stored there in the buffer. Next, a single MFC request is issued to transfer this temporary buffer to the output buffer in the main memory reserved for this SPU. After all of the SPUs finish the execution, the stub function collects the return values in the output buffer. It will create a new array object by concatenating all the data from the output buffers.

4.4 Automated Linker

With the assembly code generated by the PPU and the SPU code generators, TigC now is ready to build the executable binary. However, the assembler can accept instructions only for its target architecture, which requires TigC to provide two separate sets of code. Then together with all required system libraries and stub functions, these source files will be linked into binary files. A Python script is employed to manage all the assembly files and to carry out this linking procedure automatically.

In the backend of TigC, CBFrame contains interface functions to generate the prolog and epilog for each function. This is the point where the stack pointer is adjusted to reserve enough space for the callee's stack frame. Some important registers are also initialized at this time. These two places are modified in both PPUFrame and SPUFrame to mark the beginning and the end of each function by generating some special labels in the assembly file. The automated Linker reads these labels and splits the resulting assembly code into two separate files, after which different assemblers are used to process them.

The linking procedure requires definitions for all functions, either in source or in a library. TigC supports several built-in functions for string manipulations, array creation, and some simple I/O operations. All of these functions are implemented in *libtiger.c* in the C language and loaded by the linker on both platforms. For those stub functions generated by RCS in the C language, another file is created to include them. All of these source files will be compiled and linked together. In addition, the stub functions use SPE context and MFC operations to run functions on the SPU, which requires the Cell B.E. SDK to be linked into the final binaries.

Simply linking everything above together will generate two different executable files, one for the PPU and the other for the SPU. However, there is another way to make SPU calls more transparent for programmers. The system provides some commands so that the SPU binary file can be embed into the PPU binary. When loading the SPU program, the PPU code in *spu_init* accesses the program handle, which is declared with the following line:

```
extern spe_program_handle_t spu_main
```

This step requires that the SPU binary is already converted into a PPU object and has a program handle defined as *spu_main*. The conversion process is done by the following command:

```
ppu-embedspu [handle] [SPU binary] [output object file]
```

This command creates the object file based on the given SPU binary. This object file contains the actual definition of `spu_main`, and later it will be linked into the PPU binary as a separate segment. This procedure ends up with a single executable file.

4.5 Data Transfer Optimization

Parallel call mode allows us to distribute the program and input data to different SPUs. Input data is transferred through MFC to the local storage. The logic requires that the stub function on the PPU should create several threads; each thread keeps calling the target SPU function with different datasets in a loop until all data are processed. Obviously, the overhead of this approach is excessive:

1. Every time a thread calls an SPU function, it needs to prepare the input buffer, including initializing the header and copying the parameter data.
2. Before the SPU function starts, the input buffer needs to be transferred to its local storage with an MFC request by the PPU. This is an I/O operation, and the processor stays idle until the transfer is finished.
3. Cell B.E. is used primarily for computation-intensive tasks. Programs solving multimedia, scientific computations usually involve a large quantity of input data, such as arrays or a media stream. In these cases, each thread needs to call the SPU function repeatedly until all input data is processed. Each function call is identical to a simple call, which involves a processor switch and arguments passing across processors.

This section discusses how to optimize the parallel call mode in order to reduce the overhead of the heterogeneous function call. Most of the existing work tries to optimize the function call that is similar to the simple call mode. For example, [17] uses local storage as a cache. It analyzes the instructions and checks if the data exist in cache or should be transferred from the main memory. However, this optimization does not utilize the fact that each SPU function will be called more than once. [34] improves this by introducing double buffering logic so that the next dataset is transferred to the local storage while the SPU is processing the existing data. A similar idea is employed whereby each SPU has two buffers, one for computation and the other for receiving the next dataset. The parallel call logic is also modified to avoid unnecessary processor switches. With these modifications, the SPU can stay busy most of the time until all datasets are processed.

In the parallel call mode, input data is divided into several datasets and each thread picks up a group of them. Every time an SPU function is called from the thread, only one dataset is transferred to the SPU to process, which is the reason why several function calls are needed in a loop. In addition, the stub function first must copy the dataset values to an input buffer, because the input data exists in different buffers in the main memory. For example, a matrix multiplication function requires two matrices, and their memory buffers most of the time are not contiguous. Transferring these buffers directly requires more MFC operations and costs a longer amount of time. Later, this input buffer needs to be copied to local storage again. Instead of preparing one dataset for one SPU call, the whole input data could be simply passed to the SPU. The split logic is moved to the SPU stub function to decide which part of the input data represents a single dataset to be processed. This step allows the loop to exist also in an SPU stub function, which means that only one processor switch is required for each thread.

The input buffer formats must be modified to support this new data transfer logic. Three fields shown in Figure 4.12 are used to help us eliminate the overhead:

1. `arr_ea` specifies the address of the input data array in main memory. It points to a location of the original array instead of a new copy of the datasets for the current thread.

2. `arr_size` represents the total size of the array.

3. `spuid` tells the program that which SPU it is running on. This value is the index value assigned to this thread and is not required to be the same as the real SPU ID.

These values are employed together with parameters in `spu_for` to divide the input data on the SPUs.

As mentioned previously, the MFC requires the transfer size to be aligned with 16 bytes, or the data has sizes of 1, 2, 4, or 8 bytes. To avoid extra operations for the transfers of small data, *group* is used to represent the data size transferred within each MFC operation. If each dataset is smaller than 16 bytes, *group* will be 16-byte aligned and will contain 16 datasets; the total size will be $step\_size \times 16$. Otherwise, each *group* will only contain one dataset.

This address of the array will be used as *arr_ea* and passed to the SPU program. On the SPU, the program entry stub first checks this value. If it is not initialized and appears as value 0, the execution falls back to the old execution for the simple call mode. Otherwise, the program enters a loop to process all datasets for this thread. It applies the following equation to calculate the address of the i^{th} group in main memory:

$$addr_i = arr\_ea + (spu\_id + i * spu\_count) * group\_size$$

Instead of `step_size`, `group_size` is used in this equation, so the case can be handled correctly wherein small datasets are aggregated. The execution runs until the address of a current group goes beyond the array size. Figure 4.20 shows the logic to process all the datasets in one SPU context.

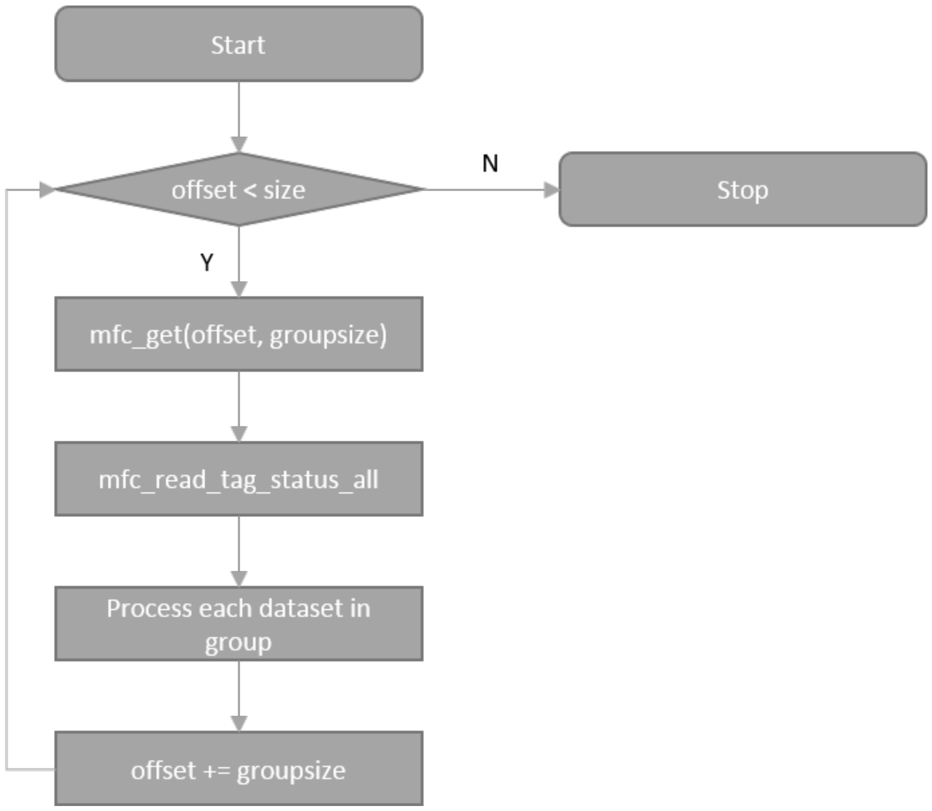


Figure 4.20: Parallel Call with Single Processor Switch

The above logic eliminates the extra SPU calls within one thread. It will transfer the input data and will be blocked until the SPU processes all the data. However, the SPU still retrieves only a small group from the main memory each time, and during the MFC transfer, the SPU is blocked there doing nothing.

Thus, another optimization for the parallel mode is to allow the SPU to trigger the transfer for the next available group into local storage while it is working on the current one. This optimization is based on the fact that both operations (computation for a current group and the MFC transfer for the next group) will take some time, and if they can be executed in parallel, time for the data transfer will be saved. It also actively checks the MFC transfer state before the computation to make sure that it is completed.

While the SPU is processing the current group, the buffer that has the input data and all other parameters used by the target SPU function must remain exclusively dedicated to this computation during the whole execution. So, another buffer in local storage is required to store the data for the next group. In the entry stub, two buffers are initialized, and only one is marked as active to hold the current group; the other remains available for transferring the next group. Every time a group is processed, the active and inactive buffers are switched for the next iteration. This is a pointer exchange operation that involves no memory copy.

As mentioned earlier, the SPU stub functions use the `mfc_get` function to trigger an MFC data transfer. This function itself starts the transfer asynchronously, meaning that it will not wait for completion, so the computation work can start right away. Another function, `mfc_read_tag_status_all`, queries the status of the MFC tag used by the `mfc_get` to see if the operation is completed. After processing the current data, this function will therefore be called to make sure the next data block is fully transferred. Figure 4.21 shows how the data transfer and the execution of the target function are interleaved.

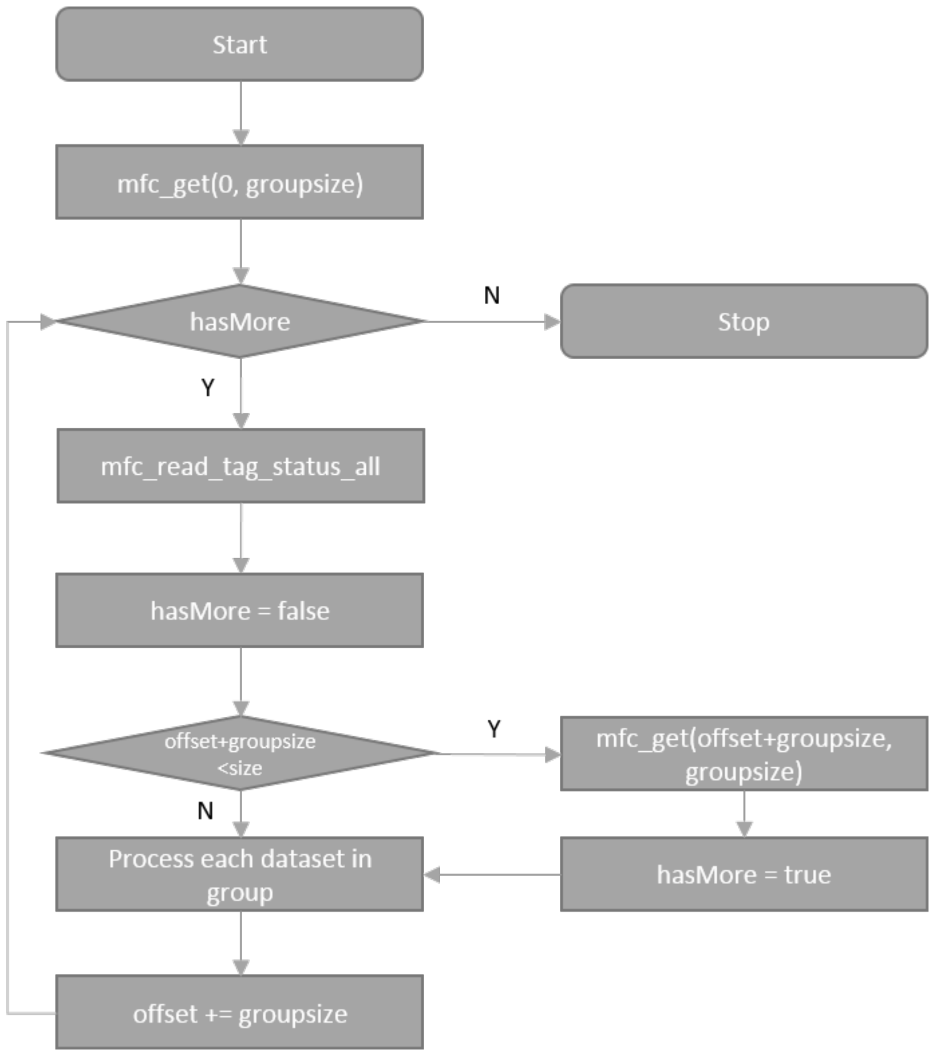


Figure 4.21: Parallel Call with Overlapped I/O

4.6 Summary

This chapter described how TigC completes each step in the compilation of the source file. In addition, it discussed how a function could target the SPU architecture to utilize the power of the Cell B.E. platform. By defining the format of the input buffer and providing corresponding marshaling logic, complicated data structures (such as arrays) can be passed to the SPU and will be correctly recognized. It also evaluated different approaches for distributing the computation to different SPUs. Various improvements were proposed and implemented to reduce the overhead of using the SPU.

5. Performance Evaluation

This chapter compares the results of executing different applications to evaluate the performance improvements from using TigC compiler against the Cell B.E. platform. First, this chapter examines the workloads employed as the input source code for TigC and goes onto compare the performance differences between running the programs on the PPU only and under the SPU call mode. Next, the results achieved by employing different number of the SPUs and increasing the input data size are compared to see how the time cost changes. Finally, this chapter describes the potential optimization opportunities that this compiler provides by processing source files for different architectures.

All the workloads are running on the IBM BladeCenter QS20 system, which contains nine core processors with one PowerPC core and eight SPE cores. Each core has a peak performance of 3.2GHz. The system is configured with 8GB of main memory, and each SPE has 256KB of local storage.

PPC64 Linux with kernel 2.6.18 is running on the PPE as the host OS to manage the system. It provides the Cell B.E. SDK which includes GCC 4.1.2 for the PPU architecture and GCC 4.1.1 for the SPU architecture.

5.1 Workload

The Cell B.E. is designed primarily for computation-intensive jobs, such as gaming, video processing and scientific computation. Matrix operations are common in these areas. For example, a gaming program may use matrix multiplication to transform a vector in 3-D space, which, in the end, represents a character's arm movement. In the experiments, two computation-intensive programs are tested for the performance evaluation:

1. *Strassen's Matrix Multiplication* Here, matrix multiplication refers to the matrix product rather than that of scalar multiplication. It is a binary operation that takes two matrices where the number of columns of the first matrix should be equal to the number of rows of the second matrix. The result is another matrix. For example, given the following two matrices:

$$A = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix} B = \begin{pmatrix} B_{11} & B_{12} & \cdots & B_{1p} \\ B_{21} & B_{22} & \cdots & B_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ B_{m1} & B_{m2} & \cdots & B_{mp} \end{pmatrix}$$

   the multiplication result is defined as

$$AB = \begin{pmatrix} AB_{11} & AB_{12} & \cdots & AB_{1p} \\ AB_{21} & AB_{22} & \cdots & AB_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ AB_{n1} & AB_{n2} & \cdots & AB_{np} \end{pmatrix}$$

   where each entry is defined by $AB_{ij} = \sum_{ik}^{kj} A_{ik}B_{kj}$. With the naive algorithm shown above, it is obvious that the time cost of multiplication for the matrix of size $n \times n$ is $O(n^3)$, which is considered inefficient for large matrices. Many more efficient algorithms exist to reduce the multiplication operations. One example is the Strassen algorithm. To simplify the problem, it assumes the input matrices have the size $2^n \times 2^n$. For other sizes, they can be transformed by padding rows and columns of 0. To calculate $C = AB$, it first splits the input matrices into the following format:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

where A_{ij} and B_{ij} are matrices of size 2^{n-1}. With the following auxiliary matrices:

$$\begin{aligned} M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ M_2 &= (A_{21} + A_{22})B_{11} \\ M_3 &= A_{11}(B_{12} + B_{22}) \\ M_4 &= A_{22}(B_{21} - B_{11}) \\ M_5 &= (A_{11} + A_{12})B_{22} \\ M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ M_7 &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned}$$

Then the result could be calculated by using:

$$C = \begin{pmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{pmatrix}$$

The computation of the auxiliary matrices also requires matrix multiplications which may be done either recursively or with the naive method. With the recursive implementation, if $f(n)$ is the number of operations involved in $2^n \times 2^n$ matrix calculation, it is clear that $f(n) = 7f(n-1) + I$ where I represents the additions to calculate C. Therefore, the asymptotic complexity for multiplying matrices of size $N = 2^n$ using the Strassen algorithm is

$$O([7 + o(1)]^n) = O(N^{log_2^{7+o(1)}}) \approx O(N^{2.807})$$

While the Strassen algorithm saves the multiplication instructions, it also introduces some extra additions and requires more memory allocations. For small matrices, the naive method offers a better solution because this overhead will be more expensive than the instructions saved. This boundary is decided by running both naive and Strassen algorithms with different matrix sizes to see at which point Strassen starts to outperform the naive method. To make a clear difference, the multiplication is run for 32K times with both methods, and the result is shown in Figure 5.1.

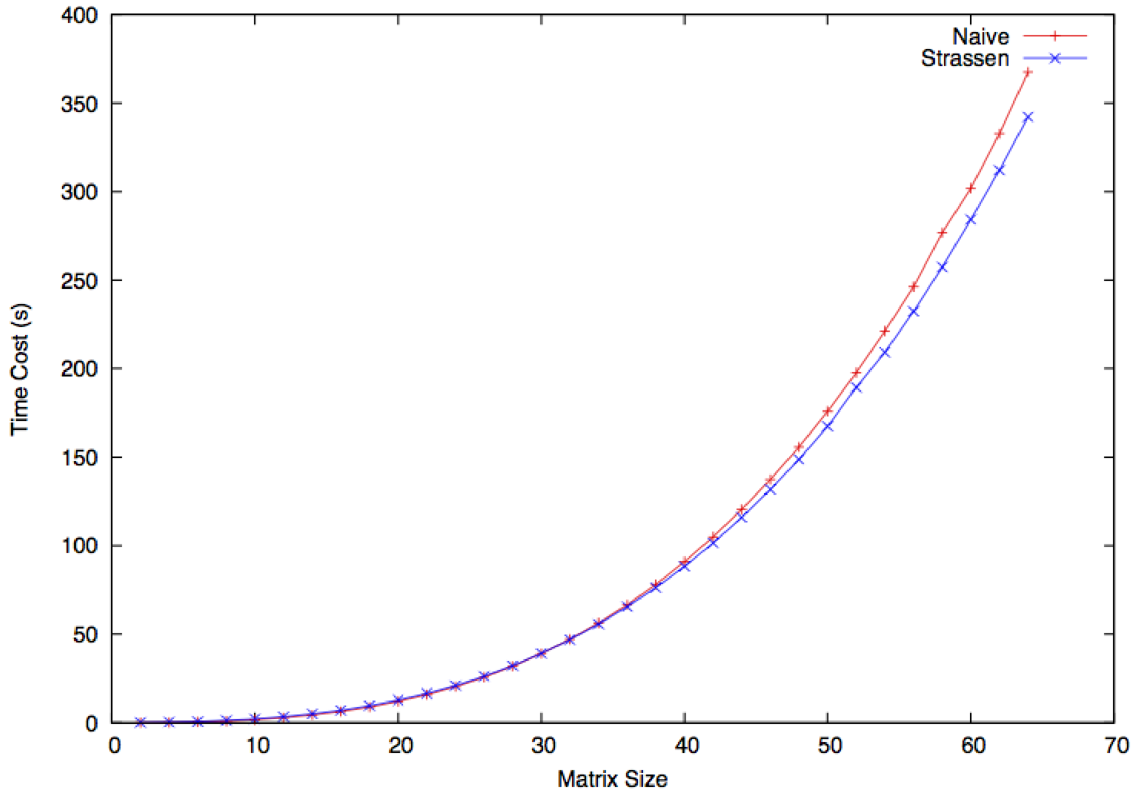


Figure 5.1: Naive vs Strassen Matrix Multiplication

It is obvious that the Strassen algorithm outperforms the naive method only when the input matrices are at least 32×32. The test program is subsequently designed in a way that large matrices can be split on the PPE recursively by using the Strassen algorithm until it reaches size 16×16. Thereafter, calculations for the auxiliary matrices can be distributed to different SPEs and solved by naive method.

In the experiments, the Strassen algorithm is run on 200 pairs of matrices with the same size $n \times n$, where n is from 64 to 512. To make the distributions easier, when the matrix size is 64×64, there will be 2 times of expansion to get 49 pairs of 16×16 matrices. These small matrices were then distributed to different SPUs.

2. *n-Queens Problem* This is a classic problem that asks how to put n queens on an ordinary $n \times n$ chess board so that none of them could hit each other within one move. In chess, a queen can move as many steps as she wants in all directions, vertically, horizontally and diagonally. This means that two queens should not appear in a line to meet the requirements. The 8-queens problem has a total of 92 solutions. Figure 5.2 shows one example of a valid solution to the 8-queens problem.

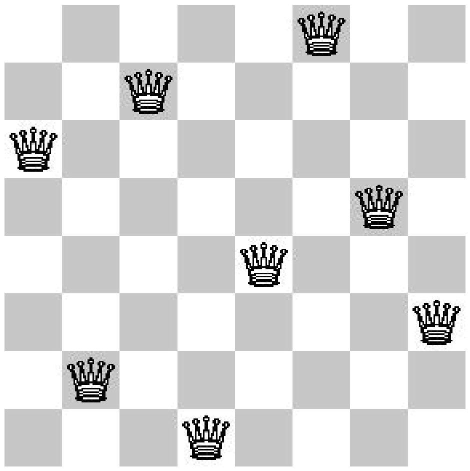


Figure 5.2: One Solution for 8-Queens Problem

This test evaluates the performance by measuring the time cost of finding all valid solutions for a given n-Queens problem. The naive solution is to enumerate all possible placements of the queens and to remove those that represent invalid solutions, a procedure which is quite computationally expensive. The implementation used in the experiment is based on the Wirth's algorithm[52]. It records the current states of the chessboard by using three vectors to represent the vertical and diagonal placements of queens, and placing a queen on the chessboard means that the appropriate elements need to be set in all three vectors. To check whether a position is valid, the program needs to check if any corresponding element in those vectors is set. Because the algorithm is working on a single row at a time, no extra vector for the row collision is required. There are constraints introduced to avoid known symmetries, but they are not relevant to the research.

To fully utilize the power of the Cell B.E., the n-Queens problem should be divided into subsets, thereby allowing more than one SPU to run together. The parallel approach is the same as the process described in [18]. Each SPE gets a set of starting positions in the first row and is asked to find all the valid solutions given the placement of the first queen. Note that the n-Queens problem does not involve too many MFC operations, differing markedly from the Strassen's matrix multiplication above.

The experiments try to solve the n-Queens problem where n is a number from 4 to 16. Note that there is no solution for $n < 4$ except $n = 1$.

Both of the workloads require a huge number of computations when the input size increases. However, only Strassen's matrix multiplication requires abundant data transfer across processors, so the behaviors of the Cell B.E. can be inspected in two different cases. In each run of the experiments, `time` command is used to measure the performance. The command reports three values:

- *real* refers to the elapsed wall clock time.
- *user* refers to the CPU time that the process spends under user mode.
- *sys* refers to the CPU time that the process spends under kernel mode.

