Diploma Thesis: Java on Cell B.E.

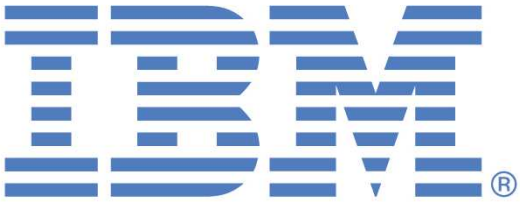
Georg Sorst

September 30, 2007

University advisor:
Prof. Dr. rer. nat. Richard Reuter

Fachhochschule Aachen, Fachbereich Elektrotechnik
Eupenerstr. 70, 52066 Aachen

Company advisor:
Markus Deuling



IBM Deutschland Entwicklung GmbH
Schoenaicher Str. 220, 71032 Boeblingen

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die im Literaturverzeichnis angegeben Quellen benutzt habe.

Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichen Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Aachen, im Monat Jahr

(Vollständige, handschriftliche Unterschrift)

Abstract

With Java becoming an increasingly important language in the HPC sector and the Cell Broadband Engine (Cell/B.E.) achieving real-life performance an order of magnitude higher than preceding computer generations coupling these two components could provide both the Java world and the Cell ecosystem with interesting opportunities. Proper use of the Cell/B.E.'s specialized Synergistic Processing Units (SPU) is the key to high performance results. Therefore a special approach to exploit their full power is necessary which must also pay attention to the limited computing environment they provide. The solution presented in this thesis consists of running a Java Virtual Machine (JVM) on the Cell/B.E.'s general purpose PowerPC Processing Unit (PPU). The JVM is extended to generate and execute native machine code for the SPUs at runtime thus requiring only little space in their limited local store memory while promising a high performance.

Contents

1 Introduction

The structure of this thesis is as follows: In section 1 the two main components making up the project, Cell/B.E. and Java will be introduced and examined. Based on some common properties they provide the motivation for this project will be explained. In section 2 a number of different concepts for the realization are discussed and a conclusion is drawn on which concept to base further work. An existing implementation of a JVM which is suitable for the concept is also introduced. In section 3 the architecture of the chosen JVM is discussed as well as the design for Java on Cell/B.E. and the resulting steps for the implementation. Section 4 then presents a number of possible implementation variations for these steps along with potential issues and their solutions. In section 5 the resulting prototype implementation is evaluated regarding capabilities and performance and some recommendations concerning the programing model are given. Section 6 concludes this thesis and in it an outlook is given for future development in this area as well as a comparison with other related projects. Finally in this section a conclusion and review of the work is performed.

As explained above in this section both Cell/B.E. and Java will be introduced including an overview over their history, the motivation behind their creation and the respective design goals. As they share a number of interesting capabilities these will be further examined and it will be explained how the motivation for this project is derived from them.

1.1 The Cell Processor

1.1.1 Motivation

With Moore's Law still holding up at the beginning of the 21st century the trend might be coming to a halt. As modern microprocessors would just be accelerated by increasing the clock frequency this would also result in an increased current leakage as well as an increase in the

generated heat. Additionally memory bandwidth was unable to catch up with the CPU frequency thus resulting in stalls of the CPU when waiting for data from the main memory. To circumvent the increase of clock frequencies as a way to increase computational speed microprocessor developers were increasingly employing multi-core designs where multiple processor cores are located on one chip. This way the chip can theoretically achieve the same computational speed at a lower frequency and thus with less heat and current leakage. However under most situations it is difficult to use several general purpose cores to their full capacity hence wasting energy and space on the chip. A better solution was seen in the coupling of a general purpose core which runs the operating system with several specialized stripped-down cores to offload computationally intensive tasks. This design has been realized in the Cell Broadband Engine Architecture (CBEA) [IBM06].

Cell/B.E. was originally thought up and jointly developed around 2001 by Sony, Toshiba and IBM (STI) as the basis for Sony's next generation of PlayStation gaming consoles, the PlayStation 3. Its planned use would outline the rough requirements for the CPU which are explained in the following.

Provide 33 times the performance of the PlayStation 2

This huge leap in performance would require a novel design approach, simply increasing the clock frequency was not expected to yield the desired performance. In the end the PlayStation 3 would achieve around 33 times the performance of the PlayStation 2 within a