In the experiments, the wall clock time (*real*) is chosen for performance evaluation. Both algorithms ran 8 times, and the average value was calculated. Also, the system has no other computation application running.

5.2 Simple Call Mode

As described earlier, the simple call refers to the case where only one SPU is used. Whenever an SPU function is called, the PPU stub function will prepare a data buffer having all the parameter data and will then transfer it to the SPU. The SPU entry function interprets this buffer to get the arguments and to run the target function. When it finishes the execution, the return value will be transferred back to the main memory. During this whole period, the PPU is blocked and waiting for the execution.

Obviously, simple call does not bring any parallel performance improvements because the work is not distributed to different processors, and extra steps are required to prepare the context for the SPU to start, which, in theory, makes the program even slower than just running on the PPU. But, this procedure provides us the performance baseline for the workloads. Figure 5.3 shows a comparison of the calculation of matrix

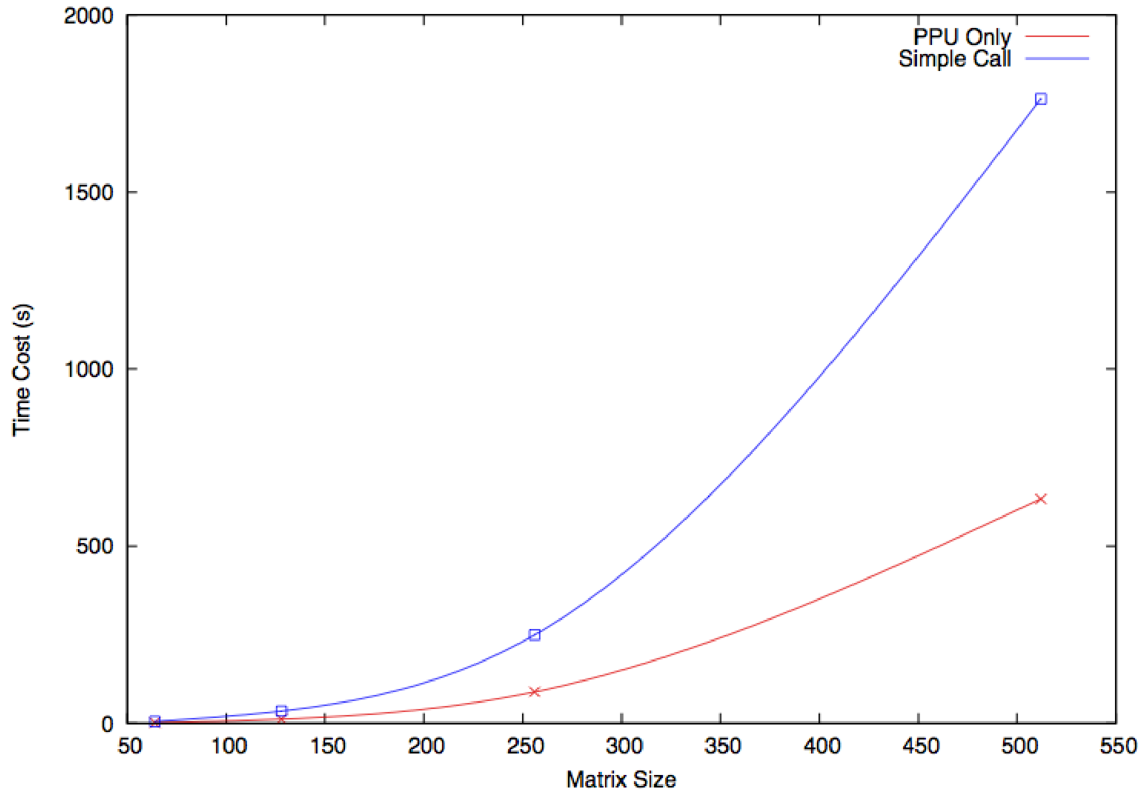


Figure 5.3: Strassen Algorithm (PPU Only vs Simple Call Mode)

multiplications with different input matrices sizes using the Strassen algorithm. With the extra operations involved in the simple call, including memory buffer initialization, MFC transfers and SPE context start/stop, the program requires a longer time than it would with PPU only. As the matrix size increases, the simple call involves additional data transfer between the main memory and the local storage, so the gap gets larger.

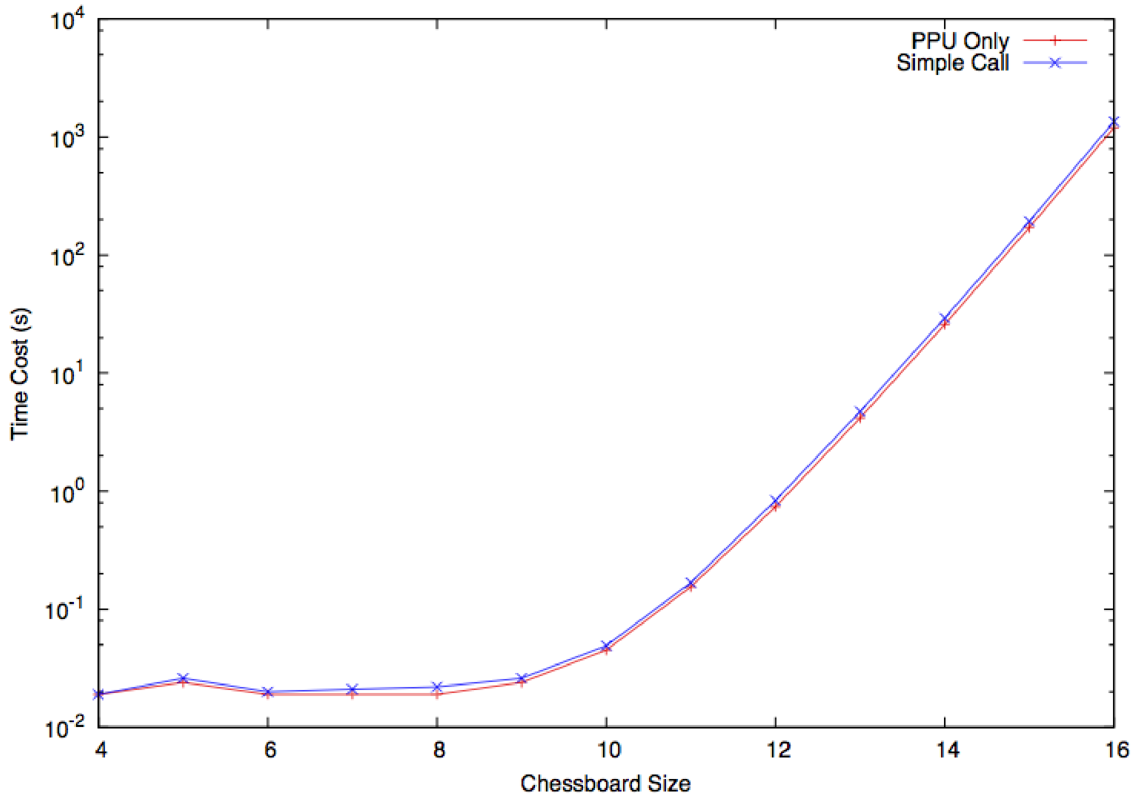


Figure 5.4: n-Queens Problem (PPU Only vs Simple Call Mode)

Figure 5.4 shows the comparison between solving the n-Queens problem on PPU only and with one SPU under simple call mode. The simple call still takes longer than it would on PPU only. However, the extra time used by the simple call is not as high as the one in the Strassen's matrix multiplication workload. This occurs because the n-Queens algorithm just transfers the chessboard size to the SPU to start the computation. Compared with transferring the matrix data, this is a relatively cheap operation.

5.3 Parallel Call Mode

With parallel call mode, ideally, the input data is distributed to different SPUs, and each SPU works solely on its fraction. So, the time cost should decrease when the number of cores increases. However, there are certain parts of the code which cannot be executed in parallel. For example, the parameter's buffer should be prepared to start the SPE context. In addition, the workload itself may have some parts that need to be done in order and cannot be split. According to Amdahl's Law, the potential performance improvement is based on the fraction that could not run in parallel.

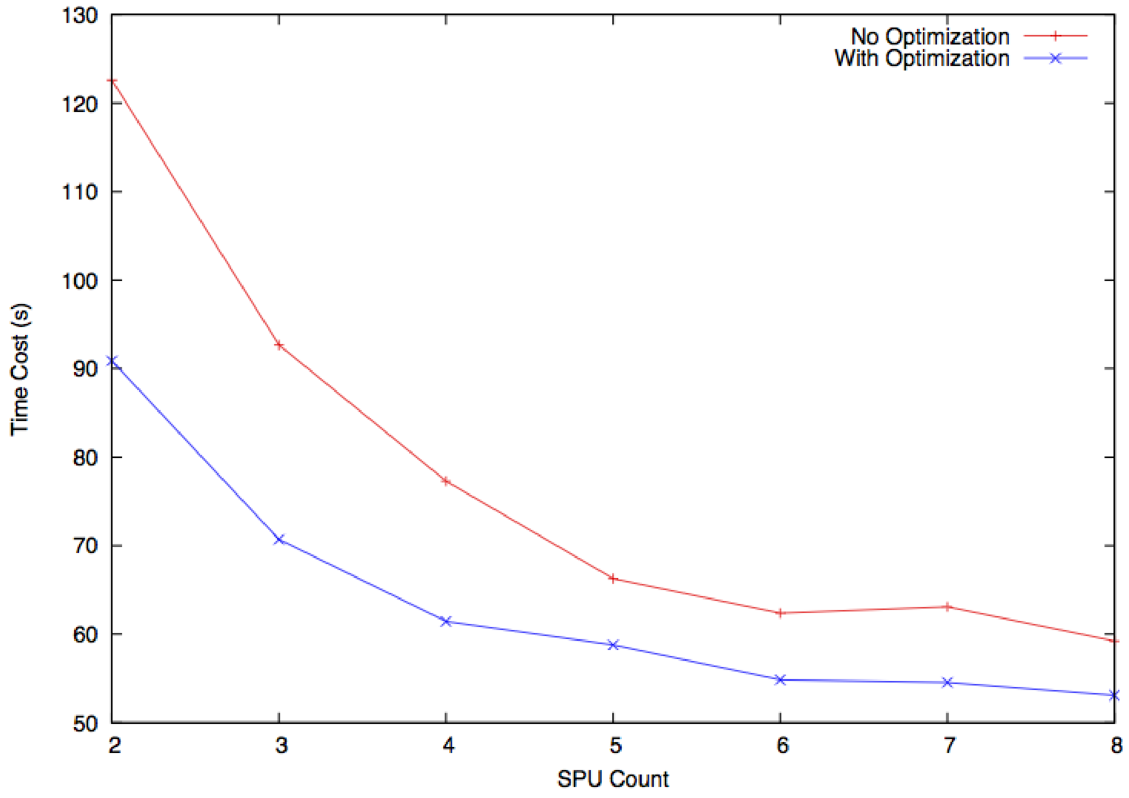


Figure 5.5: Strassen Algorithm with Matrix Size 256×256 (Data Transfer Optimization on vs off)

As mentioned in the previous chapter, when calling SPU functions under the parallel call mode, an asynchronous MFC transfer can be issued for the next batch of input data before processing the current batch. This optimization makes the data transfer in parallel with the function execution, and the SPU running time will be reduced to $max(T_{data-transfer}, T_{function-execution})$. Figure 5.5 shows the result of enabling the data transfer optimization when running the Strassen algorithm with different matrix sizes. It does provide some performance improvements because this workload involves many MFC operations to transfer a large amount of matrix data to the local storage. It also shows that as the SPU count increases, the difference between the optimization on and off actually decreases. This situation happens because when the same amount of work is distributed to more cores, each SPU ends up with less input data. As a result, less time is spent on the data transfer between processors because more MFC operations can run in parallel. This also leads to less performance improvements from the optimization.

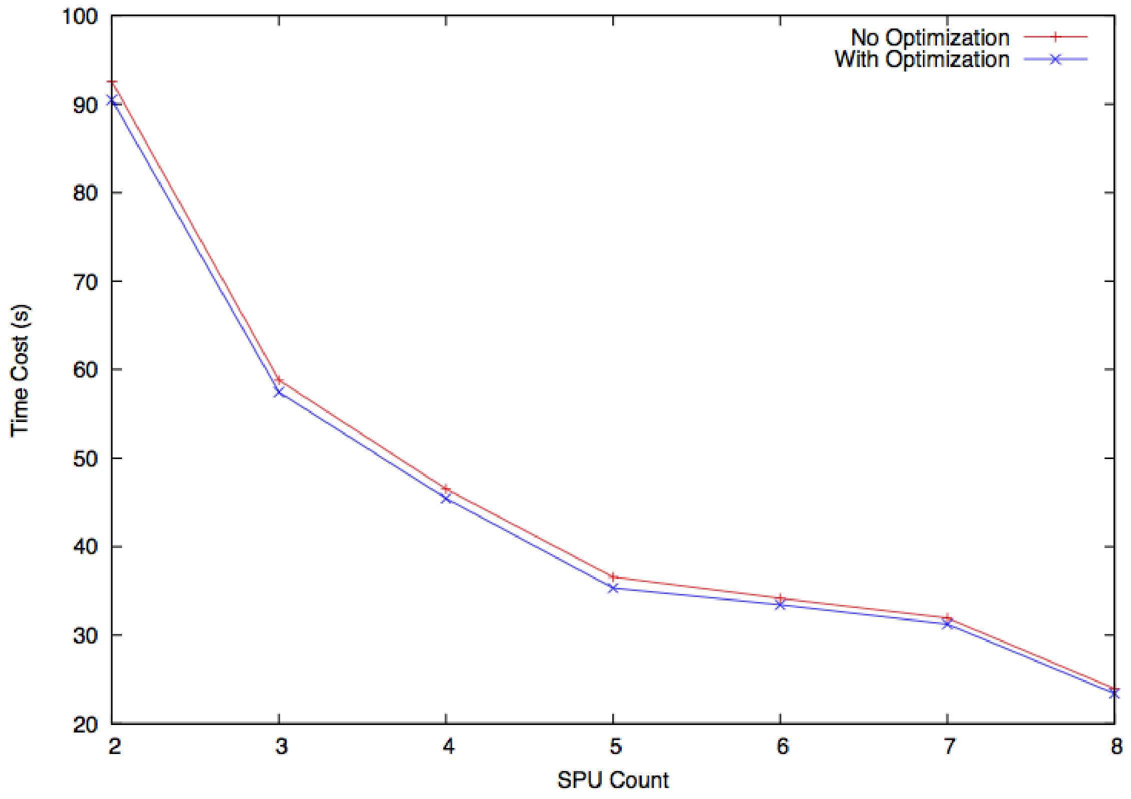


Figure 5.6: 15-Queens Problem (Data Transfer Optimization on vs off)

Figure 5.6 shows the comparison demonstrated by solving the 15-Queens problems with data transfer optimization enabled. It shows that although there are still improvements, the difference is actually quite small, and it keeps almost an unchanged value while the SPU count is increased. This outcome is expected because, for the n-Queens problem, the algorithm requires just one MFC transfer at the beginning of each parallel execution to decide its place position of the first row on the chessboard; there is nothing else to optimize for data transfer in this program.

All of the following experiments are run with data transfer optimization enabled. The performance of the parallel calls are evaluated with different SPU counts. Figure 5.7 shows the time cost of running the Strassen's matrix multiplication under the parallel call mode. It shows that with only two SPUs, the time cost is decreased significantly, nearly the same as those seen when using the PPU. However, compared to the case where only the PPU is used, it does not save half of the time cost. One reason is that there are extra operations needed to run functions on the SPUs. As described before, under the simple call mode, it also requires the same amount of this extra work. So, if comparing the results of two SPUs with the simple call, it saves much more of the running time, which is exactly expected.

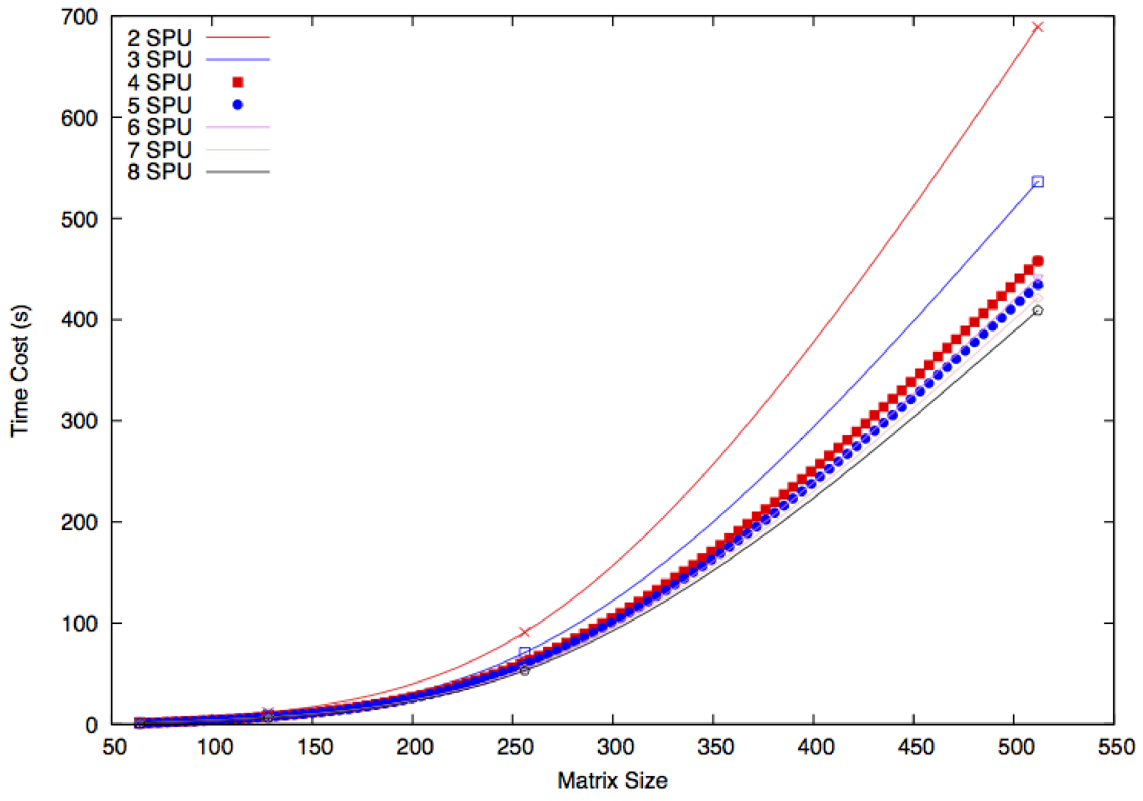


Figure 5.7: Strassen Algorithm under Parallel Call Mode

Increasing the SPU count, however, does not improve the performance at the same rate. For example, the multiplication of two 512×512 matrices for 200 times requires about 700 seconds with two SPUs, but it requires about 450 seconds with four SPUs instead of 350 seconds. This result can be explained by the following two reasons:

1. As described, the multiplications are distributed when the input matrix has a size of 64×64. So in total there will be 49 multiplications of 16×16 matrices to calculate the auxiliary matrices. The round-robin method is employed to distribute all of these multiplications to different SPUs without priority; in the end, it is likely that some SPUs will remain idle because there is no more work to do. For example, 49 multiplications require four SPUs running 13 batches, where, in the last batch, only one SPU is used. It also requires eight SPUs running seven batches.

2. Based on Amdahl's Law, the potential improvements with more cores depend on the fraction of the program that cannot run in parallel. However, with the Strassen algorithm, only the calculation of auxiliary matrices is distributed to different SPUs. The program still includes many sequential steps before and after this calculation, such as allocating buffers for those temporary matrices and adding the returning auxiliary matrices up to the final result. Thus, the time cost will not decrease at the same rate.

Figure 5.8 shows the results of solving the n-Queens problem using different numbers of the SPUs. There are several differences to be compared with the results of the Strassen algorithm:

1. Even with only two SPUs running under the parallel call mode, the procedure reduces half of the running time compared with running on PPU only. For example, it takes about 1200 seconds to solve the 16-Queens problem on the PPU, but with two SPUs, it takes only 600 seconds.

2. As the SPU count increases, the time cost decreases almost at the same rate. For example, two SPUs require around 600 seconds to solve the 16-Queens problem,

while four queens requires around 300 seconds.

This outcome is also expected because the n-Queens problem involves only one MFC operation at the very beginning for each run. In addition, with the distribution mechanism, most parts of the program can run in parallel. The only sequential operation is that of summing up the results from each call, which involves only a few instructions and is fast. Note that there are some exceptions:

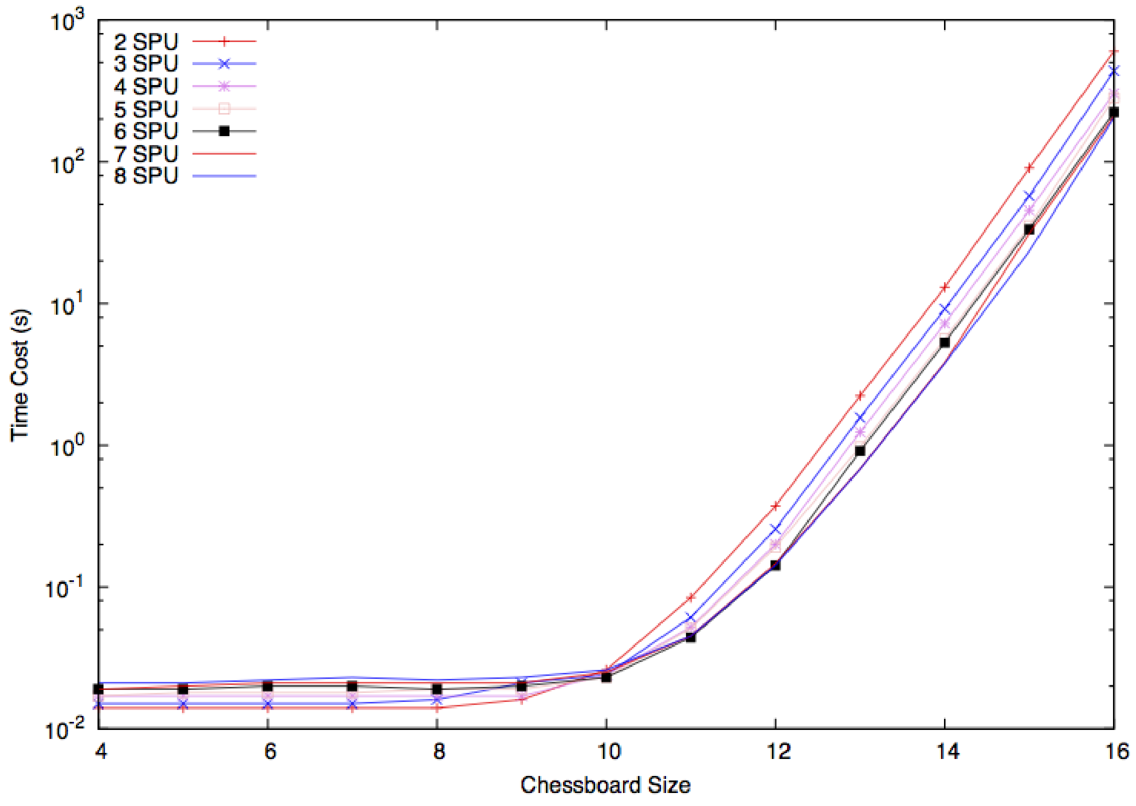


Figure 5.8: n-Queens Problem under Parallel Call Mode

1. When the chessboard size is small (in this case, if it is smaller than 10), solving the problem takes less time than SPE contexts initialization. Increasing the SPU count will eventually cost a slightly longer time because the program needs to initialize more contexts at the beginning.

2. The distribution mechanism on this workload may also waste some SPU cycles at the last iteration. For example, if eight SPUs are used on a 12-Queens

problem, only four SPUs will be employed during the last iteration; it does not bring twice the performance improvements compared to the four SPUs case.

Also different SPUs have a similar time cost when the chessboard has size 10. This can be explained as follows: the total execution involves creating SPE contexts, solving 9-Queens problems with the available SPUs, and destroying all these contexts. Except the second step which is distributed to different SPUs, the other two steps are executed sequentially on the PPU. Assume that initializing and destroying one SPE context costs $T_{context}$, and a 9-Queens problem costs T_{queen}, the total execution time costs with different SPU counts is shown in Table 5.1. It is obvious that if $T_{context}$ is close to T_{queen}, the results will converge when the chessboard is 10.

SPU Count	Time Cost
2	$2 \times T_{context} + 5 \times T_{queen}$
3	$3 \times T_{context} + 4 \times T_{queen}$
4	$4 \times T_{context} + 3 \times T_{queen}$
5	$5 \times T_{context} + 2 \times T_{queen}$
6	$6 \times T_{context} + 2 \times T_{queen}$
7	$7 \times T_{context} + 2 \times T_{queen}$
8	$8 \times T_{context} + 2 \times T_{queen}$

Table 5.1: Total Execution Time of 10-Queens problem

5.4 Potential Optimizations

As described in chapter 3, existing compilers, such as *ppu-gcc* and *spu-gcc*, work independently. *ppu-gcc* parses source files that contain only the PPU functions and generate the PPU binary, while *spu-gcc* processes only SPU source files. There is no information shared between these two compilers, making it difficult to perform future optimizations. For example, from *ppu-gcc*'s view, calling an SPU function always ends up with calling the PPU function `spe_context_run`. It will be difficult for the compiler to learn the fact that this call involves a processor switch. Even if the compiler has built-in knowledge to treat this case specially, it still needs to differentiate the preparation code, such as input buffer initialization, from the actual work.

TigC is different in that it accepts source files that contain both the PPU and the SPU functions. It will process them together and generate the result binary. This capability allows the compiler to have a global view of the program. During the compilation, the SPU function calls are represented with a single IR node `R.SPUCALL`. All the preparation code is separated from users' codes and hidden in the stub function, which makes the optimization component simple. In addition, using a different IR node instead of the original `R.CALL` provides the information that this function call actually requires a processor switch, and provides evidence that the compiler can further optimize the program.

This section discusses two optimization opportunities that can be implemented based on the knowledge provided by TigC.

5.4.1 Code Reordering

One optimization that usually happens within the processors is the out-of-order execution (OOE)[49]. The basic idea involves processing the next available instruction using some instruction cycles that would otherwise be wasted by costly delays. Previously, instructions were executed in-order, meaning that the processor fetches an instruction, and if the input operand is not available because it is being fetched from memory, the processor blocks waiting until this step is finished. With OOE, an instruction can happen before an earlier instruction if its data is ready.

The code reordering optimization contains a similar idea but at a higher level. Calling an SPU function has a costly delay because it may be a computation-intensive algorithm dealing with a large amount of input data. The goal is to run statements that are ready for execution before the call returns. To simplify the problem, the optimization focuses on a small scope where no loop is involved.

First, variables used within a statement are considered as its data. A variable is not ready if its value relies on the output of the function call, which means that such a statement will be unable to run until the call is completed. However, for statements that do not use any data modified by the function call, they can be reordered so the processor can overlap their execution with the function call. Differing from OOE, this reordering logic occurs during the compilation after the source files are translated into the IR format.

To achieve this goal, a separate optimization component needs to be added in TigC. It will be triggered directly after IR is generated. In addition, IR nodes `R.SPUCALL` and `R.PARCALL` should first be modified to be asynchronous, meaning that calling an SPU function will be non-blocking. This optimization component takes the IR nodes as input, and when a heterogeneous call is detected, a special IR node `R.SPUWAIT` must be added to explicitly wait for the completion of the SPU function call. This is a blocking operation, working as a barrier to make sure that all variables have correct values before they are used. Its initial position follows right after the function call statement. TigC needs to analyze the following statements to check whether they are dependent on the result of the function call. For one with no dependency, it will be reordered to run before `R.SPUWAIT`. After the reordering, the resulting program should be similar to the following format:

```
..
R.SPUCALL / R.PARCALL
..
Statements that do not depend on the SPU function's output
..
R.SPUWAIT
..
Statements that depend on the SPU function's output
..
```

The procedure to analyze the IR to determine if a variable is in use is called liveness analysis. A variable is live if its value may be needed later. The optimization component first converts the input IR nodes into a control-flow graph wherein each node represents a statement in the program. Any assignment to a variable *defines* this variable, and any occurrence of a variable on the right side of an expression *uses* this variable. For a statement node, `defs` and `uses` store the variables that are defined and used in this statement. To make sure the reordering does not change the program's behavior, a directed graph is built; each node represents a statement in the control-flow graph, and each edge represents the dependency between two nodes. A statement S_1 is dependent on another statement S_2 if it meets one of the following conditions:

1. There is a path in the control-flow graph from S_2 to S_1, and S_1 uses a variable that is defined only by S_2 on this path. This stipulation guarantees that a statement will not be executed before all variables it uses are initialized.

2. There is a path in the control-flow graph from S_2 to S_1, and S_1 defines a variable that is used or defined in S_2. This stipulation guarantees that a statement modifying a variable will not impact statements that should use the old value of the variable.

```
                b := spu_function(a);
                c := b + d;
                d := 7;
                e := a + 4;
                f := e + d;
                g := e + 5;
                e := 8
```

Figure 5.9: Sample Program for Code Reordering

For example, considering the program shown in Figure 5.9. Pseudo code is used here to simplify the description. `spu_function` is an SPU function and involves a heterogeneous call. Its dependency graph is shown in Figure 5.10. The first dependency rule is shown as solid edges, while the second rule is shown as dash edges. A path in this graph represents an execution order that must be maintained.

With the dependency graph, TigC can reorder the statements using Algorithm 1. In this algorithm, SPUCALL is employed to represent both SPUCALL and PARCALL. It keeps a set to record all nodes that have no incoming edges, meaning that

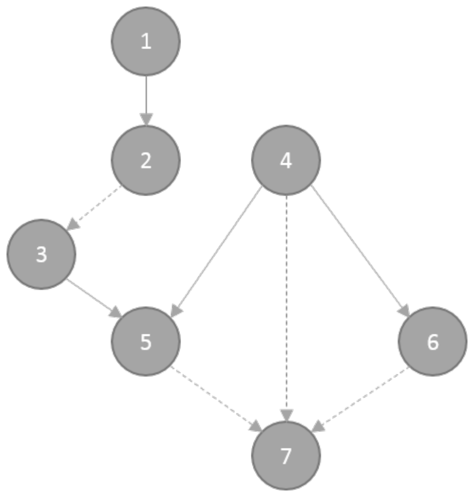


Figure 5.10: Dependency Graph of the Sample Program

their dependent nodes are already completed and ready to run. Once a node is considered as completed and removed from this set, the dependency graph needs to be updated by removing all edges from this node. This procedure is repeated until the set is empty. All the remaining nodes in the graph have dependency on the SPU call node and cannot be reordered. The resulting IR nodes after the optimization will be passed to the backend for code generation.

Algorithm 1 Algorithm to Reorder the Statements

```
Require: $S$ includes all statements after the SPU call
Require: $D$ is the DependencyGraph
    $L \leftarrow$ Empty List
    $R \leftarrow$ Set of nodes with no incoming edge
    remove $node_{spucall}$ from $R$
    while $R$ is non-empty do
        $n \leftarrow R$.popfront()
        $L$.append($n$)
        for all node $m$ with an edge $e$ from $n$ to $m$ do
            $D$.remove($e$)
            if $m$ has no incoming edge then $R$.append($m$)
            end if
        end for
    end while
    emit SPUCALL
    emit statements in $L$
    emit SPUWAIT
    emit statements in $S - L$
```

There is one important rule for code reordering: it cannot change the order of the program output. Statements such as `print` must maintain their original order. This point can be solved by stopping processing the code on the first output statement. In addition, a threshold may be set to limit the statements the compiler can reorder before the `SPUWAIT`.

5.4.2 Optimized SPU Dispatching

The experiment results show that increasing the SPU count does not bring the same rate of performance improvement. For example, from Figure 5.7, it is clear that using four SPUs does not reduce half of the execution time of two SPUs. This result is explained in Section 5.3 with Amdahl's Law and the load balancing problem. This means that given a parallel call, although using all available SPUs leads to the locally optimal execution time, it is not guaranteed that the result will be globally optimal for the whole program. In this section, another opportunity is described to further optimize the program by choosing proper amount of the SPUs for different SPU calls.

Considering the following sample program which has two parallel calls on `spufunc_a` and `spufunc_b`:

```
a = spufor spufunc_a of (a_input , 128, 1024)
b = spufor spufunc_b of (b_input , 128, 1024)
```

Assume that these two functions have no data dependency, it means neither function relies on the output of the other one. If both parallel calls take $900ms$ when running with four SPUs, and $600ms$ with eight SPUs, the current implementation of TigC will execute them sequentially, both with all eight SPUs; ultimately, the total time cost will be $600ms + 600ms = 1200ms$. However, because these two parallel calls have no data dependency, they can run in parallel with different sets of the SPUs. If eight SPUs are divided into two groups, one for `spufunc_a` and the other for `spufunc_b`, both parallel calls have four SPUs; the total time cost will be $900ms$. For each function, the execution time is $300ms$ longer than before, but the total time cost of the program is lower.

This optimization also works with IR and can be implemented in the same optimization component. It relies on the result of the previous liveness analysis to detect whether two parallel calls have any data dependency and can be reordered to execute in parallel. If so, TigC needs to determine the optimal SPU dispatch schemes. First, the compiler must estimate the time cost of the SPU function. One simple solution is to analyze all instructions involved in the call, including the stub function. This can be easily achieved because TigC processes all the source files. Because the function may contain loops, it cannot be calculated by simply summing up the CPU cycles of each function. In [44], the authors proposed a system to estimate the execution time during runtime based on static dependent cost expressions generated during compilation. These expressions can be included in a component to be invoked by the optimization component. This method requires a dynamic dispatching logic because when compiling the source code, TigC cannot decide the optimal SPU assignment. Another solution is to allow programmers to use some annotation to specify the time cost for a parallel call:

```
spufor spufunc_a (8:600, 4:900, 2:1000) of (a_input, 128, 1024)
```

This means the parallel call on `spufunc_a` will cost $600ms$ with eight SPUs, $900ms$ with four SPUs and $1000ms$ with two SPUs. These annotations will be stored in the IR node and used by the optimization component, and the dispatching can happen during the compilation procedure.

Considering the static dispatching as an example, with the time cost estimation for each PARCALL, Algorithm 2 is activated to decide if some parallel calls should be merged to emit the final code. PARCALL is modified to accept the SPU count assigned for this parallel call. Given a valid assignment, the algorithm first considers PARCALLs that have an SPU count greater than zero. These PARCALLs will be put in a single batch and run in parallel, so their time cost depends on the slowest one. For all PARCALLs with zero SPU assigned, they are processed in different batches, so a recursive way is used to find their best assignment. In the end, the parallel call is emitted for each batched assignment with an SPUWAIT. A dynamic dispatching mechanism may be implemented through applying a similar idea. Instead of emitting an IR node, it can make parallel call directly.

Algorithm 2 Algorithm to Dispatch SPU for multiple PARCALLs

```
Require: $S$ represents PARCALL statements with no dependency
Require: $E$ is the time cost estimation
    function DispatchWithMinCost($S$, $E$, $R$)
        $T \leftarrow$ MAX
        for all $A$ as a valid SPU assignment for $S$ do
            $L \leftarrow 0$
            for all PARCALL $c \in S$ where $A[c] \neq 0$ do
                if $E[c][A[c]] > L$ then
                    $L \leftarrow E[c][A[c]]$
                end if
            end for
            $M \leftarrow$ every statement $s$ where $A[s] = 0$
            $L+ =$ DispatchWithMinCost($M$, $E$, $A$)
            if $L < T$ then
                $R$.insert($A$), $T \leftarrow L$
            end if
        end for
        return $T$
    end function
    DispatchWithMinCost($S$, $E$, $R$)
    for all Batched Assignment $A$ in $R$ do
        for all PARCALL $c$ in $A$ do
            emit PARCALL($c$, $A[c]$)
        end for
        emit SPUWAIT
    end for
```

5.5 Summary

This chapter evaluated TigC by using different applications. First it described the implementations of two workloads used in the performance evaluation: Strassen's Matrix Multiplication and n-Queens problem. Then, it compared the differences between using PPU only and simple call mode, with different sizes of input data. After that, the parallel call mode was evaluated with an increasing SPU count. The results show that the performance gained from multiprocessing is consistent with Amdahl's Law. By enabling the data transfer optimization, the time cost TigC processes both types of source files, it provides a global view to do further optimizations. Then, this chapter discussed two potential optimizations that can be done with TigC.

6. Conclusion and Future Work

This study has presented the design, implementation, and performance evaluation of a Tiger compiler, TigC, on the Cell B.E. system. It extends the standard Tiger language to allow function definitions for both the PPU and the SPU architectures. The compiler includes two backends tailored to emit correct assembly instructions and to automatically generate processor switch and data transfer operations. An RCS generator is implemented to read information about all the functions defined and to create stubs for triggering the SPU calls. It then investigated how to call a specified SPU method with a generic entry function, using a predefined input buffer layout for the SPU programs to support array data.

The parallel call mode is introduced to more fully utilize the computational power of Cell B.E. systems. A new loop syntax is added to Tiger to distribute the input data to different SPUs. Instead of calling an SPU function and waiting for the completion, now the program is able to create multiple threads, where each thread is assigned one SPU to process part of the input data. Finally, overlapped I/O is implemented wherein the computation and the transfer for the next dataset between the main memory and the local storage can thus run in parallel.

The most important benefit from TigC is that it provides a general purpose compiler architecture for asymmetric processors. Programmers are not required to provide two different sets of source files. All function definitions can be put into a single file and processed by a single compiler. Although this feature is not novel, it is not available on Cell B.E. systems yet. Calling functions on a different processor does not need explicitly programmed context initialization and program loading. The compiler hides all the details about the system-specific operations. This capability provides greater knowledge of the program and offers a better platform for future optimizations.

In the future, there are several places that can be extended. TigC currently does not take advantage of the SIMD capability on the PPU and the SPU cores. For operations, such as some multimedia operations with the same simple addition or subtraction to large amounts of data, SIMD offers data level parallelism and huge performance improvements. This is especially important for the SPU, because all its registers are SIMD registers. Even scalar value is processed with these registers, which is less efficiency than generic scalar architecture because the load and store instructions require data alignment. Some decision-making component can be implemented to automatically recognize functions that should be executed on the SPUs. This addition will eliminate the usage of extended keywords. Another improvement is about the entry function. Instead of starting from the same entry in the SPU binary, the Cell B.E. SDK allows us to specify an entry offset in the binary, which makes it possible to call a function directly. Finally, the compiler should be extended to allow communications between the SPUs, so that, when necessary, different steps of an operation can be distributed.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson/Addison Wesley, Boston, MA, USA, 2007.

[2] G. M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proc. SJC '67*, pages 483–385. ACM, 1967.

[3] M. Annavaram, E. Grochowski, and J. Shen. Mitigating Amdahl's law through EPI throttling. In *Proc. ISCA '05*, pages 298–309. IEEE, 2005.

[4] A. W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, New York, NY, USA, 1st edition, 1998.

[5] E. Ayguade et al. A Proposal to Extend the OpenMP Tasking Model for Heterogeneous Architectures. In *Proc. IWOMP '09*, pages 154–167. Springer-Verlag, 2009.

[6] J.-L. Baer. Multiprocessing Systems. *IEEE Transactions on Computers*, 100(12):1271–1277, 1976.

[7] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes. The ILLIAC IV Computer. *IEEE Transactions on Computers*, 17(8):746–757, 1968.

[8] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. CellSs: a Programming Model for the Cell B.E. Architecture. In *Proc. SC '06*, pages 5–5. ACM, 2006.

[9] G. Blake, R. G. Dreslinski, T. Mudge, and K. Flautner. Evolution of Thread-level Parallelism in Desktop Applications. In *Proc. ISCA '10*, pages 302–313. ACM, 2010.

[10] W. J. Bolosky and M. L. Scott. False sharing and its effect on shared memory performance. In *Proc.SEDMS '93*. Usenix Association, 1993.

[11] J. R. Callahan and J. M. Purtilo. Using an architectural approach to integrate heterogeneous, distributed software components. Technical report, NASA/WVU Software IV & V Facility Software Research Laboratory, 1995.

[12] R. Chandra et al. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[13] L. Courtès. C Language Extensions for Hybrid CPU/GPU Programming with StarPU. *CoRR*, abs/1304.0878, 2013.

[14] L. Dagum and R. Menon. OpenMP: an Industry Standard API for Shared-memory Programming. *IEEE Computational Science & Engineering*, 5(1):46–55, 1998.

[15] T. Deepak Shekhar et al. Comparison of Parallel Programming Models for Multicore Architectures. In *IPDPSW '11*, pages 1675–1682. IEEE Computer Society, 2011.

[16] R. Duncan. A Survey of Parallel Computer Architectures. *Computer*, 23(2):5–16, 1990.

[17] A. E. Eichenberger et al. Optimizing Compiler for the Cell Processor. In *Proc. PACT '05*, pages 161–172. IEEE, 2005.

[18] H. Espeland. *Investigation of Parallel Programming on Heterogeneous Multiprocessors*. Master's thesis, Department of Informatics, University of Oslo, Oslo, Norway, 2008.

[19] P. B. Gibbons. A Stub Generator for Multilanguage RPC in Heterogeneous Environments. *IEEE Transactions on Software Engineering*, 13(1):77–87, 1987.

[20] M. W. Hall et al. Maximizing Multiprocessor Performance with the SUIF Compiler. *Computer*, 29(12):84–89, 1996.

[21] M. D. Hill and M. R. Marty. Amdahl's Law in the Multicore Era. *Computer*, 41(7):33–38, 2008.

[22] M.-C. Hsiao et al. Implementation of a Portable Parallelizing Compiler with Loop Partitioning. In *ICPADS '94*, pages 333–338. IEEE, 1994.

[23] J. R. Humphrey, D. K. Price, K. E. Spagnoli, A. L. Paolini, and E. J. Kelmelis. CULA: Hybrid GPU Accelerated Linear Algebra Routines. In *SPIE '10*, pages 770502–770502. SPIE, 2010.

[24] IBM. *PowerPC Architecture Book*. `http://www.ibm.com/developerworks/systems/library/es-archguide-v2.html`, 2005.

[25] IBM. *Cell Broadband Engine Architecture*. `https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/1AEEE1270EA2776387257060006E61BA`, 2007.

[26] IBM. *SPU Application Binary Interface Specification*. `https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/02E544E65760B0BF87257060006F8F20`, 2008.

[27] IBM. *SPU Assembly Language Specification*. `https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/EFA2B196893B550787257060006FC9FB`, 2008.

[28] IBM. *Cell Broadband Engine Programming Handbook Including the PowerX-Cell 8i Processor*. `https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/7A77CCDF14FE70D5852575CA0074E8ED`, 2009.

[29] C. R. Johns and D. A. Brokenshire. Introduction to the Cell Broadband Engine Architecture. *IBM Journal of Research and Development*, 51(5):503–519, 2007.

[30] S. Lee, S.-J. Min, and R. Eigenmann. OpenMP to GPGPU: a Compiler Framework for Automatic Translation and Optimization. *ACM SIGPLAN Notices*, 44(4):101–110, 2009.

[31] B. Lewis and D. J. Berg. *Multithreaded Programming with Pthreads*. Prentice-Hall, Inc., 1998.

[32] D. J. Lilja. Exploiting the Parallelism Available in Loops. *Computer*, 27(2):13–26, 1994.

[33] D. Luebke. CUDA: Scalable Parallel Programming for High-Performance Scientific Computing. In *Proc. ISBI '08*, pages 836–838. IEEE, 2008.

[34] F. Mao and X. Shen. LU Decomposition on Cell Broadband Engine: an Empirical Study to Exploit Heterogeneous Chip Multiprocessors. In *Proc. NPC '10*, pages 61–75. Springer, 2010.

[35] C. Meyers, C. Clack, and E. Poon. *Programming with Standard ML*. Prentice-Hall, Inc., 1993.

[36] Microsoft. *MIPS Stack Frame Layout*. `http://msdn.microsoft.com/en-us/library/aa448710.aspx`, 2006.

[37] B. J. Nelson. *Remote Procedure Call*. PhD thesis, Palo Alto Research Center, Carnegie Mellon University, Pittsburgh, PA, USA, 1981.

[38] NVIDA. *Parallel Programming and Computing Platform: CUDA*. `http://www.nvidia.com/object/cuda_home_new.html`, 2013.

[39] K. O'Brien, K. O'Brien, Z. Sura, T. Chen, and T. Zhang. Supporting OpenMP on Cell. *International Journal of Parallel Programming*, 36(3):289–311, 2008.

[40] OpenACC. *The OpenACC Application Programming Interface, v1.0* . CAPS Enterprise, Cray Inc., NVIDIA, and the Portland Group, 2011.

[41] D. Pham et al. The Design and Implementation of a First-Generation Cell Processor. In *Proc. ISSCC '05*, pages 184–592. IEEE, 2005.

[42] B. R. Rau and J. A. Fisher. Instruction-Level Parallel Processing: History, Overview, and Perspective. *The Journal of Supercomputing*, 7(1-2):9–50, 1993.

[43] B. R. Rau and J. A. Fisher. Instruction-Level Parallelism. In *Encyclopedia of Computer Science*, pages 883–887. John Wiley and Sons Ltd., 2003.

[44] B. Reistad and D. K. Gifford. Static dependent costs for estimating execution time. In *LFP '94*, pages 65–78. ACM, 1994.

[45] M. W. Riley, J. D. Warnock, and D. F. Wendel. Cell Broadband Engine Processor: Design and Implementation. *IBM Journal of Research and Development*, 51(5):545–557, 2007.

[46] R. R. Schaller. Moore's Law: Past, Present and Future. *IEEE Spectrum*, 34(6):52–59, 1997.

[47] T. R. Scogland, B. Rountree, W.-c. Feng, and B. R. de Supinski. Heterogeneous Task Scheduling for Accelerated OpenMP. In *IPDPS '12*, pages 144–155. IEEE, 2012.

[48] A. L. Sepillo. A Comparative Study on Symmetric and Asymmetric Multiprocessors. Technical report, University of the Philippines, 2006.

[49] W. Stallings. *Computer Organization and Architecture: Designing for Performance*. Prentice-Hall, Inc., 2005.

[50] W. J. Watson. The TI ASC: A Highly Modular and Flexible Super Computer Architecture. In *AFIPS '72 (Fall, part I)*, pages 221–228. ACM, 1972.

[51] H. Wei and J. Yu. Loading OpenMP to Cell: An Effective Compiler Framework for Heterogeneous Multi-Core Chip. In *IWOMP '07*, pages 129–133. Springer-Verlag, 2008.

[52] N. Wirth. Program Development by Stepwise Refinement. *Communications of the ACM*, 14(4):221–227, 1971.

[53] C. Yang and C.-C. Lim. Speculative Parallel Threading Architecture and Compilation. In *ICPPW '05*, pages 285–294. IEEE Computer Society, 2005.

[54] H. Zhong, M. Mehrara, S. A. Lieberman, and S. A. Mahlke. Uncovering Hidden Loop Level Parallelism in Sequential Applications. In *HPCA '08*, pages 290–301. IEEE, 2008.

A. Frontend Modification of TigC

In this chapter, we show the modification in the frontend of TigC in details to support PPU and SPU functions.

A.1 Lexical Analysis

Listing A.1: tiger.lex

```
(* tiger scanner *)

type pos = int;                                 (* %pos int given in tiger.grm *)
type svalue = Tokens.svalue
type ('a, 'b) token = ('a, 'b) Tokens.token
type lexresult = (svalue, pos) token;

val lineNum = ErrorMsg.lineNum
val linePos = ErrorMsg.linePos

fun nextLn pos = ( lineNum := !lineNum+1
                 ; linePos := pos :: !linePos)

fun ctrlAlphaU c = chr ((ord c) - (ord #"A") + 1)

fun ctrlAlphaL c = chr ((ord c) - (ord #"a") + 1)

fun ctrlChar c = let fun tr #"[" = 27
                       | tr #"\\" = 28
                       | tr #"]" = 29
                       | tr #"^" = 30
                       | tr #"_" = 31
                       | tr #"@" = 0
                       | tr _ = ( ErrorMsg.impossible "CTRL CHAR CONSISTENCY"
                                ; 0)
                 in chr (tr c)
                 end

val strBuf : (pos * string option) ref = ref (0, NONE)

fun strApp s = case !strBuf of
                   (_, NONE) => ErrorMsg.impossible "STRING CONSISTENCY"
                 | (l, SOME s') => strBuf := (l, SOME (s' ^ s))

fun err (p1,p2) = ErrorMsg.error p1

val comments : pos list ref = ref []

fun eof () = let val pos = hd(!linePos)
             in ( case !comments of
                      [] => ()
                    | [c] => ErrorMsg.error pos ("Unfinished comment starting at line " ^ (Int.toString c))
                    | (c::cs) => let fun loop [] = ( ErrorMsg.impossible "COMMENT CONSISTENCY"
                                                   ; "bogus" )
                                       | loop [c] = " and " ^ Int.toString c
                                       | loop (c::cs) = ", " ^ Int.toString c ^ loop cs
                                 in ErrorMsg.error pos ("Unfinished comments from lines " ^ Int.toString c ^ loop cs)
                                 end
                ; comments := []
                ; case !strBuf of
                      (_, NONE) => ()
                    | (l, SOME _) => ErrorMsg.error pos ("Unfinished string started at line " ^ (Int.toString l))
                ; strBuf := (0, NONE)
                ; Tokens.EOF(pos,pos) )
             end

%%
nonzero=[1-9];
```

```
digit=[0-9];
alpha=[a-zA-Z];
alphanum=[a-zA-Z0-9_];
whitespace=[ \009\011\012\013];
ctrl="\\^";
ctrlch={ctrl}[\\@^_[\]];

%s INITIAL COMMENT STRING FMTSEQ;

%header (functor TigerLexFun(structure Tokens : Tiger_TOKENS)  (* %name Tiger given in tiger.grm *));

%%
<INITIAL>","    => (Tokens.COMMA(yypos,yypos+ size yytext));
<INITIAL>":"    => (Tokens.COLON(yypos,yypos+ size yytext));
<INITIAL>";"    => (Tokens.SEMICOLON(yypos,yypos+ size yytext));
<INITIAL>"("    => (Tokens.LPAREN(yypos,yypos+ size yytext));
<INITIAL>")"    => (Tokens.RPAREN(yypos,yypos+ size yytext));
<INITIAL>"["    => (Tokens.LBRACK(yypos,yypos+ size yytext));
<INITIAL>"]"    => (Tokens.RBRACK(yypos,yypos+ size yytext));
<INITIAL>"{"    => (Tokens.LBRACE(yypos,yypos+ size yytext));
<INITIAL>"}"    => (Tokens.RBRACE(yypos,yypos+ size yytext));
<INITIAL>"."    => (Tokens.DOT(yypos,yypos+ size yytext));
<INITIAL>"+"    => (Tokens.PLUS(yypos,yypos+ size yytext));
<INITIAL>"-"    => (Tokens.MINUS(yypos,yypos+ size yytext));
<INITIAL>"*"    => (Tokens.TIMES(yypos,yypos+ size yytext));
<INITIAL>"/"    => (Tokens.DIVIDE(yypos,yypos+ size yytext));
<INITIAL>"="    => (Tokens.EQ(yypos,yypos+ size yytext));
<INITIAL>"<>"   => (Tokens.NEQ(yypos,yypos+ size yytext));
<INITIAL>"<"    => (Tokens.LT(yypos,yypos+ size yytext));
<INITIAL>"<="   => (Tokens.LE(yypos,yypos+ size yytext));
<INITIAL>">"    => (Tokens.GT(yypos,yypos+ size yytext));
<INITIAL>">="   => (Tokens.GE(yypos,yypos+ size yytext));
<INITIAL>"&"    => (Tokens.AND(yypos,yypos+ size yytext));
<INITIAL>"|"    => (Tokens.OR(yypos,yypos+ size yytext));
<INITIAL>":="   => (Tokens.ASSIGN(yypos,yypos+ size yytext));

<INITIAL>var    => (Tokens.VAR(yypos,yypos+ size yytext));
<INITIAL>while  => (Tokens.WHILE(yypos,yypos+ size yytext));
<INITIAL>for    => (Tokens.FOR(yypos,yypos+ size yytext));
<INITIAL>spufor => (Tokens.SPUFOR(yypos,yypos+ size yytext));
<INITIAL>to     => (Tokens.TO(yypos,yypos+ size yytext));
<INITIAL>break  => (Tokens.BREAK(yypos,yypos+ size yytext));
<INITIAL>let    => (Tokens.LET(yypos,yypos+ size yytext));
```

```
<INITIAL>in      => (Tokens.IN(yypos,yypos+ size yytext));
<INITIAL>end     => (Tokens.END(yypos,yypos+ size yytext));
<INITIAL>type    => (Tokens.TYPE(yypos,yypos+ size yytext));
<INITIAL>array   => (Tokens.ARRAY(yypos,yypos+ size yytext));
<INITIAL>if      => (Tokens.IF(yypos,yypos+ size yytext));
<INITIAL>then    => (Tokens.THEN(yypos,yypos+ size yytext));
<INITIAL>else    => (Tokens.ELSE(yypos,yypos+ size yytext));
<INITIAL>do      => (Tokens.DO(yypos,yypos+ size yytext));
<INITIAL>of      => (Tokens.OF(yypos,yypos+ size yytext));
<INITIAL>nil     => (Tokens.NIL(yypos,yypos+ size yytext));
<INITIAL>spu_function   => (Tokens.SPU_FUNCTION(yypos,yypos+ size yytext));
<INITIAL>ppu_function   => (Tokens.PPU_FUNCTION(yypos,yypos+ size yytext));

<INITIAL>({alpha}{alphanum}*)   => (Tokens.ID(yytext, yypos, yypos + size yytext));
<INITIAL>({digit}{digit}*)      => ( case Int.fromString yytext of
                                        NONE => ( ErrorMsg.error yypos ("number not integer " ^ yytext)
                                                ; continue())
                                      | SOME z => Tokens.INT(z,yypos,yypos+size yytext));

<INITIAL>"/*"   => ( YYBEGIN COMMENT
                   ; comments := yypos :: !comments
                   ; continue());

<COMMENT>"/*"   => ( comments := yypos :: !comments
                   ; continue());

<COMMENT>\n     => ( nextLn yypos
                   ; continue());

<COMMENT>"*/"   => ( case !comments of
                         [] => ( YYBEGIN INITIAL
                               ; ErrorMsg.impossible "COMMENTS STATE CONSISTENCY")
                       | [c] => ( YYBEGIN INITIAL
                                ; comments := [])
                       | (c::cs) => comments := cs
                   ; continue());

<COMMENT>.      => (continue());

<INITIAL>\"     => ( YYBEGIN STRING
                   ; strBuf := (!lineNum, SOME "")
                   ; continue());
<STRING>\"      => ( YYBEGIN INITIAL
                    ; case !strBuf of
                          (_, NONE) => ( ErrorMsg.impossible "STRING STATE CONSISTENCY"
                                       ; continue())
                        | (1, SOME s) => ( strBuf := (0, NONE)
                                         ; Tokens.STRING(s, 1, size s)));
<STRING>\\n     => ( strApp "\n"
                    ; nextLn yypos
                    ; continue());
<STRING>\\t     => ( strApp "\t"
                    ; continue());
<STRING>\\\"    => ( strApp "\\"
                    ; continue());
<STRING>({ctrl}[a-z])   => ( strApp (str (ctrlAlphaL (String.sub (yytext, 2))))
                            ; continue());
<STRING>({ctrl}[A-Z])   => ( strApp (str (ctrlAlphaU (String.sub (yytext, 2))))
                            ; continue());
<STRING>({ctrlch})      => ( strApp (str (ctrlChar (String.sub (yytext, 2))))
                            ; continue());
<STRING>(\\{digit}{digit}{digit})       => ( case Int.fromString (substring (yytext, 1, 3))
                                              of NONE => ErrorMsg.impossible "DECIMAL CONTROL STRING CONSISTENCY"
                                               | SOME n => if n>=0 andalso n<128 then strApp (str (chr n))
                                                           else ErrorMsg.error yypos "invalid decimal control character"
                                            ; continue());
<STRING>(\\{whitespace})        => ( YYBEGIN FMTSEQ
                                    ; continue());
<STRING>\\\n    => ( YYBEGIN FMTSEQ
                    ; continue());
<STRING>\\\\    => ( strApp "\\"
                    ; continue());
<STRING>\\      => ( ErrorMsg.error yypos "invalid escape sequence"
                    ; continue());
<STRING>\n      => ( ErrorMsg.error yypos "newline in string constant"
                    ; nextLn yypos
                    ; continue());
<STRING>.       => ( strApp yytext
                    ; continue());
<FMTSEQ>({whitespace})  => (continue());
<FMTSEQ>\n      => ( nextLn yypos
                    ; continue());
<FMTSEQ>\\      => ( YYBEGIN STRING
                    ; continue());
<FMTSEQ>.       => ( YYBEGIN STRING
                    ; ErrorMsg.error yypos ("illegal format character " ^ yytext)
```

```
                      ; strApp yytext
                      ; continue());
<INITIAL>\n     => ( nextLn yypos
                      ; continue());
<INITIAL>({whitespace}) => (continue());
<INITIAL>.      => ( ErrorMsg.error yypos ("illegal character " ^ yytext)
                      ; continue());
```

A.2 Parser

Listing A.2: tiger.grm

```
(* Tiger grammar *)
structure A = Absyn
structure S = Symbol

fun mkSeqExp [(x, p)] = x
  | mkSeqExp xps = A.SeqExp xps

fun mkArrayExp (A.SimpleVar (i, _), exp1, exp2, pos) = A.ArrayExp { typ = i, size = exp1, init = exp2, pos = pos }
  | mkArrayExp (_, _, _, pos) = ( ErrorMsg.error pos "Invalid array element type"
                                ; A.NilExp )

type nexp = (S.symbol * A.exp * A.pos)

type tdec = { name : S.symbol, ty : A.ty, pos : A.pos}

type optty = (S.symbol * A.pos) option

type pexp = (A.exp * A.pos)

%%
%term EOF
    | ID of string
```

```
        | INT of int | STRING of string
        | COMMA | COLON | SEMICOLON | DOT
        | LPAREN | RPAREN | LBRACK | RBRACK | LBRACE | RBRACE
        | PLUS | MINUS | TIMES | DIVIDE | EQ | NEQ | LT | LE | GT | GE
        | AND | OR | ASSIGN
        | ARRAY | IF | THEN | ELSE | WHILE | FOR | SPUFOR | TO | DO | LET | IN | END | OF
        | BREAK | NIL
        | SPU_FUNCTION | PPU_FUNCTION | VAR | TYPE
        | UMINUS | DECL

%nonterm  program of A.exp
        | exp of A.exp
        | exps of pexp list
        | exp' of pexp list
        | dec of A.dec
        | decs of A.dec list
        | fdec of A.fundec
        | fdecs of A.fundec list
        | tdec of tdec
        | tdecs of tdec list
        | ty of A.ty
        | tann of S.symbol
        | optty of (S.symbol * A.pos) option
        | field of A.field
        | fields of A.field list
        | field' of A.field list
        | var of A.var
        | args of A.exp list
        | arg' of A.exp list
        | nexp of nexp
        | nexps of nexp list
        | nexp' of nexp list

%keyword WHILE DO FOR SPUFOR TO BREAK
         LET IN END
         SPU_FUNCTION PPU_FUNCTION VAR TYPE ARRAY OF
         IF THEN ELSE
         NIL

%start program
%eop EOF
%noshift EOF

```

```
%pos int
%verbose
%name Tiger

%right    DECL
%right    TYPE SPU_FUNCTION

%right    DO
%left     OF
%nonassoc ASSIGN THEN
%nonassoc ELSE
%left     OR
%left     AND
%nonassoc EQ NEQ GT LT GE LE
%left     PLUS MINUS
%left     TIMES DIVIDE
%nonassoc UMINUS

%prefer THEN ELSE LPAREN

%value ID ("bogus")
%value INT (1)
%value STRING ("")

%%

program : exp                                               (exp)

exp : var                                                   (A.VarExp var)
    | var ASSIGN exp                                        (A.AssignExp { var = var, exp = exp, pos = ASSIGNleft })
    | ID LPAREN args RPAREN                                 (A.CallExp { func = S.symbol ID, args = args, pos = LPARENleft
      })
    | ID LBRACE nexps RBRACE                                (A.RecordExp { fields = nexps, typ = S.symbol ID, pos =
      LBRACEleft })
    | var LBRACK exp RBRACK OF exp                          (mkArrayExp (var, exp1, exp2, LBRACKleft))

    | SPUFOR ID OF LPAREN args RPAREN                       (A.SpuForExp { func = S.symbol ID, args = args, pos = LPARENleft
      })

    | NIL                                                   (A.NilExp)
    | INT                                                   (A.IntExp INT)
    | STRING                                                (A.StringExp (STRING, STRINGleft))

```

```
      | MINUS exp                         %prec UMINUS    (A.OpExp { left = A.IntExp 0, oper = A.MinusOp, right = exp, pos
           = MINUSleft })
      | exp PLUS exp                                      (A.OpExp { left = exp1, oper = A.PlusOp, right = exp2, pos =
          PLUSleft })
      | exp MINUS exp                                     (A.OpExp { left = exp1, oper = A.MinusOp, right = exp2, pos =
          MINUSleft })
      | exp TIMES exp                                     (A.OpExp { left = exp1, oper = A.TimesOp, right = exp2, pos =
          TIMESleft })
      | exp DIVIDE exp                                    (A.OpExp { left = exp1, oper = A.DivideOp, right = exp2, pos =
          DIVIDEleft })
      | exp EQ exp                                        (A.OpExp { left = exp1, oper = A.EqOp, right = exp2, pos =
          EQleft })
      | exp NEQ exp                                       (A.OpExp { left = exp1, oper = A.NeqOp, right = exp2, pos =
          NEQleft })
      | exp GT exp                                        (A.OpExp { left = exp1, oper = A.GtOp, right = exp2, pos =
          GTleft })
      | exp LT exp                                        (A.OpExp { left = exp1, oper = A.LtOp, right = exp2, pos =
          LTleft })
      | exp GE exp                                        (A.OpExp { left = exp1, oper = A.GeOp, right = exp2, pos =
          GEleft })
      | exp LE exp                                        (A.OpExp { left = exp1, oper = A.LeOp, right = exp2, pos =
          LEleft })

      | exp AND exp                                       (A.IfExp { test = exp1, then' = exp2, else' = SOME (A.IntExp 0),
           pos = ANDleft })
      | exp OR exp                                        (A.IfExp { test = exp1, then' = A.IntExp 1, else' = SOME exp2,
          pos = ORleft })

      | IF exp THEN exp ELSE exp                          (A.IfExp { test = exp1, then' = exp2, else' = SOME exp3, pos =
          IFleft })
      | IF exp THEN exp                                   (A.IfExp { test = exp1, then' = exp2, else' = NONE, pos = IFleft
           })

      | WHILE exp DO exp                                  (A.WhileExp { test = exp1, body = exp2, pos = WHILEleft})
      | BREAK                                             (A.BreakExp BREAKleft)
   (* | FOR ID ASSIGN exp TO exp DO exp                   ...         *)
      | LET decs IN exps END                              (A.LetExp { decs = decs, body = mkSeqExp exps, pos = LETleft })
      | LPAREN exps RPAREN                                (mkSeqExp exps)

  dec : tdecs                    (A.TypeDec tdecs)
      | VAR ID optty ASSIGN exp   (A.VarDec { name = S.symbol ID,
                                              escape = ref true,
                                              typ = optty,
                                            init = exp,
                                            pos = VARleft })
    | fdecs                     (A.FunctionDec fdecs)

decs : (* empty *)      ([])
     | dec decs         (dec :: decs)

tdecs : tdec            %prec DECL      ([tdec])
      | tdec tdecs                      ((tdec :: tdecs))

tdec : TYPE ID EQ ty    ({ name = S.symbol ID, ty = ty, pos = TYPEleft })

fdecs : fdec            %prec DECL      ([fdec])
      | fdec fdecs                      ((fdec :: fdecs))

fdec : SPU_FUNCTION ID LPAREN fields RPAREN optty EQ exp        ({ name = S.symbol ID,
                                                                   params = fields,
                                                                   result = optty,
                                                                   body = exp,
                                                                   pos = SPU_FUNCTIONleft,
                                                                   tag = A.SPU_Func })
     | PPU_FUNCTION ID LPAREN fields RPAREN optty EQ exp        ({ name = S.symbol ID,
                                                                   params = fields,
                                                                   result = optty,
                                                                   body = exp,
                                                                   pos = PPU_FUNCTIONleft,
                                                                   tag = A.PPU_Func })

var : ID                        (A.SimpleVar (S.symbol ID, IDleft))
    | var DOT ID                (A.FieldVar (var, S.symbol ID, DOTleft))
    | var LBRACK exp RBRACK     (A.SubscriptVar (var, exp, LBRACKleft))

optty : (* empty *)             (NONE)
      | tann                    (SOME(tann, tannleft))

ty : ID                         (A.NameTy (S.symbol ID, IDleft))
   | LBRACE fields RBRACE       (A.RecordTy fields)
   | ARRAY OF ID                (A.ArrayTy (S.symbol ID, ARRAYleft))

tann : COLON ID                 (S.symbol ID)

field : ID tann                 ({ name = S.symbol ID, escape = ref true, typ = tann, pos = tannleft })

```

```
fields : (* empty *)            ([])
       | field'                 (field')

field' : field                  ([field])
       | field COMMA field'     (field :: field')

exps : (* empty *)              ([])
     | exp'                     (exp')

exp' : exp                      ([(exp, expleft)])
     | exp SEMICOLON exp'       ((exp, expleft) :: exp')

args : (* empty *)              ([])
     | arg'                     (arg')

arg' : exp                      ([exp])
     | exp COMMA arg'           (exp :: arg')

nexp : ID EQ exp                ((S.symbol ID, exp, EQleft))

nexps : (* empty *)             ([])
      | nexp'                   (nexp')

nexp' : nexp                    ([nexp])
      | nexp COMMA nexp'        (nexp :: nexp')
```

B. Backend Implementation of TigC

Here we show the details of the backend implementation of TigC for both PPU and SPU architectures.

B.1 Cell B.E. Frame Layout

This file generates the PPU and SPU frame layout.

Listing B.1: cbframe.sml

```
structure SPUFrame =
struct
    structure M = Temp

    fun makeReg (s, n) = ("$" ^ (Int.toString (n+4)), M.newtemp ())
    fun makeRegs (s, n) = if (0=n)
                          then [makeReg (s, 0)]
                          else (makeReg (s, n))::makeRegs (s, n-1)

    fun makeReg' (s, n) = ("$" ^ (Int.toString (n+80)), M.newtemp())
    fun makeRegs' (s, n) = if (0=n)
                           then [makeReg' (s, 0)]
```

```
                               else (makeReg' (s, n))::makeRegs' (s, n-1)

        fun makeReg'' (s, n) = ("$" ^ (Int.toString (n+75)), M.newtemp())
        fun makeRegs'' (s, n) = if (0=n)
                                then [makeReg'' (s, 0)]
                                else (makeReg'' (s, n))::makeRegs'' (s, n-1)

        val argregs = rev (makeRegs ("r", 6))
        val calleesaveregs = rev (makeRegs' ("r", 10))
        val callersaveregs = rev (makeRegs'' ("r", 4))

        val wordSize = 16
    end

    structure PPUFrame =
    struct
        structure M = Temp

        fun makeReg (s, n) = (Int.toString (n+4), M.newtemp ())
        fun makeRegs (s, n) = if (0=n)
                              then [makeReg (s, 0)]
                              else (makeReg (s, n))::makeRegs (s, n-1)

        fun makeReg' (s, n) = (Int.toString (n+14), M.newtemp())
        fun makeRegs' (s, n) = if (0=n)
                               then [makeReg' (s, 0)]
                               else (makeReg' (s, n))::makeRegs' (s, n-1)

        fun makeReg'' (s, n) = (Int.toString (n+11), M.newtemp())
        fun makeRegs'' (s, n) = if (0=n)
                                then [makeReg'' (s, 0)]
                                else (makeReg'' (s, n))::makeRegs'' (s, n-1)

        val argregs = rev (makeRegs ("r", 6))           (* r4 -r10 are used for arguments *)
        val calleesaveregs = rev (makeRegs' ("r", 17))  (* r14-r32 are non-valitile *)
        val callersaveregs = rev (makeRegs'' ("r", 1))  (* r11-r12 are volatile *)

        val wordSize = 8
    end

    structure CBFrame : FRAME =
    struct
        structure E = ErrorMsg
```

```
structure A = Assem
structure M = Temp
structure MB = M.Table
structure R = Tree
structure S = Symbol
structure U = UnparseAbsyn

type register = string
val mainLabel = M.namedlabel "tigermain"

fun regName (s, _) = s
fun regTemp (_, t) = t

val SP = M.newtemp ()
fun SPreg ftag = (case ftag of Absyn.SPU_Func => "$sp" | Absyn.PPU_Func => "1", SP)

val LR = M.newtemp ()
fun LRreg ftag = (case ftag of Absyn.SPU_Func => "$lr" | Absyn.PPU_Func => "0", LR)

val RV = M.newtemp()
fun RVreg ftag = (case ftag of Absyn.SPU_Func => "$3" | Absyn.PPU_Func => "3", RV)

fun specialregs ftag = [ LRreg ftag, SPreg ftag, RVreg ftag ]
fun argregs ftag = [(RVreg ftag)] @ (case ftag of Absyn.SPU_Func => SPUFrame.argregs | Absyn.PPU_Func => PPUFrame.
    argregs)
fun calleesaveregs ftag = case ftag of Absyn.SPU_Func => SPUFrame.calleesaveregs | Absyn.PPU_Func => PPUFrame.
    calleesaveregs
fun callersaveregs ftag = case ftag of Absyn.SPU_Func => SPUFrame.callersaveregs | Absyn.PPU_Func => PPUFrame.
    callersaveregs
fun regs ftag = ((calleesaveregs ftag) @ (callersaveregs ftag) @ (argregs ftag) @ (specialregs ftag))

(* both ppu and spu use 8 registers 3-10 for argument passing *)
fun args ftag = map regTemp (argregs ftag)
fun calleesaves ftag = map regTemp (case ftag of Absyn.SPU_Func => SPUFrame.calleesaveregs
                                               | Absyn.PPU_Func => PPUFrame.calleesaveregs)
fun callersaves ftag = map regTemp (case ftag of Absyn.SPU_Func => SPUFrame.callersaveregs
                                               | Absyn.PPU_Func => PPUFrame.callersaveregs)
fun registers ftag = map regName (regs ftag)
fun calldefs ftag = [LR, RV]@(callersaves ftag)

fun tempMap {name, fname, formals, size, moves, saves, ftag} = foldr (fn ((s, m), t) => MB.enter (t, m, s))
                                                                     MB.empty
                                                                     (regs ftag)
```

```

fun tempName frame t = case (MB.look (tempMap frame, t))
                   of NONE => M.makestring t
                    | SOME s => s

datatype access = InFrame of int
                | InReg of M.temp

type frame = { name : M.label
             , fname : string
             , formals : access list
             , size : int ref
             , moves : R.stm (* moves to get InReg args into the argument temporaries $a0 -- $a3 *)
             , saves : R.exp list (* pre-allocated memory locations for saving caller-save registers *)
             , ftag : Absyn.ftag}

fun formals {name, fname, formals, size, moves, saves, ftag} = formals

fun fname   {name, fname, formals, size, moves, saves, ftag} = fname

fun name    {name, fname, formals, size, moves, saves, ftag} = name

fun size    {name, fname, formals, size, moves, saves, ftag} = !size

fun moves   {name, fname, formals, size, moves, saves, ftag} = moves

fun ftag    {name, fname, formals, size, moves, saves, ftag} = ftag

fun wordSize ftag = case ftag of Absyn.SPU_Func => SPUFrame.wordSize
                               | Absyn.PPU_Func => PPUFrame.wordSize

fun postinc r v = let val n = !r
                  in ( r := n + v
                     ; n
                     )
                  end

fun allocLocal { name
               , fname
               , formals
               , size
               , moves
               , saves
```

```
                        , ftag} true = InFrame (postinc size (case ftag of Absyn.SPU_Func => SPUFrame.wordSize
                                                                          | Absyn.PPU_Func => PPUFrame.wordSize))
          | allocLocal _ false = InReg (M.newtemp ())

        fun externalCall (s, args) = R.CALL (R.NAME (Temp.namedlabel ("_" ^ s)), args)

        fun exp (InFrame k) sp = R.MEM (R.BINOP (R.PLUS, sp, (R.CONST k)))
          | exp (InReg r) _ = R.TEMP r

        fun staticLink {name,fname,formals=[],size, moves, saves, ftag}                 = E.impossible "NO STATIC LINK"
          | staticLink {name,fname,formals=(sl::realFormals), size, moves, saves, ftag} = sl

        fun newFrame {name, fname, formals, tag} =
            let fun placeFormal (_,     (offset, [],          formals, stm)) = ( TextIO.output (TextIO.stdErr
                                                                                                 , "out of registers: offset "
                                                                                                      ^ Int.toString offset ^
                                                                                                     "\n")
                                                                                 ; ((case tag of Absyn.SPU_Func => offset+
                                                                                    SPUFrame.wordSize
                                                                                               | Absyn.PPU_Func => offset+
                                                                                                  PPUFrame.wordSize)
                                                                                               , []
                                                                                               , (InFrame offset)::formals
                                                                                               , stm
                                                                                    )
                                                                                  )
                    | placeFormal (true,  (offset, rs,          formals, stm)) = ( TextIO.output (TextIO.stdErr, "in-frame at
                         " ^ Int.toString offset ^ "\n")
                                                                                  ; ((case tag of Absyn.SPU_Func => offset+
                                                                                     SPUFrame.wordSize
                                                                                               | Absyn.PPU_Func => offset+
                                                                                                  PPUFrame.wordSize)
                                                                                      , rs
                                                                                      , (InFrame offset)::formals
                                                                                      , stm
                                                                                     )
                                                                                   )
                    | placeFormal (false, (offset, ((_,r)::rs), formals, stm)) = ( TextIO.output (TextIO.stdErr, "in-register
                         " ^ Int.toString (length rs) ^ "\n")
                                                                                   ; let val tmp = M.newtemp ()
                                                                                     in (offset, rs, (InReg tmp) ::formals, (R
                                                                                        .MOVE (R.TEMP tmp, R.TEMP r))::stm)
                                                                                     end
```

```
                                                                        )
            in let val (saves, _, formals, stm) = foldl placeFormal
                                                    (case tag of Absyn.SPU_Func => 32
                                                               | Absyn.PPU_Func => 128, argregs tag, [], [])
                                                    formals
               in let val (locals, saveExps) = if name = mainLabel
                                               then (saves, [])
                                               else foldr (fn (_, (saves, exps)) => ( (case tag of Absyn.SPU_Func => saves+
                                                   SPUFrame.wordSize
                                                                                                | Absyn.PPU_Func => saves+
                                                                                                    PPUFrame.wordSize)
                                                                                    , (exp (InFrame saves) (R.TEMP SP))::
                                                                                        exps
                                                                                    )
                                                          )
                                                          (saves, [])
                                                          (callersaves tag)
                   in { name    = name
                      , fname   = fname
                      , formals = rev formals
                      , size    = ref locals
                      , moves   = R.seq stm
                      , saves   = saveExps
                      , ftag    = tag}
                   end
               end
            end

        fun formalString (InFrame n) = (Int.toString n) ^ "(sp)"
          | formalString (InReg t)   = M.makestring t

        fun  frameStats { name
                        , fname
                        , formals
                        , size
                        , moves
                        , saves
                        , ftag} = "#FRAME " ^ (case ftag of Absyn.PPU_Func => "<ppu> "
                                                          | Absyn.SPU_Func => "<spu> ") ^ S.name name ^ " \"" ^ fname ^ "\"
                                                              "
                                    ^ "(" ^ U.commaSep (map formalString formals)
                                    ^ ") of size " ^ Int.toString (!size) ^ "\n"
```

```
(* generate the callee-saves move instructions *)
fun procEntryExit1 (frame as {name
                             ,fname
                             ,formals
                             ,size
                             ,moves
                             ,saves
                             ,ftag}
                             , body) = if name = mainLabel
                                       then body
                                       else foldr (fn (r, stm) => let val place = exp (allocLocal frame true) (R.
                                           TEMP SP)
                                                                      val rtemp = R.TEMP r
                                                                  in R.seq [ R.MOVE (place, rtemp),
                                                                             stm,
                                                                             R.MOVE (rtemp, place) ]
                                                                  end)
                                                  (R.SEQ (moves, body))
                                                  (calleesaves ftag)

fun procEntryExit2 ({ name
                    , fname
                    , formals
                    , size
                    , moves
                    ,saves
                    ,ftag}
                    , body) = if name = mainLabel
                              then body
                              else (A.OPER { assem = "# START\n",
                                             src=[],
                                             dst=[LR, SP]@calleesaves ftag@args ftag,
                                             jump=NONE })
                                    :: body
                                    @ [A.OPER { assem = "# END\n",
                                                src=[LR, SP, RV]@calleesaves ftag,
                                                dst=[],
                                                jump=SOME[] }]

fun spuProcProlog (name, fname, stat, size) = "#SPU_BEGIN\n.text\n.globl " ^ name ^ "\n" ^
                                              "#PROCEDURE\n" ^ name ^ ":\n" ^ "########"^" \"" ^ fname ^ "\" "  ^ "
                                                  :\n" ^
                                              stat ^ "stqd $lr, 16($sp)\n" ^
                                              "stqd $sp, -" ^ Int.toString (!size) ^ "($sp)\n" ^
                                              "ai $sp, $sp, -" ^ Int.toString (!size) ^ "\n"
fun spuProcEpilog (name, size) = "ai $sp, $sp, " ^ Int.toString(!size) ^ "\n" ^
                                 "lqd $sp, -" ^ Int.toString(!size) ^ "($sp)\n" ^
                                 "lqd $lr, 16($sp)\n" ^
                                 "bi $lr\n" ^
                                 "#SPU_END " ^ name ^ "\n\n"

fun ppuProcProlog (name, fname, stat, size) = ".align 2\n.globl " ^ fname ^ "\n.section \".opd\",\"aw\"\n.align 3\n"
    ^ fname ^ ":\n" ^
                                              ".quad " ^ name ^ ",.TOC.@tocbase,0\n.previous\n.type " ^ fname ^ ",
                                                  @function\n" ^
                                              "#PROCEDURE\n" ^ name ^ ":\n" ^ "########"^" \"" ^ name ^ "\" "  ^ ":\
                                                  n" ^
                                              stat ^ "stdu 1, -" ^ Int.toString(!size) ^ "(1)\n" ^
                                              "mflr 0\nstd 31, " ^ Int.toString(!size - 8) ^ "(1)\n" ^
                                              "std 0, " ^ Int.toString(!size + 16) ^ "(1)\n"
fun ppuProcEpilog (name, fname, size) = "ld 1,0(1)\n" ^
                                 "ld 0,16(1)\n" ^
                                 "mtlr 0\n" ^
                                 "blr\n" ^
                                 ".long 0\n.byte 0,0,0,0,128,1,0,1\n.size " ^ fname ^ ",.-" ^ name ^ "\n"
fun procEntryExit3 (frame as { name
                             , fname
                             , formals
                             , size
                             , moves
                             , saves
                             ,ftag}
                             , body) = if name = mainLabel
                                       then
                                       {
                                            prolog = ".section \".toc\",\"aw\"\n.section \".text\"\n.align 2\n.globl
                                                 main\n"
                                                   ^ ".section \".opd\",\"aw\"\n.align 3\nmain:\n"
                                                   ^ ".quad " ^ S.name name ^ ",.TOC.@tocbase,0\n.previous\n.type
                                                       main,@function\n"
                                                   ^ "#PROCEDURE\n" ^ S.name name ^":\n"^ "########"^" \"" ^ fname ^
                                                        "\" " ^ ":\n"
                                                   ^ frameStats frame
                                                   ^"stdu 1, -"^ Int.toString(!size) ^ "(1)\n"
                                                   ^"mflr 0\n"
                                                   ^ "std 0, " ^ Int.toString (!size + 16)^"(1)\n"
                                                  ^ "bl spu_init\n"

                                            , body = body
                                            , epilog = "bl spu_exit\n"
                                                      ^"ld 1,0(1)\n"
                                                      ^"ld 0,16(1)\n"
                                                      ^"mtlr 0\n"
                                                      ^"blr\n"
                                                      ^"#END " ^ S.name name ^ "\n.long 0\n.byte 0,0,0,0,128,1,0,1\
                                                          n.size main, .-"
                                                      ^ S.name name ^"\n" }

                                      else{ prolog = case ftag of Absyn.PPU_Func => ppuProcProlog(S.name name,
                                          fname, frameStats frame, size)
                                                                | Absyn.SPU_Func => spuProcProlog(S.name name,
                                                                    fname, frameStats frame, size)
                                          , body=body
                                          , epilog = case ftag of Absyn.PPU_Func => ppuProcEpilog(S.name name,
                                              fname, size)
                                                                | Absyn.SPU_Func => spuProcEpilog(S.name name, size
                                                                    ) }
fun string (l, s) =    ".data\n"
                      ^ ".align 4\n"
                      ^ (S.name l) ^ ": .string " ^ "\"" ^ (String.toCString s) ^ "\"" ^ "\n"

datatype frag = PROC of {body : Tree.stm, frame : frame }
              | STRING of M.label * string

fun showRegs frame stream = (app (fn regs => TextIO.output (stream, "#REGS: "
                                                          ^ U.commaSep (map (fn (r, t) => (r
                                                                                          ^ "=="
                                                                                          ^ (M.makestring t))) regs
                                                                                              )
                                                          ^ "\n"
                                                          )
                                 )
                                 [specialregs (ftag frame)
                                 , argregs (ftag frame)
                                 , calleesaveregs (ftag frame)
                                 , callersaveregs (ftag frame)
                                 ]
                            )

```

```
        fun showFrame stream frame = TextIO.output (stream, frameStats frame)

    end
```

B.2 Cell B.E. Code Generator

This is the code generation module which translates the IR into assembly code.

Listing B.2: codegen.sml

```
signature CODEGEN =
sig
    structure Frame : FRAME
    val codegen : Frame.frame -> Tree.stm ->Assem.instr list
end

structure CodeGen : CODEGEN =
struct
    structure Frame = CBFrame
    fun codegen (frame as {name, fname, formals, size, moves, saves, ftag}) stm =
        case ftag of Absyn.PPU_Func => ppuGen.codegen frame stm
                   | Absyn.SPU_Func => spuGen.codegen frame stm
end
```

B.2.1 PPU Code Generator

Listing B.3: ppuGen.sml

```
structure ppuGen =
struct

structure Frame = CBFrame

structure R = Tree
structure A = Assem
structure E = ErrorMsg
structure F = CBFrame
structure S = Symbol

fun codegen (frame as {name, fname, formals, size, moves, saves, ftag}) stm =

    let val ilist = ref []
        fun int z = if (z<0)
                    then "-" ^ (Int.toString (0 - z))
                    else Int.toString z
        fun emit x = ilist := x:: !ilist
        fun result gen = let val t = Temp.newtemp()
                         in ( gen t
                            ; t)
                         end
        val calldefs = F.calldefs Absyn.PPU_Func

        fun munchStm(R.SEQ(s1,s2)) = (munchStm s1; munchStm s2)
          | munchStm(R.LABEL l) = emit(A.LABEL{assem=S.name l ^ ":\n", lab=l})
          | munchStm(R.JUMP(e, ls)) = emit(A.OPER{assem="mtctr `s0\nbctr\n",
                                                  src=[munchExp e],
                                                  dst=[],
                                                  jump=SOME ls})
          | munchStm(R.CJUMP(R.NE, e1, R.CONST 0, t, f)) = emit(A.OPER{assem="cmpwi cr7,`s0,0\nbne cr7,"^S.name t^"\n",
                                                                        src=[munchExp e1],
                                                                        dst=[],
                                                                        jump=SOME [t,f]})
          | munchStm(R.CJUMP(R.NE, e1, R.CONST i, t, f)) = emit(A.OPER{assem="cmpwi cr7, `s0, "^int i^"\nbne cr7,"^S.
            name t^"\n",
                                                                 src=[munchExp e1],
                                                                 dst=[],
                                                                 jump=SOME [t,f]})
    | munchStm(R.CJUMP(R.NE, e1, e2, t, f)) = emit(A.OPER{assem="cmpw cr7, `s0, `s1\nbne cr7,"^S.name t^"\n",
                                                          src=[munchExp e1, munchExp e2],
                                                          dst=[],
                                                          jump=SOME [t,f]})

    | munchStm(R.CJUMP(R.EQ, e1, R.CONST 0, t, f)) = emit(A.OPER{assem="cmpwi cr7,`s0,0\nbeq cr7,"^S.name t^"\n",
                                                                 src=[munchExp e1],
                                                                 dst=[],
                                                                 jump=SOME [t, f]})
    | munchStm(R.CJUMP(R.EQ, e1, R.CONST i, t, f)) = emit(A.OPER{assem="cmpwi cr7, `s0, "^int i^"\nbeq cr7,"^S.
        name t^"\n",
                                                                 src=[munchExp e1],
                                                                 dst=[],
                                                                 jump=SOME [t,f]})
    | munchStm(R.CJUMP(R.EQ, e1, e2, t, f)) = emit(A.OPER{assem="cmpw cr7, `s0, `s1\nbeq cr7,"^S.name t^"\n",
                                                          src=[munchExp e1, munchExp e2],
                                                          dst=[],
                                                          jump=SOME [t,f]})

    | munchStm(R.CJUMP(R.GT, e1, R.CONST i, t, f)) = emit(A.OPER{assem="cmpwi cr7, `s0, "^int i^ "\nbgt cr7,"^S.
        name t^"\n",
                                                                 src=[munchExp e1],
                                                                 dst=[],
                                                                 jump=SOME [t,f]})
    | munchStm(R.CJUMP(R.GT, e1, e2, t, f)) = emit(A.OPER{assem="cmpw cr7,`s0,`s1\nbgt cr7,"^S.name t^"\n",
                                                       src=[munchExp e1, munchExp e2],
                                                       dst=[],
                                                       jump=SOME [t,f]})
    | munchStm(R.CJUMP(R.LT, e1, e2, t, f)) = emit(A.OPER{assem="cmpw cr7,`s0,`s1\nblt cr7,"^S.name t^"\n",
                                                       src=[munchExp e1, munchExp e2],
                                                       dst=[],
                                                       jump=SOME [t,f]})
    | munchStm(R.CJUMP(R.LE, e1, e2, t, f)) = emit(A.OPER{assem="cmpw cr7,`s0,`s1\nble cr7,"^S.name t^"\n",
                                                       src=[munchExp e1, munchExp e2],
                                                       dst=[],
                                                       jump=SOME [t,f]})
    | munchStm(R.MOVE(R.MEM(R.OFFSET(i,s)),e)) = emit(A.ARG{ assem = "std `a, -" ^ "`o(1)\n",
                                                             arg = munchExp e,
                                                             off = i,
                                                             fsize = s})
    | munchStm(R.MOVE(R.MEM(R.BINOP(R.PLUS, e1, R.CONST i)), e2)) = emit(A.OPER{assem="std ‘s1, " ^ int i ^"(‘s0)\n",
                                                                              src=[munchExp e1, munchExp e2],
                                                                              dst=[],
                                                                              jump=NONE})
    | munchStm(R.MOVE(R.MEM(R.BINOP(R.PLUS, R.CONST i, e1)), e2)) = emit(A.OPER{assem="std ‘s1, "^ int i ^"(‘s0)\n",
                                                                              src=[munchExp e1, munchExp e2],
                                                                              dst=[],
                                                                              jump=NONE})
    | munchStm(R.MOVE(R.MEM(R.BINOP(R.PLUS, e1, e2)), e3)) = emit(A.OPER{assem="stdx ‘s2, ‘s0, ‘s1\n",
                                                                       src=[munchExp e1, munchExp e2, munchExp e3],
                                                                       dst=[],
                                                                       jump=NONE})
    | munchStm(R.MOVE(R.MEM(R.CONST i), e2)) = let val r = result(fn r => emit(A.OPER{assem="li ‘d0, "^int i^"\n",
                                                                                    src=[],
                                                                                    dst=[r],
                                                                                    jump=NONE}))
                                           in
                                             emit(A.OPER{assem="std ‘s0, 0(‘d0)\n",
                                                         src=[munchExp e2],
                                                         dst=[r],
                                                         jump=NONE})
                                           end
    | munchStm(R.MOVE(R.MEM (e1), e2)) = emit(A.OPER{assem="std ‘s1, 0(‘s0)\n",
                                                      src=[munchExp e1, munchExp e2],
                                                      dst=[],
                                                      jump=NONE})
    | munchStm(R.MOVE(R.TEMP t, R.MEM(R.BINOP(R.PLUS, R.CONST i, e2)))) = emit(A.OPER{assem="ld ‘d0, "^ int i^"(‘s0) \n",
                                                                                    src=[munchExp e2],
                                                                                    dst=[t],
                                                                                    jump=NONE})
    | munchStm(R.MOVE(R.TEMP t, R.MEM(R.BINOP(R.PLUS, e2, R.CONST i)))) = emit(A.OPER{assem = "ld ‘d0, "^ int i^"(‘s0) \n",
                                                                                    src=[munchExp e2],
                                                                                    dst=[t],
                                                                                    jump=NONE})
    | munchStm(R.MOVE(R.TEMP t, R.MEM(R.CONST i))) = let val r = result(fn r => emit(A.OPER{assem="li ‘d0, "^int i^"\n",
                                                                                        src=[],
                                                                                        dst=[r],
                                                                                                      jump=NONE}))
                                                               in emit(A.OPER{assem="ld `d0, 0(`s0)\n",
                                                                              src=[r],
                                                                              dst=[t],
                                                                              jump=NONE})
                                                               end
              | munchStm(R.MOVE(R.TEMP t, R.CONST i)) = emit(A.OPER{assem="li `d0," ^int i^ "\n",
                                                                    src=[],
                                                                    dst=[t],
                                                                    jump=NONE})
              | munchStm(R.MOVE(R.TEMP t, R.MEM(e2))) = emit(A.OPER{assem="ld `d0, "^ int 0 ^"(`s0)\n",
                                                                    src=[munchExp e2],
                                                                    dst=[t],
                                                                    jump=NONE})
              | munchStm(R.MOVE(R.TEMP t, R.BINOP(R.PLUS, R.CONST i, e2))) = emit(A.OPER{assem="addi `d0, `s0, "^int i^"\n",
                                                                                         src=[munchExp e2],
                                                                                         dst=[t],
                                                                                         jump=NONE})
              | munchStm(R.MOVE(R.TEMP t, R.BINOP(R.PLUS, e2, R.CONST i))) = emit(A.OPER{assem="addi `d0, `s0, "^int i^"\n",
                                                                                         src=[munchExp e2],
                                                                                         dst=[t],
                                                                                         jump=NONE})

              | munchStm(R.MOVE(R.TEMP t1, R.TEMP t2)) = emit(A.MOVE{assem="mr `d0, `s0\n",
                                                                     src=(t2),
                                                                     dst=(t1)})(*load register lr rt, ra = ori rt, ra, 0*)
              | munchStm(R.MOVE(R.TEMP t, e2)) = emit(A.MOVE{assem="mr `d0, `s0\n",
                                                                 src=(munchExp e2),
                                                                 dst=(t)})
              | munchStm(R.EXP e1) = ignore (munchExp e1)
              | munchStm _ = E.impossible "CODEGEN STM MISMATCH"

    (*------munchExp------*)
            and munchExp(R.ESEQ(s,e)) = (munchStm s; munchExp e)
              | munchExp(R.MEM(R.BINOP(R.PLUS,e1,R.CONST i)))=result(fn r=>emit(A.OPER{assem="ld `d0, " ^ int i ^ "(`s0)\n",
                                                                                       src=[munchExp e1],
                                                                                       dst=[r],
                                                                                       jump=NONE}))

              | munchExp(R.MEM(R.BINOP(R.PLUS, R.CONST i, e1))) = result(fn r => emit(A.OPER{assem="ld `d0, " ^ int i ^ "(`
                  s0)\n",
                                                                                             src=[munchExp e1],
                                                                                             dst=[r],
                                                                                   jump=NONE}))
    | munchExp(R.MEM(R.CONST i)) = let val r1 = result(fn r1 => emit(A.OPER{assem="li `d0, "^int i^"\n",
                                                                            src=[],
                                                                            dst=[r1],
                                                                            jump=NONE}))
                                   in result(fn r => emit(A.OPER{assem="ld `d0, 0(`s0)\n",
                                                                 src=[r1],
                                                                 dst=[r],
                                                                 jump=NONE}))
                                   end
    | munchExp(R.MEM e) = result(fn r => emit(A.OPER{assem="ld `d0, 0(`s0)\n",
                                                     src=[munchExp e],
                                                     dst=[r],
                                                     jump=NONE}))
    (*BINOP(o, e1, e2)*)
    | munchExp(R.BINOP(R.PLUS, R.CONST i, e1)) = result(fn r => emit(A.OPER{assem="addi `d0, `s0, " ^ int i ^ "\n"
        ,
                                                                            src=[munchExp e1],
                                                                            dst=[r],
                                                                            jump=NONE}))

    | munchExp(R.BINOP(R.PLUS, e1, R.CONST i)) = result(fn r => emit(A.OPER{assem="addi `d0, `s0, " ^ int i ^ "\n"
        ,
                                                                            src=[munchExp e1],
                                                                            dst=[r],
                                                                            jump=NONE}))

    | munchExp(R.BINOP(R.PLUS, e1, e2)) = result(fn r => emit(A.OPER{assem="add `d0, `s0, `s1\n",
                                                                     src=[munchExp e1, munchExp e2],
                                                                     dst=[r],
                                                                     jump=NONE}))

    | munchExp(R.BINOP(R.MINUS, e1, R.CONST i)) = result(fn r => emit(A.OPER{assem="addi `d0, `s0, -" ^ int i ^ "\
        n",
                                                                             src=[munchExp e1],
                                                                             dst=[r],
                                                                             jump=NONE}))

    | munchExp(R.BINOP(R.MINUS, e1, e2)) = result(fn r => emit(A.OPER{assem="subf `d0, `s1, `s0\n",
                                                                      src=[munchExp e1, munchExp e2],
                                                                      dst=[r],
                                                                      jump=NONE}))
    | munchExp(R.BINOP(R.MUL, e1, R.CONST 2)) = result(fn r => emit(A.OPER{assem="rldic `d0, `s0, "^int 1^", 0\n",
                                                                       src=[munchExp e1],
                                                                       dst=[r],
                                                                       jump=NONE}))
| munchExp(R.BINOP(R.MUL, R.CONST 2, e1)) = result(fn r => emit(A.OPER{assem="rldic `d0, `s0, "^int 1^", 0\n",
                                                                       src=[munchExp e1],
                                                                       dst=[r],
                                                                       jump=NONE}))
| munchExp(R.BINOP(R.MUL, e1, R.CONST 0)) =  result(fn r => emit(A.OPER{assem="li `d0, "^int 0^"\n",
                                                                        src=[],
                                                                        dst=[r],
                                                                        jump=NONE}))
| munchExp(R.BINOP(R.MUL, R.CONST 0, e1)) =  result(fn r => emit(A.OPER{assem="li `d0, "^int 0^"\n",
                                                                        src=[],
                                                                        dst=[r],
                                                                        jump=NONE}))
| munchExp(R.BINOP(R.MUL, e1, R.CONST i)) = result(fn r => emit(A.OPER{assem="muli `d0, `s0, " ^ int i ^ "\n",
                                                                       src=[munchExp e1],
                                                                       dst=[r],
                                                                       jump=NONE}))

| munchExp(R.BINOP(R.MUL, R.CONST i, e2)) = result(fn r => emit(A.OPER{assem="muli `d0, `s0, " ^ int i ^ "\n",
                                                                       src=[munchExp e2],
                                                                       dst=[r],
                                                                       jump=NONE}))
| munchExp(R.BINOP(R.MUL, e1, e2)) = result(fn r => emit(A.OPER{assem="mullw `d0, `s0, `s1\n",
                                                                src=[munchExp e1, munchExp e2],
                                                                dst=[r],
                                                                jump=NONE}))
| munchExp(R.BINOP(R.DIV, e1, R.CONST 0)) = result(fn r =>emit(A.OPER{assem="li `d0,"^int 0^"\n",
                                                                    src=[],
                                                                    dst=[r],
                                                                    jump=NONE}))
| munchExp(R.BINOP(R.DIV, e1, R.CONST 2)) = result(fn r => emit(A.OPER{assem="sradi `d0, `s0, "^int 1^"\n",
                                                                       src=[munchExp e1],
                                                                       dst=[r],
                                                                       jump=NONE}))
| munchExp(R.BINOP(R.AND, e1, R.CONST i)) =  result(fn r => emit(A.OPER{assem="andi `d0, `s0, " ^ int i ^ "\n"
    ,
                                                                        src=[munchExp e1],
                                                                        dst=[r],
                                                                        jump=NONE}))

| munchExp(R.BINOP(R.AND, R.CONST i, e1)) = result(fn r => emit(A.OPER{assem="andi `d0, `s0, " ^ int i ^ "\n",
                                                            src=[munchExp e1],
                                                            dst=[r],
                                                            jump=NONE}))

| munchExp(R.BINOP(R.AND, e1, e2)) = result(fn r => emit(A.OPER{assem="and `d0, `s0, `s1\n",
                                                      src=[munchExp e1, munchExp e2],
                                                      dst=[r],
                                                      jump=NONE}))

| munchExp(R.BINOP(R.OR, e1, R.CONST i)) = result(fn r => emit(A.OPER{assem="ori `d0, `s0," ^ int i ^ "\n",
                                                           src=[munchExp e1],
                                                           dst=[r],
                                                           jump=NONE}))

| munchExp(R.BINOP(R.OR, R.CONST i, e1)) = result(fn r => emit(A.OPER{assem="ori `d0, `s0, " ^ int i ^ "\n",
                                                           src=[munchExp e1],
                                                           dst=[r],
                                                           jump=NONE}))

| munchExp(R.BINOP(R.OR, e1, e2)) = result(fn r => emit(A.OPER{assem="or `d0, `s0, `s1\n",
                                                     src=[munchExp e1, munchExp e2],
                                                     dst=[r],
                                                     jump=NONE}))

| munchExp(R.BINOP(R.LSHIFT, e1, R.CONST i)) = result(fn r => emit(A.OPER{assem="rldic `d0, `s0, " ^ int i ^ "
   \n",
                                                              src=[munchExp e1],
                                                              dst=[r],
                                                              jump=NONE}))

| munchExp(R.BINOP(R.LSHIFT, e1, e2)) = result(fn r => emit(A.OPER{assem="sld `d0, `s0, `s1\n",
                                                        src=[munchExp e1, munchExp e2],
                                                        dst=[r],
                                                        jump=NONE}))
| munchExp(R.BINOP(R.RSHIFT, e1, R.CONST 0)) = result(fn r => emit(A.OPER{assem="ld `d0, 0(`s0)\n",
                                                              src=[munchExp e1],
                                                              dst=[r],
                                                              jump=NONE}))

| munchExp(R.BINOP(R.RSHIFT, e1, R.CONST i)) = result(fn r => emit(A.OPER{assem="sradi `d0, `s0, "^int i^"\n",
                                                           src=[munchExp e1],
                                                           dst=[r],
                                                           jump=NONE}))
```

```
  | munchExp(R.BINOP(R.RSHIFT, e1, e2)) = result(fn r => emit(A.OPER{assem="srd `d0, `s0, `s1\n",
                                                                    src=[munchExp e1, munchExp e2],
                                                                    dst=[r],
                                                                    jump=NONE}))
  | munchExp(R.BINOP(R.XOR, e1, R.CONST i)) = result(fn r => emit(A.OPER{assem="xori `d0, `s0, "^int i^"\n",
                                                                        src=[munchExp e1],
                                                                        dst=[r],
                                                                        jump=NONE}))

  | munchExp(R.BINOP(R.XOR, e1, e2)) = result(fn r => emit(A.OPER{assem="xor `d0, `s0, `s1\n",
                                                                 src=[munchExp e1, munchExp e2],
                                                                 dst=[r],
                                                                 jump=NONE}))
  (*TEMP(t)*)
  | munchExp(R.TEMP t) = t
  (*NAME l*)
  | munchExp (R.NAME l) = result (fn r => emit (A.OPER { assem="lis `d0, " ^ (S.name l) ^ "@highest\nori `d0, `
    d0, " ^ (S.name l) ^ "@higher\n" ^
                                                             "rldicr `d0,`d0,32,31\noris `d0,`d0," ^ (S.name l
                                                                 ) ^ "@h\nori `d0,`d0," ^ (S.name l) ^ "@l\n",
                                                      src=[],
                                                      dst=[r],
                                                      jump=NONE }))

  (*CONST(i)*)
  | munchExp(R.CONST 0) = result(fn r => emit(A.OPER{assem="li `d0, " ^ int 0 ^ "\n",
                                                  src=[],
                                                  dst=[r],
                                                  jump=NONE}))
  | munchExp(R.CONST i) = result(fn r => emit(A.OPER{assem="li `d0, " ^ int i ^ "\n",
                                                  src=[],
                                                  dst=[r],
                                                  jump=NONE}))
  (*IMPORTANT brsl $lr, S.name f*)
  | munchExp (R.CALL (R.NAME f, es)) = result (fn r => ( if name = F.mainLabel then ()
                                                      else ListPair.appEq (fn (r, m) => munchStm (R.MOVE (m,
                                                          R.TEMP r)))
                                                                       (F.callersaves Absyn.PPU_Func,
                                                                        saves)
                                                           handle ListPair.UnequalLengths => E.impossible "
                                                               CODGEN: wrong number of caller-saves vs
                                                               locations"
```

```
                                                                   ; emit (A.OPER { assem="bl "^ S.name f^ "\n",
                                                                                    src=(munchArgs es),
                                                                                    dst=r::calldefs,
                                                                                    jump=NONE })
                                                                   ; emit (A.MOVE { assem= "mr `d0, `s0\n",
                                                                                    src=F.RV,
                                                                                    dst=r })
                                                                   ; if name = F.mainLabel then ()
                                                                     else ListPair.appEq (fn (r, m) => munchStm (R.MOVE (R.
                                                                         TEMP r, m)))
                                                                                         (F.callersaves Absyn.PPU_Func,
                                                                                          saves)
                                                                          handle ListPair.UnequalLengths => E.impossible "
                                                                              CODGEN: wrong number of caller-saves vs
                                                                              locations"
                                                                   )
                                                          )
              | munchExp (R.SPUCALL (R.NAME f, es)) = result (fn r => ( if name = F.mainLabel then ()
                                                                     else ListPair.appEq (fn (r, m) => munchStm (R.MOVE (m,
                                                                         R.TEMP r)))
                                                                                         (F.callersaves Absyn.PPU_Func,
                                                                                          saves)
                                                                          handle ListPair.UnequalLengths => E.impossible "
                                                                              CODGEN: wrong number of caller-saves vs
                                                                              locations"
                                                                   ; emit (A.OPER { assem="# call spu function\nbl spucall_"
                                                                       ^ S.name f^ "\n",
                                                                                    src=(munchArgs es),
                                                                                    dst=r::calldefs,
                                                                                    jump=NONE })
                                                                   ; emit (A.MOVE { assem= "mr `d0, `s0\n",
                                                                                    src=F.RV,
                                                                                    dst=r })
                                                                   ; if name = F.mainLabel then ()
                                                                     else ListPair.appEq (fn (r, m) => munchStm (R.MOVE (R.
                                                                         TEMP r, m)))
                                                                                         (F.callersaves Absyn.PPU_Func,
                                                                                          saves)
                                                                          handle ListPair.UnequalLengths => E.impossible "
                                                                              CODGEN: wrong number of caller-saves vs
                                                                              locations"
                                                                   )
                                                          )
```

```
      | munchExp (R.PARCALL (R.NAME f, es)) = result (fn r => ( if name = F.mainLabel then ()
                                                             else ListPair.appEq (fn (r, m) => munchStm (R.MOVE (m,
                                                                 R.TEMP r)))
                                                                                 (F.callersaves Absyn.PPU_Func,
                                                                                  saves)
                                                                  handle ListPair.UnequalLengths => E.impossible "
                                                                      CODGEN: wrong number of caller-saves vs
                                                                      locations"
                                                           ; emit (A.OPER { assem="# par-call spu function\nbl
                                                               parcall_"^ S.name f^ "\n",
                                                                            src=(munchArgs es),
                                                                            dst=r::calldefs,
                                                                            jump=NONE })
                                                           ; emit (A.MOVE { assem= "mr `d0, `s0\n",
                                                                            src=F.RV,
                                                                            dst=r })
                                                           ; if name = F.mainLabel then ()
                                                             else ListPair.appEq (fn (r, m) => munchStm (R.MOVE (R.
                                                                 TEMP r, m)))
                                                                                 (F.callersaves Absyn.PPU_Func,
                                                                                  saves)
                                                                  handle ListPair.UnequalLengths => E.impossible "
                                                                      CODGEN: wrong number of caller-saves vs
                                                                      locations"
                                                           )
                                                  )
      | munchExp e = ( PrintTree.printTree (TextIO.stdErr, R.EXP e)
                     ; ErrorMsg.impossible "CODEGEN MUNCH EXP NO MATCH"
                     )



    (*------munchArgs(i, es)------*)
        and munchArgs es = ListPair.map (fn (t, e) => let val src = munchExp e
                                                      in ( emit (A.MOVE { assem="mr `d0, `s0\n",
                                                                          src=src,
                                                                          dst=t })
                                                         ; src)
                                                      end)
                                        (F.args Absyn.PPU_Func, es)
            handle ListPair.UnequalLengths => E.impossible "CALL: MORE ARGS THAN REGISTERS"

in ( munchStm stm
          handle _ => ErrorMsg.impossible "munchStm crashed"
        ; rev (!ilist)
        )
    end
end
```

B.2.2 SPU Code Generator

Listing B.4: spuGen.sml

```
structure spuGen =
struct

structure Frame = CBFrame

structure R = Tree
structure A = Assem
structure E = ErrorMsg
structure F = CBFrame
structure S = Symbol

fun codegen (frame as {name, fname, formals, size, moves, saves, ftag}) stm =

    let val ilist = ref []
        fun int z = if (z<0)
                    then "-" ^ (Int.toString (0 - z))
                    else Int.toString z
        fun emit x = ilist := x:: !ilist
        fun result gen = let val t = Temp.newtemp()
                         in ( gen t
                            ; t)
                         end
        val calldefs = F.calldefs Absyn.SPU_Func

```

```
(*------munchStm------*)

(*SEQ(s1, s2)*)
    fun munchStm(R.SEQ(s1,s2)) = (munchStm s1; munchStm s2)
     (*LABEL l*)
     | munchStm(R.LABEL l) = emit(A.LABEL{assem=S.name l ^ ":\n",
                       lab=l})

(*JUMP(e, labs)*)
     | munchStm(R.JUMP(e, ls)) = emit(A.OPER{assem="bi `s0\n",
                            src=[munchExp e],
                            dst=[],
                            jump=SOME ls})


(*CJump(o, e1, e2, t, f)*)
     | munchStm(R.CJUMP(R.NE, e1, R.CONST 0, t, f)) = emit(A.OPER{assem="brnz `s0, "^S.name t^"\n",
                                           src=[munchExp e1],
                                           dst=[],
                                           jump=SOME [t,f]})

     | munchStm(R.CJUMP(R.NE, e1, R.CONST i, t, f)) = let val r = result(fn r => emit(A.OPER{assem="ceqi `d0, `s0,
       "^int i^"\n",
                                                    src=[munchExp e1
                                                      ],
                                                    dst=[r],
                                                    jump=NONE}))
                                     in
                                      emit(A.OPER{assem="brnz `s0, "^ S.name f^"\
                                        n",
                                            src=[r],
                                            dst=[],
                                            jump=SOME [t,f]})
                                     end

     | munchStm(R.CJUMP(R.NE, e1, e2, t, f)) = let val r = result(fn r => emit(A.OPER{assem="ceq `d0, `s0, `s1 \n",
                                                src=[munchExp e1,
                                                  munchExp e2],
                                                dst=[r],
                                                jump=NONE}
                                            )
                                     )
                                 in
```

```
                                                           emit(A.OPER{assem="brnz `s0, "^ S.name f^"\n",
                                                                       src=[r],
                                                                       dst=[],
                                                                       jump=SOME [t,f]}
                                                               )
                                                          end
| munchStm(R.CJUMP(R.EQ, e1, R.CONST 0, t, f)) = emit(A.OPER{assem="brz `s0, "^S.name t^"\n",
                                                             src=[munchExp e1],
                                                             dst=[],
                                                             jump=SOME [t, f]})
| munchStm(R.CJUMP(R.EQ, e1, R.CONST i, t, f)) = let val r = result(fn r => emit(A.OPER{assem="ceqi `d0, `s0,
    "^int i^ "\n",
                                                                                              src=[munchExp e1
                                                                                                  ],
                                                                                              dst=[r],
                                                                                              jump=NONE}))
                                                 in
                                                      emit(A.OPER{assem="brnz `s0, "^ S.name t^"\n",
                                                                  src=[r],
                                                                  dst=[],
                                                                  jump=SOME [t,f]})
                                                 end

| munchStm(R.CJUMP(R.EQ, e1, e2, t, f)) =let val r =  result(fn r => emit(A.OPER{assem="ceq `d0, `s0, `s1 \n",
                                                                                      src=[munchExp e1,
                                                                                          munchExp e2],
                                                                                      dst=[r],
                                                                                      jump=NONE}))
                                         in
                                             emit(A.OPER{assem="brnz `s0, "^ S.name t^"\n",
                                                         src=[r],
                                                         dst=[],
                                                         jump=SOME [t,f]})
                                         end

| munchStm(R.CJUMP(R.GT, e1, R.CONST i, t, f)) = let val r = result(fn r => emit(A.OPER{assem="cgti `d0, `s0,
    "^int i^ "\n",
                                                                                        src=[munchExp e1],
                                                                                        dst=[r],
                                                                                        jump=NONE}))
                                                 in
                                                     emit(A.OPER{assem="brnz `s0, "^ S.name t^"\n",
                                                                 src=[r],
                                                                 dst=[],
                                                                 jump=SOME [t,f]})
                                                 end

| munchStm(R.CJUMP(R.GT,e1,e2,t, f)) = let val r = result(fn r => emit(A.OPER{assem="cgt `d0, `s0, `s1\n",
                                                                              src=[munchExp e1, munchExp e2],
                                                                              dst=[r],
                                                                              jump=NONE}))
                                       in
                                           emit(A.OPER{assem="brnz `s0, "^S.name t^"\n",
                                                       src=[r],
                                                       dst=[],
                                                       jump=SOME [t,f]})
                                       end

| munchStm(R.CJUMP(R.LT, e1, e2, t, f)) =  let val r = result(fn r => emit(A.OPER{assem="cgt `d0, `s1, `s0\n",
                                                                                  src=[munchExp e1, munchExp
                                                                                      e2],
                                                                                  dst=[r],
                                                                                  jump=NONE}))
                                           in
                                               emit(A.OPER{assem="brnz `s0, "^S.name t^"\n",
                                                           src=[r],
                                                           dst=[],
                                                           jump=SOME [t,f]})
                                           end
| munchStm(R.CJUMP(R.LE, e1, e2, t, f)) = let val r1 = result(fn r => emit(A.OPER{assem="cgt `d0, `s1, `s0\n",
                                                                              src=[munchExp e1, munchExp e2],
                                                                              dst=[r],
                                                                              jump=NONE}))
                                              val r2 = result(fn r=>emit(A.OPER{assem="ceq `d0, `s0, `s1 \n",
                                                                                src=[munchExp e1, munchExp e2
                                                                                    ],
                                                                                dst=[r],
                                                                                jump=NONE}))
                                          in let val r = result(fn r => emit(A.OPER{assem="or `d0, `s0, `s1 \n
                                              ",
                                                                                  src=[r1,r2],
                                                                                  dst=[r],
                                                                                  jump=NONE}))
                                             in
                                                 emit(A.OPER{assem="brnz `s0, "^S.name t^"\n",
                                                             src=[r],
                                                                       dst=[],
                                                                       jump=SOME [t,f]})
                                                       end
                                                    end



          (*MOVE(MEM(e1), e2)*)

          (*Allocating Callee's Frame...*)
          | munchStm(R.MOVE(R.MEM(R.OFFSET(i,s)),e)) = emit(A.ARG{ assem = "stqd `a, -" ^ "`o($sp)\n",
                                                                   arg = munchExp e,
                                                                   off = i,
                                                                   fsize = s})

          | munchStm(R.MOVE(R.MEM(R.BINOP(R.PLUS,e1,R.CONST i)),e2))=emit(A.OPER{assem="stqd `s1, " ^ int i ^"(`s0)\n",
                                                                                 src=[munchExp e1, munchExp e2],
                                                                                 dst=[],
                                                                                 jump=NONE})
          | munchStm(R.MOVE(R.MEM(R.BINOP(R.PLUS, R.CONST i, e1)), e2)) = emit(A.OPER{assem="stqd `s1, "^ int i ^"(`s0)\
            n",
                                                                                      src=[munchExp e1, munchExp e2],
                                                                                      dst=[],
                                                                                      jump=NONE})
          | munchStm(R.MOVE(R.MEM(R.BINOP(R.PLUS, e1, e2)), e3)) = emit(A.OPER{assem="stqx `s2, `s0, `s1\n",
                                                                               src=[munchExp e1, munchExp e2, munchExp
                                                                                   e3],
                                                                               dst=[],
                                                                               jump=NONE})
          | munchStm(R.MOVE(R.MEM(R.CONST i), e2)) = emit(A.OPER{assem="stqa `s0, "^ int i^"\n",
                                                                 src=[munchExp e2],
                                                                 dst=[],
                                                                 jump=NONE})
          | munchStm(R.MOVE(R.MEM (e1), e2)) = emit(A.OPER{assem="stqd `s1, 0(`s0)\n",
                                                           src=[munchExp e1, munchExp e2],
                                                           dst=[],
                                                           jump=NONE})
(*MOVE(TEMP t, e)*)
          | munchStm(R.MOVE(R.TEMP t,R.MEM(R.BINOP(R.PLUS,R.CONST i,e2))))=emit(A.OPER
                                                                                {
                                                                                   assem="lqd `d0, "^ int i^"(`s0) \n",
                                                                                   src=[munchExp e2],
                                                                                   dst=[t],
                                                                                   jump=NONE
                                                                                }
                                                                               )
          | munchStm(R.MOVE(R.TEMP t,R.MEM(R.BINOP(R.PLUS,e2,R.CONST i))))=emit(A.OPER{assem="lqd `d0, "^ int i^"(`s0) \
              n",
                                                                                       src=[munchExp e2],
                                                                                       dst=[t],
                                                                                       jump=NONE})
          | munchStm(R.MOVE(R.TEMP t, R.MEM(R.CONST i))) = emit(A.OPER{assem="lqa `d0, " ^int i^ "\n",
                                                                       src=[],
                                                                       dst=[t],
                                                                       jump=NONE})
          | munchStm(R.MOVE(R.TEMP t, R.CONST i)) = emit(A.OPER{assem="il `d0," ^int i^ "\n",
                                                                src=[],
                                                                dst=[t],
                                                                jump=NONE})
          | munchStm(R.MOVE(R.TEMP t, R.MEM(e2))) = emit(A.OPER{assem="lqd `d0, "^ int 0 ^"(`s0)\n",
                                                                src=[munchExp e2],
                                                                dst=[t],
                                                                jump=NONE})
          | munchStm(R.MOVE(R.TEMP t, R.BINOP(R.PLUS, R.CONST i, e2))) = emit(A.OPER{assem="ai `d0, `s0, "^int i^"\n",
                                                                                     src=[munchExp e2],
                                                                                     dst=[t],
                                                                                     jump=NONE})
          | munchStm(R.MOVE(R.TEMP t, R.BINOP(R.PLUS, e2, R.CONST i))) = emit(A.OPER{assem="ai `d0, `s0, "^int i^"\n",
                                                                                     src=[munchExp e2],
                                                                                     dst=[t],
                                                                                     jump=NONE})

          | munchStm(R.MOVE(R.TEMP t1, R.TEMP t2)) = emit(A.MOVE{assem="lr `d0, `s0\n",
                                                                 src=(t2),
                                                                 dst=(t1)})(*load register lr rt, ra = ori rt, ra, 0*)
          | munchStm(R.MOVE(R.TEMP t, e2)) = emit(A.MOVE{assem="lr `d0, `s0\n",
                                                            src=(munchExp e2),
                                                            dst=(t)})

          | munchStm(R.EXP e1) = ignore (munchExp e1)
          | munchStm _ = E.impossible "CODEGEN STM MISMATCH"



(*------munchExp------*)

```

```
(*ESEQ(s,e)*)
        and munchExp(R.ESEQ(s,e)) = (munchStm s; munchExp e)
(*MEM(e)*)
              | munchExp(R.MEM(R.BINOP(R.PLUS, e1, R.CONST i))) = result(fn r => emit(A.OPER{assem="lqd `d0, " ^ int i ^
                   "(`s0)\n",
                                                                                             src=[munchExp e1],
                                                                                             dst=[r],
                                                                                             jump=NONE}))

              | munchExp(R.MEM(R.BINOP(R.PLUS, R.CONST i, e1))) = result(fn r => emit(A.OPER{assem="lqd `d0, " ^ int i ^
                   "(`s0)\n",
                                                                                             src=[munchExp e1],
                                                                                             dst=[r],
                                                                                             jump=NONE}))

              | munchExp(R.MEM(R.CONST i)) = result(fn r => emit(A.OPER{assem="lqa `d0, " ^ int i ^ "\n",
                                                                        src=[],
                                                                        dst=[r],
                                                                        jump=NONE}))
              | munchExp(R.MEM e) = result(fn r => emit(A.OPER{assem="lqd `d0, 0(`s0)\n",
                                                               src=[munchExp e],
                                                               dst=[r],
                                                               jump=NONE}))
(*BINOP(o, e1, e2)*)
              | munchExp(R.BINOP(R.PLUS, R.CONST i, e1)) = result(fn r => emit(A.OPER{assem="ai `d0, `s0, " ^ int i ^ "\
                  n",
                                                                                      src=[munchExp e1],
                                                                                      dst=[r],
                                                                                      jump=NONE}))

              | munchExp(R.BINOP(R.PLUS, e1, R.CONST i)) = result(fn r => emit(A.OPER{assem="ai `d0, `s0, " ^ int i ^ "\
                  n",
                                                                                      src=[munchExp e1],
                                                                                      dst=[r],
                                                                                      jump=NONE}))

              | munchExp(R.BINOP(R.PLUS, e1, e2)) = result(fn r => emit(A.OPER{assem="a `d0, `s0, `s1\n",
                                                                               src=[munchExp e1, munchExp e2],
                                                                               dst=[r],
                                                                               jump=NONE}))

              | munchExp(R.BINOP(R.MINUS, e1, R.CONST i)) = result(fn r => emit(A.OPER{assem="ai `d0, `s0, -" ^ int i ^
                  "\n",
                                                                             src=[munchExp e1],
                                                                             dst=[r],
                                                                             jump=NONE}))

    | munchExp(R.BINOP(R.MINUS, e1, e2)) = result(fn r => emit(A.OPER{assem="sf `d0, `s1, `s0\n",
                                                                      src=[munchExp e1, munchExp e2],
                                                                      dst=[r],
                                                                      jump=NONE}))
    | munchExp(R.BINOP(R.MUL, e1, R.CONST 2)) = result(fn r => emit(A.OPER{assem="shli `d0, `s0, "^int 1^"\n",
                                                                           src=[munchExp e1],
                                                                           dst=[r],
                                                                           jump=NONE}))
    | munchExp(R.BINOP(R.MUL, R.CONST 2, e1)) = result(fn r => emit(A.OPER{assem="shli `d0, `s0, "^int 1^"\n",
                                                                           src=[munchExp e1],
                                                                           dst=[r],
                                                                           jump=NONE}))
    | munchExp(R.BINOP(R.MUL, e1, R.CONST 0)) =  result(fn r => emit(A.OPER{assem="il `d0, "^int 0^"\n",
                                                                            src=[],
                                                                            dst=[r],
                                                                            jump=NONE}))
    | munchExp(R.BINOP(R.MUL, R.CONST 0, e1)) =  result(fn r => emit(A.OPER{assem="il `d0, "^int 0^"\n",
                                                                            src=[],
                                                                            dst=[r],
                                                                            jump=NONE}))
    | munchExp(R.BINOP(R.MUL, e1, R.CONST i)) = result(fn r => emit(A.OPER{assem="mpyi `d0, `s0, " ^ int i ^ "
        \n",
                                                                           src=[munchExp e1],
                                                                           dst=[r],
                                                                           jump=NONE}))

    | munchExp(R.BINOP(R.MUL, R.CONST i, e2)) = result(fn r => emit(A.OPER{assem="mpyi `d0, `s0, " ^ int i ^ "
        \n",
                                                                           src=[munchExp e2],
                                                                           dst=[r],
                                                                           jump=NONE}))


    | munchExp(R.BINOP(R.MUL, e1, e2)) = result(fn r => emit(A.OPER{assem="mpy `d0, `s0, `s1\n",
                                                                    src=[munchExp e1, munchExp e2],
                                                                    dst=[r],
                                                                    jump=NONE}))
    | munchExp(R.BINOP(R.DIV, e1, R.CONST 0)) =result(fn r =>emit(A.OPER{assem="il `d0,"^int 0^"\n",
                                                                         src=[],
                                                                     dst=[r],
                                                                     jump=NONE}))
| munchExp(R.BINOP(R.DIV, e1, R.CONST 2)) = let val r1 = result(fn r1 => emit(A.OPER{assem="rotmi `d0, `s0
    , "^int 1^"\n",
                                                                                     src=[munchExp e1],
                                                                                     dst=[r1],
                                                                                     jump=NONE}))
                                            in
                                                let val r2 = result(fn r2 => emit(A.OPER{assem="shli `d0,
                                                    `s0, "^int 1^"\n",
                                                                                         src=[r1],
                                                                                         dst=[r2],
                                                                                         jump=NONE}));
                                                in
                                                    result(fn r => emit(A.OPER{assem="rotmi `d0, `s0, "^
                                                        int 1^"\n",
                                                                               src=[r2],
                                                                               dst=[r],
                                                                               jump=NONE}))
                                                end

                                            end


| munchExp(R.BINOP(R.AND, e1, R.CONST i)) =  result(fn r => emit(A.OPER{assem="andi `d0, `s0, " ^ int i ^
    "\n",
                                                                        src=[munchExp e1],
                                                                        dst=[r],
                                                                        jump=NONE}))

| munchExp(R.BINOP(R.AND, R.CONST i, e1)) = result(fn r => emit(A.OPER{assem="andi `d0, `s0, " ^ int i ^ "
    \n",
                                                                       src=[munchExp e1],
                                                                       dst=[r],
                                                                       jump=NONE}))

| munchExp(R.BINOP(R.AND, e1, e2)) = result(fn r => emit(A.OPER{assem="and `d0, `s0, `s1\n",
                                                                src=[munchExp e1, munchExp e2],
                                                                dst=[r],
                                                                jump=NONE}))

| munchExp(R.BINOP(R.OR, e1, R.CONST i)) = result(fn r => emit(A.OPER{assem="ori `d0, `s0," ^ int i ^ "\n"
    ,
                                   src=[munchExp e1],
                                   dst=[r],
                                   jump=NONE}))

| munchExp(R.BINOP(R.OR, R.CONST i, e1)) = result(fn r => emit(A.OPER{assem="ori `d0, `s0, " ^ int i ^ "\n
  ",
                                   src=[munchExp e1],
                                   dst=[r],
                                   jump=NONE}))

| munchExp(R.BINOP(R.OR, e1, e2)) = result(fn r => emit(A.OPER{assem="or `d0, `s0, `s1\n",
                               src=[munchExp e1, munchExp e2],
                               dst=[r],
                               jump=NONE}))

| munchExp(R.BINOP(R.LSHIFT, e1, R.CONST i)) = result(fn r => emit(A.OPER{assem="shli `d0, `s0, " ^ int i
  ^ "\n",
                                     src=[munchExp e1],
                                     dst=[r],
                                     jump=NONE}))

| munchExp(R.BINOP(R.LSHIFT, e1, e2)) = result(fn r => emit(A.OPER{assem="shl `d0, `s0, `s1\n",
                                 src=[munchExp e1, munchExp e2],
                                 dst=[r],
                                 jump=NONE}))
| munchExp(R.BINOP(R.RSHIFT, e1, R.CONST 0)) = result(fn r => emit(A.OPER{assem="lqd `d0, 0(`s0)\n",
                                     src=[munchExp e1],
                                     dst=[r],
                                     jump=NONE}))

| munchExp(R.BINOP(R.RSHIFT, e1, R.CONST i)) =let val r1 = result(fn r1 => emit(A.OPER{assem="rotmi `d0, `
  s0, "^int i^"\n",
                                      src=[munchExp e1],
                                      dst=[r1],
                                      jump=NONE}))
                       in
                         let val r2 = result(fn r2 => emit(A.OPER{assem="shli `d0
                           , `s0, "^int i^"\n",
                                             src=[r1],
                                             dst=[r2],
                                             jump=NONE}))
                         in
```

```
                                                              result(fn r => emit(A.OPER{assem="rotmi `d0, `s0, "^int i^
                                                                  "\n",
                                                                                         src=[r2],
                                                                                         dst=[r],
                                                                                         jump=NONE}))
                                                                end
                                                            end


              | munchExp(R.BINOP(R.RSHIFT, e1, e2)) =let val r1=result(fn r1=>emit(A.OPER{assem="rotm `d0, `s0, `s1\n",
                                                                                          src=[munchExp e1, munchExp e2
                                                                                              ],
                                                                                          dst=[r1],
                                                                                          jump=NONE}))
                                                     in
                                                         let val r2 = result(fn r2 => emit(A.OPER{assem="shl `d0, `s0, `
                                                             s1\n",
                                                                                                  src=[r1, munchExp e2],
                                                                                                  dst=[r2],
                                                                                                  jump=NONE}))
                                                         in
                                                             result(fn r => emit(A.OPER{assem="rotm `d0, `s0, `s1\n",
                                                                                        src=[r2, munchExp e2],
                                                                                        dst=[r],
                                                                                        jump=NONE}))
                                                         end
                                                     end

              | munchExp(R.BINOP(R.XOR, e1, R.CONST i)) = result(fn r => emit(A.OPER{assem="xori `d0, `s0, "^int i^"\n",
                                                                                 src=[munchExp e1],
                                                                                 dst=[r],
                                                                                 jump=NONE}))

              | munchExp(R.BINOP(R.XOR, e1, e2)) = result(fn r => emit(A.OPER{assem="xor `d0, `s0, `s1\n",
                                                                                 src=[munchExp e1, munchExp e2],
                                                                                 dst=[r],
                                                                                 jump=NONE}))

(*TEMP(t)*)
              | munchExp(R.TEMP t) = t
(*NAME l*)
              | munchExp (R.NAME l) = result (fn r => emit (A.OPER { assem="ila `d0, " ^ (S.name l) ^ "\n",
                                                                     src=[],
                                                                     dst=[r],
                                                                     jump=NONE }))


(* CONST(i) *)
              | munchExp(R.CONST 0) = result(fn r => emit(A.OPER{assem="il `d0, " ^ int 0 ^ "\n",
                                                                 src=[],
                                                                 dst=[r],
                                                                 jump=NONE}))
              | munchExp(R.CONST i) = result(fn r => emit(A.OPER{assem="il `d0, " ^ int i ^ "\n",
                                                                 src=[],
                                                                 dst=[r],
                                                                 jump=NONE}))
(* brsl $lr, S.name f *)
              | munchExp (R.CALL (R.NAME f, es)) = result (fn r => ( if name = F.mainLabel then ()
                                                              else ListPair.appEq (fn (r, m) => munchStm (R.MOVE (m, R.
                                                                  TEMP r)))
                                                                                (F.callersaves Absyn.SPU_Func,
                                                                                 saves)
                                                                   handle ListPair.UnequalLengths => E.impossible "
                                                                       CODGEN: wrong number of caller-saves vs locations
                                                                       "
                                                            ; emit (A.OPER { assem="brsl $lr,"^ S.name f^ "\n",
                                                                             src=(munchArgs es),
                                                                             dst=r::calldefs,
                                                                             jump=NONE })
                                                            ; emit (A.MOVE { assem= "lr `d0, `s0\n",
                                                                             src=F.RV,
                                                                             dst=r })
                                                            ; if name = F.mainLabel then ()
                                                              else ListPair.appEq (fn (r, m) => munchStm (R.MOVE (R.TEMP
                                                                   r, m)))
                                                                                (F.callersaves Absyn.SPU_Func,
                                                                                 saves)
                                                                   handle ListPair.UnequalLengths => E.impossible "
                                                                       CODGEN: wrong number of caller-saves vs locations
                                                                       "
                                                            )
                                                   )


              | munchExp e = ( PrintTree.printTree (TextIO.stdErr, R.EXP e)
                              ; ErrorMsg.impossible "CODEGEN MUNCH EXP NO MATCH"
                              )



(*------munchArgs(i, es)------*)
            and munchArgs es = ListPair.map (fn (t, e) => let val src = munchExp e
                                                           in ( emit (A.MOVE { assem="lr `d0, `s0\n",
                                                                               src=src,
                                                                               dst=t })
                                                              ; src)
                                                           end)
                                             (F.args Absyn.SPU_Func, es)
                                handle ListPair.UnequalLengths => E.impossible "CALL: MORE ARGS THAN REGISTERS"

    in ( munchStm stm
         handle _ => ErrorMsg.impossible "munchStm crashed"
       ; rev (!ilist)
       )
    end
end
```

C. RCS Generator and Automated Linker

This is the script for RCS generation and automated linking. Optimizations are also implemented in this part.

Listing C.1: tigc

```python
#!/usr/bin/python

import sys
import os
import stat

from subprocess import Popen, PIPE

SRC_DIR = "../test"
XLC_DIR = os.path.join(SRC_DIR, "xlc")
SML_DIR = "~/sml-2"
SML_BIN = os.path.join(SML_DIR, "bin/sml")

SDK_INC = "/opt/cell/sdk/usr/spu/include"
PPU_INC = "/opt/cell/sdk/usr/include"
PPU_LIB = "/opt/cell/sdk/usr/lib"

VERBOSE = False

#
```

```python
# executes an external command and returns error code
# if useException, raise an exception instead
#
def run(cmd, useException=True):
    if VERBOSE: print "LOG: invoke command %s" % cmd
    p = Popen(cmd, shell=True, stdout=PIPE, stderr=PIPE)
    child_stdout, child_stderr = (p.stdout, p.stderr)
    retval = p.wait()

    if retval != 0:
        lines = child_stderr.readlines()
        if useException: raise Exception("Command %s error:\n%s" % (cmd, "".join(lines)))
    elif VERBOSE:
        for l in child_stderr.readlines(): print l

    return (retval, child_stdout.readlines())

#
# given tiger source file, build the parameter map in the
# following format:
#
# func_name: (type) (id, type), (id, type), ...
#
def build_param_map(srcfile):
    cmd = "%s @SMLload=sources -a %s/%s.tig" % (SML_BIN, SRC_DIR, srcfile)
    retval, output = run(cmd)

    param_map = {}
    for line in output:
        line = line.strip()
        if not line.startswith("#FUNC<SPU>"): continue
        func_dec = line[10:]

        func_name = func_dec[0:func_dec.index('(')]
        func_def = func_dec[len(func_name):]
        func_param, func_rets = func_def.split('-')
        params = func_param.split()

        param_map[func_name] = []
        param_map[func_name].append(func_rets.strip())
        for param in params:
            if param.find('(') + 1 >= param.find(')'): continue
            param = param[param.index('(') + 1 : param.index(')')]
            if len(param) == 0: continue

            fields = param.split(',')
            if len(fields) != 3: raise Exception("Invalid parameter field")
            field_id = fields[0]
            field_type = fields[2]
            param_map[func_name].append((field_id, field_type))

    return param_map

SPU_INVOKE_CODE = """
#include <stdio.h>
#include <stdlib.h>

#include <spu_intrinsics.h>
#include <spu_mfcio.h>

#define IN_BUFF_SIZE 16 * 1024
#define OUT_BUFF_SIZE 32 * 1024

typedef struct {
        uint64_t index;
        uint64_t ea_out;
        uint64_t arg_flags;
        uint64_t arg_size;

        // used for data transfer optimization
        uint64_t ea_array;
        uint64_t arr_size;
        uint64_t spuid;

        char arg_data[1];
} call_ctx;

volatile char in_data[IN_BUFF_SIZE] __attribute__ ((aligned(128)));
volatile char out_data[OUT_BUFF_SIZE] __attribute__ ((aligned(128)));

// this is the tranformed data used in parcall mode
char tran_data[IN_BUFF_SIZE];

typedef int (*target_func)(uint32_t a, uint32_t b, uint32_t c);

int invoke(uint64_t addr, uint64_t argsize, char argdata[])
```

```c
{
        uint64_t* adata = (uint64_t*)argdata;

        target_func target = (target_func)addr;
        uint32_t a = argsize >= 1 ? adata[0] : 0;
        uint32_t b = argsize >= 2 ? adata[2] : 0;
        uint32_t c = argsize >= 3 ? adata[4] : 0;

        // Only allows 3 parameters, so we can treat all spu functions
        // in this format, as the parameters are passed with registers
        // instead of in stack
        return target(a, b, c);
}

int main(int speid, uint64_t argp)
{
        uint32_t tag_id;
        call_ctx* callctx;
        int* ret_buffer;
        int retval = 0;
        int i = 0;

        struct { const char * name; uint64_t addr; } func_list[] = {
                FUNC_DICT
        };
        int func_rets[] = {
                FUNC_RETS
        };

        if ((tag_id = mfc_tag_reserve()) == MFC_TAG_INVALID)
                return 1;

        mfc_get((void *)in_data, argp, sizeof(in_data), tag_id, 0, 0);
        mfc_write_tag_mask(1 << tag_id);
        mfc_read_tag_status_all();

        callctx = (call_ctx*)in_data;

        /*
        printf("calling function %s 0x%08llx with %lld parameters[%llx]\\n",
                      func_list[callctx->index].name,
                      func_list[callctx->index].addr,
                      callctx->arg_size,
                    callctx->arg_flags);
        */

        uint64_t * argdata = (uint64_t *)callctx->arg_data;
        if (callctx->ea_array != 0) {
            // this is parcall mode, we have a fixed format of spu function:
            //    [array start size]
            //
            // array address: callctx->ea_array
            // array size   : callctx->arr_size
            // spuid        : callctx->spuid
            // spucount     : argdata[2]
            // step size    : argdata[4]

            memcpy(tran_data, in_data, 128);
            callctx = (call_ctx *)tran_data;
            argdata = (uint64_t *)callctx->arg_data;

            // Set the array address ( 7 because of the alignment )
            argdata[0] = (uint64_t)(argdata + 7);

            // Set the array length
            argdata[6] = (argdata[4] | argdata[4] << 32); argdata[7] = argdata[6];

            uint64_t groupsize = argdata[4] < 16 ? 16 : argdata[4];
            uint64_t framesize = argdata[4];

            uint64_t curr = callctx->spuid * groupsize;

            // Get the first group of data
            mfc_get((void *)in_data, callctx->ea_array + curr * sizeof(uint64_t), groupsize * sizeof(uint64_t), tag_id,
                0, 0);

            uint64_t result_curr = 1;
            while (curr < callctx->arr_size) {

                // Make sure I have the input data ready
                mfc_read_tag_status_all();

                uint64_t * input = (uint64_t *)in_data;

                uint64_t frame = 0;
                uint64_t valid_frame = groupsize / framesize;
```

```
                    if (valid_frame > (callctx->arr_size - curr) / framesize)
                        valid_frame = (callctx->arr_size - curr) / framesize;

                    // Save the argdata so in_data could be used for next iteration
                    for (i = 0; i < framesize * valid_frame; ++i) {
                        argdata[i * 2 + 8] = (input[i] | input[i] << 32);
                    }

                    // If we still have data to process
                    curr += groupsize * argdata[2];
                    if (curr < callctx->arr_size)
                        mfc_get((void *)in_data, callctx->ea_array + curr * sizeof(uint64_t), groupsize * sizeof(uint64_t),
                            tag_id, 0, 0);

                    // Process current group, which may contain several frames
                    for (frame = 0; frame < valid_frame; ++frame) {

                        for (i = 0; i < framesize; ++i) {
                            argdata[i * 2 + 8] = argdata[i * 2 + 8 + frame * framesize * 2];
                            argdata[i * 2 + 9] = argdata[i * 2 + 8 + frame * framesize * 2];
                        }

                        retval = invoke((uint64_t)func_list[callctx->index].addr, callctx->arg_size, argdata);
                        //printf("par %llu return %d\\n", callctx->spuid, retval);
                        if (func_rets[callctx->index] == 1)
                        {
                            int* ret_array = (int*)retval;
                            ret_buffer = (int*)out_data;
                            //printf("frame return %d\\n", ret_array[0]);
                            for (i = 1; i <= ret_array[0]; ++i)
                                ret_buffer[result_curr++] = ret_array[i * 4];
                            free(ret_array);
                        }
                    }
                }
                if (func_rets[callctx->index] == 1)
                {
                    ret_buffer[0] = result_curr - 1;
                    mfc_put((void *)out_data, callctx->ea_out, sizeof(out_data) / 2, tag_id, 0, 0);
                    mfc_read_tag_status_all();

                    mfc_put((void *)out_data + 16 * 1024, callctx->ea_out + 16 * 1024, sizeof(out_data) / 2, tag_id, 0, 0);
                    mfc_read_tag_status_all();
```

```
            }

            mfc_tag_release(tag_id);
            return 0;
        }

        uint64_t * arrdata = (uint64_t *)(callctx->arg_data + 48);
        for (i = 0; i < callctx->arg_size; ++i) {
            if (callctx->arg_flags & (1 << i)) {
                argdata[i * 2] = (uint64_t)(arrdata + 1);
                // printf("setting array argument %d  array size: %d pointer: %llx\\n", i, arrdata[0], argdata[i * 2]);
                arrdata += 2 * (1 + arrdata[0]);
            }
        }

        retval = invoke((uint64_t)func_list[callctx->index].addr, callctx->arg_size, callctx->arg_data);
        if (func_rets[callctx->index] == 0)
        {
            *((int *)out_data) = retval;
            // printf("return value: %lu\\n", retval);
        }
        else
        {
            int* ret_array = (int*)retval;
            int* ret_buffer = (int*)out_data;
            ret_buffer[0] = ret_array[0];
            for (i = 1; i <= ret_array[0]; ++i)
                ret_buffer[i] = ret_array[i * 4];
        }

        mfc_put((void *)out_data, callctx->ea_out, sizeof(out_data) / 2, tag_id, 0, 0);
        mfc_read_tag_status_all();

        mfc_put((void *)out_data + 16 * 1024, callctx->ea_out + 16 * 1024, sizeof(out_data) / 2, tag_id, 0, 0);
        mfc_read_tag_status_all();

        mfc_tag_release(tag_id);
        return 0;
    }

    """

    PPU_STUB_CODE = """
```

```c

#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <libspe2.h>

#include <pthread.h>

#define IN_BUFF_SIZE 16 * 1024
#define OUT_BUFF_SIZE 32 * 1024

#define SPU_COUNT 2
#define MAIN_SPU_ID 0

typedef struct {
        uint64_t index;
        uint64_t ea_out;
        uint64_t arg_flags;
        uint64_t arg_size;

        // used for data transfer optimization
        uint64_t ea_array;
        uint64_t arr_size;
        uint64_t spuid;

        char arg_data[1];
} call_ctx;

extern spe_program_handle_t spu_main;
spe_context_ptr_t spe_ctx[SPU_COUNT];

volatile char in_data[SPU_COUNT][IN_BUFF_SIZE] __attribute__ ((aligned(128)));
volatile char out_data[SPU_COUNT][OUT_BUFF_SIZE] __attribute__ ((aligned(128)));

volatile char* tran_data;

void spu_init(void)
{
    if (posix_memalign(&tran_data, 128, 32 << 20) != 0) {
        perror("failed to allocate tran_data");
        exit(1);
    }
    int spuid = 0;
```

```
        for (spuid = 0; spuid < SPU_COUNT; ++spuid) {
            if ((spe_ctx[spuid] = spe_context_create(0, NULL)) == NULL)
            {
                    perror("Failed creating context");
                    exit(1);
            }

            // printf("loading program...\\n");
            if (spe_program_load(spe_ctx[spuid], &spu_main)) {
                    perror("Failed loading program");
                    exit(1);
            }
        }
    }

    void spu_exit(void)
    {
        int spuid = 0;
        for (spuid = 0; spuid < SPU_COUNT; ++spuid) {
            if (spe_context_destroy(spe_ctx[spuid])) {
                    perror("failed to destroy context");
                    exit(1);
            }
        }
        free(tran_data);
    }

    int spu_invoke(int ctxid)
    {
        uint32_t entry = SPE_DEFAULT_ENTRY;
        spe_stop_info_t stop_info;
        if (spe_context_run(spe_ctx[ctxid], &entry, 0, (void*)(in_data[ctxid]), NULL, &stop_info) != 0) {
            perror("Failed running context");
            exit(1);
        }
        return 0;
    }

    int spu_invoke2(int ctxid)
    {
        if (spe_program_load(spe_ctx[ctxid], &spu_main)) {
                perror("Failed loading program");
                exit(1);
        }

        spu_invoke(ctxid);
    }

    void * sputhread(void * param)
    {
        spe_stop_info_t stop_info;
        int * spuid = (int *)param;
        uint32_t entry = SPE_DEFAULT_ENTRY;

        call_ctx* callctx = (call_ctx*)in_data[*spuid];
        callctx->ea_out = (uint64_t)out_data[*spuid];

        // printf("running program... %d %llx\\n", *spuid, callctx->ea_array);
        if (spe_context_run(spe_ctx[*spuid], &entry, 0, (void*)(callctx), NULL, &stop_info) < 0) {
            perror("Failed running context");
            exit(1);
        }
        // printf("[%d] return: %d\\n", *spuid, ((int *)out_data[*spuid])[0]);
        pthread_exit(NULL);
    }

    """

    def build_sources(srcfile, param_map):
        ppusrc = os.path.join(XLC_DIR, srcfile + "_ppu.s")
        spusrc = os.path.join(XLC_DIR, srcfile + "_spu.s")

        cmd = "%s @SMLload=sources -r %s/%s.tig" % (SML_BIN, SRC_DIR, srcfile)
        retval, output = run(cmd)

        namedict = {}

        ppufile = open(ppusrc, "w")
        spufile = open(spusrc, "w")

        inspu = False
        for line in output:
            if line.startswith("#SPU_BEGIN"):
                inspu = True
                continue
            if line.startswith("#SPU_END"):
                inspu = False
                continue

            if inspu and line.startswith("#FRAME <spu>"):
                fields = line.split()
                func_label = fields[2]
                func_name = fields[3].replace("\"", "")
                namedict[func_name] = func_label

            if inspu: spufile.write(line)
            else: ppufile.write(line)

        ppufile.close()
        spufile.close()

        typedict = { "int" : "int", "intArray" : "uint64_t *" }

        stubsrc = os.path.join(XLC_DIR, srcfile + "_stub.c")
        stubfile = open(stubsrc, "w")

        stubfile.write(PPU_STUB_CODE)

        func_extern = []
        func_index = 0
        for func_name, func_params in param_map.items():
            func_label = namedict[func_name]
            func_ret = func_params[0]
            func_params = func_params[1:]

            arg_assign = []
            pos = 0
            arg_flags = 0
            arg_size = len(func_params)
            arg_assign.append("            uint64_t* dest = (uint64_t *)(callctx->arg_data + 48); int i = 0;")
            arg_it = 0
            for i,t in func_params:
                if t == 'intArray':
                    arg_flags = arg_flags | (1 << arg_it)
                    arg_assign.append("            // printf(\"preparing array length: %%d\\n\", %s[0]);" % i)
                    arg_assign.append("            *(dest++) = (%s[0] | (%s[0] << 32));*dest = *(dest - 1); dest++;" % (i, i))
                    arg_assign.append("            for (i = 0; i < %s[0]; ++i) {*(dest++) = (%s[i+1] | (%s[i+1] << 32)); *dest = *(dest - 1);dest++;}" % (i, i, i))
                arg_assign.append("            *((uint64_t*)(callctx->arg_data + %d)) = 0;" % pos)
            else:
                arg_assign.append("            *((uint64_t*)(callctx->arg_data + %d)) = %s;" % (pos, i))
            pos = pos + 16
            arg_it = arg_it + 1

        func_text = '';
        if func_ret == 'intArray': func_text = 'int*'
        else: func_text = 'int'
        func_text += " spucall_%s(" % func_label
        func_text += ",".join([typedict[t] + " " + i for i,t in func_params])
        func_text += ")\n{\n"
        func_text += """
            call_ctx* callctx;
            int* res = (int*)out_data[MAIN_SPU_ID];
        \n"""
        func_text += """
            callctx = (call_ctx*)in_data[MAIN_SPU_ID];
            memset(callctx, 0, sizeof(*callctx));
            callctx->index = %d;
            callctx->ea_out = (uint64_t)out_data[MAIN_SPU_ID];
            callctx->arg_flags = %d;
            callctx->arg_size = %d;
            callctx->ea_array = 0;
            callctx->arr_size = 0;
            callctx->spuid = MAIN_SPU_ID;
            %s
            spu_invoke2(MAIN_SPU_ID);
        \n""" % (func_index, arg_flags, arg_size, "\n".join(arg_assign))
        if func_ret == 'intArray':
            func_text += """
            {
            long long* result = (long long *)malloc(sizeof(long long) * (res[0] + 1));
            for (i = 0; i <= res[0]; ++i) result[i] = (long long)res[i];
            return result;
            }
            }

            """
        else:
            func_text += "return res[0];\n}\n\n"

        if func_ret == 'intArray': func_text += 'int*'
```

```
            else: func_text += 'int'

            func_text += """ parcall_%s(uint64_t* a, int step, int len)
    {
        int start = 0;
        int spuid = 0;
        pthread_t threads[SPU_COUNT];
        int threadid[SPU_COUNT];
        call_ctx* callctx;
        uint64_t* arrdata;

        // try to use all the spu cores
        int spucount = SPU_COUNT;
        int groupsize = step;

        pthread_attr_t attr;
        pthread_attr_init(&attr);
        pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

        // check how many spu cores we really need
        if (groupsize * sizeof(uint64_t) < 128)
            groupsize = 128 / sizeof(uint64_t);
        if (spucount > (len + groupsize - 1) / groupsize)
            spucount = (len + groupsize - 1) / groupsize;

        memcpy(tran_data, a + 1, len * sizeof(uint64_t));

        for (spuid = 0; spuid < spucount; ++spuid) {
            callctx = (call_ctx*)in_data[spuid];
            memset(callctx, 0, sizeof(*callctx));
            callctx->index = %d;
            callctx->ea_out = (uint64_t)out_data[spuid];
            callctx->arg_flags = %d;
            callctx->arg_size = %d;

            // in parcall mode, let spu handle the data
            callctx->ea_array = (uint64_t)tran_data;
            callctx->arr_size = len;
            callctx->spuid = spuid;

            /*
            dest = (uint64_t*)(callctx->arg_data + 48);
            *(dest++) = ((uint64_t)step | ((uint64_t)step << 32)); *dest = *(dest - 1); dest++;
            for (i = start + spuid * step; i < start + (spuid + 1) * step; ++i) { *(dest++) = (a[i+1] | (a[i+1] << 32)); *
                dest = *(dest - 1); dest++; }
            */

            *((uint64_t*)(callctx->arg_data + 0)) = 0;
            *((uint64_t*)(callctx->arg_data + 16)) = spucount;
            *((uint64_t*)(callctx->arg_data + 32)) = step;

        }

        for (spuid = 0; spuid < spucount; ++spuid) {
            threadid[spuid] = spuid;
            pthread_create(&threads[spuid], &attr, sputhread, &threadid[spuid]);
        }
        for (spuid = 0; spuid < spucount; ++spuid) {
            void* status = NULL;
            pthread_join(threads[spuid], &status);
        }
        pthread_attr_destroy(&attr);
    """ %(func_label, func_index, arg_flags, arg_size)

            if func_ret == 'intArray':
                func_text += """
        {
            int total_len = 0;
            int i = 0;
            for (spuid = 0; spuid < spucount; ++spuid)
            {
                int* spures = (int*)out_data[spuid];
                total_len += spures[0];
            }
            long long* result = (long long *)malloc(sizeof(long long) * (total_len + 1));
            int currpos = 1;
            result[0] = total_len;
            for (spuid = 0; spuid < spucount; ++spuid)
            {
                int* spures = (int*)out_data[spuid];
                for (i = 1; i <= spures[0]; ++i) result[currpos++] = (long long)spures[i];
            }
            return result;
        }
    }
                """
```

```python

            else:
                func_text += "return 0;\n}\n";

            func_extern.append("extern int %s(%s);" % (func_label, ",".join([t for i,t in func_params])))
            func_index = func_index + 1

            stubfile.write(func_text)

        stubfile.close()

        spumainsrc = os.path.join(XLC_DIR, srcfile + "_spumain.c")
        spumainfile = open(spumainsrc, "w")

        funcdict = ",\n".join(["{\"%s\", (uint64_t)%s }" % (namedict[fname], namedict[fname]) for fname in param_map.keys()
            ])
        def rettype(typename):
            if typename == 'intArray': return '1'
            return '0'
        funcrets = ",\n".join([rettype(params[0]) for params in param_map.values()])

        spumainfile.write("#include <stdint.h>\n")
        spumainfile.write("typedef uint32_t * intArray;\n")
        spumainfile.write("\n".join(func_extern))
        spumainfile.write(SPU_INVOKE_CODE.replace("FUNC_DICT", funcdict).replace("FUNC_RETS", funcrets))

        spumainfile.close()

        spusrcs = [ spumainsrc, spusrc ]
        ppusrcs = [ stubsrc, ppusrc ]
        return spusrcs, ppusrcs

    #
    # build the target executable file with all sources file generated
    #
    def build_target(target, spusrcs, ppusrcs):
        spugcc = "/usr/bin/spu-gcc -I%s -O2 -o %s/spu_main %s ./libtiger.c" % (SDK_INC, XLC_DIR, " ".join(spusrcs))
        run(spugcc)

        embedgcc = "/usr/bin/ppu-embedspu spu_main %s/spu_main %s/spu_main-embed.o" % (XLC_DIR, XLC_DIR)
        run(embedgcc)

        argcc = "/usr/bin/ppu-ar -qcs %s/spu_main.a %s/spu_main-embed.o" % (XLC_DIR, XLC_DIR)
        run(argcc)

        ppugcc = "/usr/bin/ppu-gcc -g -I%s -L%s -R%s -mabi=altivec -maltivec -O2 %s ./libtiger.c ./libtiger2.c %s/spu_main.a
             -o %s -lspe2 -lpthread" % (
            PPU_INC, PPU_LIB, PPU_LIB,
            " ".join(ppusrcs), XLC_DIR, target)
        run(ppugcc)

    def process_options(options):
        global VERBOSE
        for option in options:
            if option == "-v": VERBOSE=True

    def main(argv):
        print "Tiger Compiler"
        if len(argv) < 2:
            raise Exception("Bad usage: tigc [src] [target]")

        print argv[0]

        process_options(argv[2:])

        param_map = build_param_map(argv[0])
        print param_map

        target = argv[1]

        spusrcs, ppusrcs = build_sources(argv[0], param_map)
        build_target(target, spusrcs, ppusrcs)

    if __name__ == '__main__':
        main(sys.argv[1:])
```

D. Testcases

D.1 Strassen Algorithm

Listing D.1: Strassen Algorithm

```
let
    type intArray = array of int
    var N := 256

    var mat1 := intArray[N * N] of 1
    var mat2 := intArray[N * N] of 2

    spu_function spu_matmul(a: intArray, s: int, l: int) : intArray =
        let
            var result := intArray[256] of 0
            var tmp := 0
        in
        (
            for r := 0 to 15 do
            (
                for c := 0 to 15 do
                (
                    tmp := 0;
                    for i := 0 to 15 do
                    (tmp := tmp + a[r * 16 + i] * a[i * 16 + c + 256]);
                result[r * 16 + c] := tmp
            )
        );
        result
    )
    end

ppu_function strassen64(a: intArray, b: intArray) : intArray =
    let
        var minput := strassen_prepare(a, b)
    in
        let
            var tr := spufor spu_matmul of (minput, 16 * 16 * 2, 16 * 16 * 98)
        in
            strassen64_sumup(tr)
        end
    end

ppu_function strassen(a: intArray, b: intArray, s: int) : intArray =
    if s = 64 then strassen64(a, b)
    else
        let
            var subsize := div(s, 2)
            var am1 := strassen_getm(a, s, 1)
            var am2 := strassen_getm(a, s, 2)
            var am3 := strassen_getm(a, s, 3)
            var am4 := strassen_getm(a, s, 4)
            var am5 := strassen_getm(a, s, 5)
            var am6 := strassen_getm(a, s, 6)
            var am7 := strassen_getm(a, s, 7)
            var bm1 := strassen_getm(b, s, 8)
            var bm2 := strassen_getm(b, s, 9)
            var bm3 := strassen_getm(b, s, 10)
            var bm4 := strassen_getm(b, s, 11)
            var bm5 := strassen_getm(b, s, 12)
            var bm6 := strassen_getm(b, s, 13)
            var bm7 := strassen_getm(b, s, 14)
        in
            let
                var m1 := strassen(am1, bm1, subsize)
                var m2 := strassen(am2, bm2, subsize)
                var m3 := strassen(am3, bm3, subsize)
```

```
                    var m4 := strassen(am4, bm4, subsize)
                    var m5 := strassen(am5, bm5, subsize)
                    var m6 := strassen(am6, bm6, subsize)
                    var m7 := strassen(am7, bm7, subsize)
                in
                    strassen_sumup(m1, m2, m3, m4, m5, m6, m7)
                end
            end

in
    for i := 0 to 199 do
        strassen(mat1, mat2, N)
end
```

D.2 n-Queens Problem

Listing D.2: 16-queen Problem

```
let
    type intArray = array of int
    var S := 15
    var tryvalue := intArray [S * 16] of 0
    var count := 0

    spu_function try(col: intArray, diag1:intArray, diag2:intArray, c:int) : int =
    let
        var count := 0
        var N := 15
    in
    (
        if c=N then count := 1
        else for r := 0 to N-1 do
            if col[r] = 0 & diag1[r+c] = 0 & diag2[r+N-1-c] = 0 then
            (
              col[r] := 1; diag1[r+c] := 1;
              diag2[r+N-1-c] := 1;
              count := count + try(col, diag1, diag2, c+1);
              col[r] := 0; diag1[r+c] := 0;
              diag2[r+N-1-c] := 0
            );
          count
        )
      end

      spu_function queen(a:intArray, s:int, l:int) : intArray =
      let
        var N := 15
        var col := intArray [N] of 0
        var diag1 := intArray [N+N-1] of 0
        var diag2 := intArray [N+N-1] of 0
        var count := intArray [1] of 0
      in
      (
        col[a[0]] := 1; diag1[a[0]] := 1; diag2[a[0] + N - 1] := 1;
        count[0] := count[0] + try(col, diag1, diag2, 1);
        count
      )
      end
    in
    (
      for i := 0 to S-1 do
      (
        tryvalue[i * 16] := i
      );
      let var result := spufor queen of (tryvalue, 16, S * 16)
      in
      (
        for i := 0 to S-1 do
        (
          count := count + result[i]
        );
        print(itoa(count, 10))
      )
      end
    )
    end
```

E. Result Data of Performance Evaluation

Size	PPU				1 SPU			
	Max	Min	Avg	Stdev	Max	Min	Avg	Stdev
64	1.58	1.556	1.569	0.012	4.99	4.89	4.95	0.052
128	11.312	11.287	11.297	0.013	34.972	34.021	34.407	0.5
256	88.503	88.337	88.424	0.083	249.98	248.97	249.423	0.512
512	634.79	632.56	633.716	1.117	1765.37	1762.25	1763.69	1.573

Table E.1: Strassen Algorithm (PPU Only vs Simple Call Mode)

SPU	No Optimization				With Optimization			
	Max	Min	Avg	Stdev	Max	Min	Avg	Stdev
2	123.92	122.03	122.84	0.972	91.355	90.013	90.751	0.68
3	92.989	91.798	92.485	0.616	71.501	70.559	70.922	0.506
4	78.337	76.943	77.692	0.702	61.837	60.344	61.208	0.773
5	66.97	65.881	66.377	0.551	58.929	57.932	58.549	0.539
6	63.104	62.475	62.65	0.396	54.915	54.403	54.724	0.28
7	63.97	62.776	63.283	0.616	54.972	54.219	54.583	0.377
8	60.103	58.659	59.328	0.727	53.966	52.679	53.243	0.658

Table E.2: Strassen Algorithm with Matrix Size 256 $\times$ 256 (Data Transfer Optimization on vs off)

SPU	64				128			
	Max	Min	Avg	Stdev	Max	Min	Avg	Stdev
2	1.673	1.604	1.6385	0.048	11.947	11.576	11.7615	0.262
3	1.37	1.05	1.21	0.226	9.022	8.766	8.894	0.181
4	1.056	1.022	1.039	0.024	7.699	7.553	7.626	0.103
5	0.989	0.932	0.9605	0.04	7.402	7.021	7.2115	0.269
6	0.903	0.872	0.8875	0.021	6.698	6.621	6.6595	0.054
7	0.901	0.866	0.8835	0.024	6.589	6.502	6.5455	0.061
8	0.872	0.843	0.8575	0.02	6.293	6.233	6.263	0.042

Table E.3: Strassen Algorithm under Parallel Call Mode

SPU	256				512			
	Max	Min	Avg	Stdev	Max	Min	Avg	Stdev
2	91.355	90.013	90.75	0.68	690.724	688.981	689.8525	1.232
3	71.501	70.559	70.922	0.506	536.971	535.683	536.327	0.9
4	61.837	60.344	61.208	0.773	458.376	456.925	457.65	1.026
5	58.929	57.932	58.549	0.539	435.759	433.773	434.766	1.404
6	54.915	54.403	54.72	0.28	442.892	440.971	441.931	1.358
7	54.972	54.219	54.58	0.377	422.329	420.799	421.564	1.081
8	53.966	52.679	53.243	0.658	409.727	407.938	408.832	1.265

Table E.4: Strassen Algorithm under Parallel Call Mode (contd.)

Chessboard Size	PPU				SPU			
	Max	Min	Avg	Stdev	Max	Min	Avg	Stdev
4	0.019	0.019	0.019	0	0.019	0.019	0.019	0
5	0.025	0.023	0.024	0.001	0.025	0.026	0.0255	0.0007
6	0.019	0.019	0.019	0	0.02	0.02	0.02	0
7	0.019	0.019	0.019	0	0.021	0.021	0.021	0
8	0.02	0.019	0.0195	0.0007	0.023	0.021	0.022	0.001
9	0.024	0.024	0.024	0	0.027	0.026	0.0265	0.0007
10	0.046	0.044	0.045	0.001	0.049	0.049	0.049	0
11	0.156	0.154	0.155	0.001	0.169	0.166	0.1675	0.002
12	0.739	0.738	0.7385	0.0007	0.836	0.832	0.834	0.002
13	4.192	4.176	4.184	0.011	4.799	4.721	4.76	0.055
14	25.903	25.434	25.668	0.331	29.387	29.224	29.3055	0.115
15	171.927	171.033	171.48	0.632	194.129	193.048	193.588	0.764
16	1201.98	1200.89	1201.44	0.774	1355.87	1355.44	1355.66	0.303

Table E.5: n-Queen Problem (PPU Only vs Simple Call Mode)

SPU Count	No Optimization				With Optimization			
	Max	Min	Avg	Stdev	Max	Min	Avg	Stdev
2	92.903	91.977	92.44	0.654	90.975	90.219	90.597	0.534
3	58.934	58.472	58.703	0.326	57.966	57.244	57.605	0.510
4	46.975	46.233	46.604	0.524	45.931	45.299	45.615	0.446
5	36.89	36.312	36.601	0.408	35.997	35.121	35.559	0.619
6	34.699	34.035	34.367	0.469	33.936	33.218	33.577	0.507
7	31.998	31.421	31.7095	0.408	31.943	31.136	31.5395	0.570
8	23.972	23.766	23.869	0.145	23.572	23.012	23.292	0.395

Table E.6: 15-Queen Problem (Data Transfer Optimization on vs off)

SPU	4-Queen				5-Queen				6-Queen			
	Max	Min	Avg	Stdev	Max	Min	Avg	Stdev	Max	Min	Avg	Stdev
2	0.014	0.014	0.014	0	0.015	0.014	0.014	0.0005	0.015	0.014	0.0146	0.0005
3	0.016	0.015	0.015	0.0005	0.016	0.014	0.015	0.0011	0.016	0.015	0.0156	0.0005
4	0.018	0.017	0.017	0.0005	0.017	0.017	0.017	0	0.017	0.017	0.017	0
5	0.018	0.017	0.017	0.0005	0.019	0.018	0.018	0.0005	0.018	0.018	0.018	0
6	0.019	0.019	0.019	0	0.02	0.019	0.019	0.0005	0.02	0.02	0.02	0
7	0.02	0.019	0.019	0.0005	0.021	0.02	0.02	0.0005	0.023	0.02	0.021	0.0015
8	0.022	0.021	0.021	0.0005	0.022	0.021	0.021	0.0005	0.023	0.021	0.022	0.001

Table E.7: n-Queen Problem under Parallel Call Mode

SPU	7-Queen				8-Queen				9-Queen			
	Max	Min	Avg	Stdev	Max	Min	Avg	Stdev	Max	Min	Avg	Stdev
2	0.015	0.014	0.014	0.0005	0.015	0.014	0.014	0.0005	0.017	0.016	0.016	0.0005
3	0.016	0.015	0.015	0.0005	0.016	0.016	0.016	0	0.029	0.017	0.021	0.0066
4	0.018	0.017	0.017	0.0005	0.017	0.017	0.017	0	0.018	0.017	0.017	0.0005
5	0.019	0.018	0.018	0.0005	0.019	0.019	0.019	0	0.019	0.019	0.019	0
6	0.021	0.02	0.02	0.0005	0.02	0.019	0.019	0.0005	0.021	0.02	0.020	0.0005
7	0.022	0.02	0.021	0.001	0.022	0.021	0.021	0.0005	0.023	0.021	0.021	0.0011
8	0.023	0.023	0.023	0	0.023	0.022	0.022	0.0005	0.024	0.023	0.023	0.0005

Table E.8: n-Queen Problem under Parallel Call Mode (contd.)

SPU	10-Queen				11-Queen				12-Queen			
	Max	Min	Avg	Stdev	Max	Min	Avg	Stdev	Max	Min	Avg	Stdev
2	0.027	0.026	0.026	0.0005	0.085	0.084	0.084	0.0005	0.373	0.373	0.373	0
3	0.024	0.024	0.024	0	0.062	0.061	0.061	0.0005	0.257	0.256	0.256	0.0005
4	0.024	0.024	0.024	0	0.053	0.051	0.052	0.001	0.201	0.199	0.2	0.001
5	0.023	0.023	0.023	0	0.051	0.051	0.051	0	0.191	0.191	0.191	0
6	0.024	0.023	0.023	0.0005	0.045	0.044	0.044	0.0005	0.143	0.142	0.142	0.0005
7	0.026	0.025	0.025	0.0005	0.045	0.045	0.045	0	0.15	0.145	0.146	0.002
8	0.027	0.025	0.026	0.001	0.045	0.045	0.045	0	0.144	0.141	0.142	0.001

Table E.9: n-Queen Problem under Parallel Call Mode (contd.)

SPU	13-Queen				14-Queen			
	Max	Min	Avg	Stdev	Max	Min	Avg	Stdev
2	2.236	2.236	2.236	0	13.004	12.999	13.0006	0.002
3	1.573	1.569	1.570	0.002	9.162	9.156	9.158	0.003
4	1.238	1.237	1.237	0.0005	7.235	7.231	7.232	0.002
5	0.976	0.973	0.974	0.001	5.623	5.622	5.622	0.0005
6	0.911	0.911	0.911	0	5.304	5.303	5.303	0.0005
7	0.69	0.678	0.682	0.006	3.845	3.843	3.844	0.001
8	0.679	0.678	0.678	0.0005	3.784	3.782	3.783	0.001

Table E.10: n-Queen Problem under Parallel Call Mode (contd.)

SPU	15-Queen				16-Queen			
	Max	Min	Avg	Stdev	Max	Min	Avg	Stdev
2	90.555	0.384	90.555	0.384	603.293	603.157	603.232	0.069
3	57.558	0.369	57.558	0.369	438.221	438.202	438.213	0.010
4	45.56	0.329	45.560	0.329	304.532	304.463	304.504	0.036
5	35.479	0.459	35.479	0.459	282.091	282.073	282.082	0.009
6	33.529	0.368	33.529	0.368	223.961	223.914	223.933	0.024
7	31.44	0.438	31.440	0.438	210.59	210.533	210.57	0.032
8	23.33	0.287	23.33	0.287	205.086	205.023	205.061	0.033

Table E.11: n-Queen Problem under Parallel Call Mode (contd.)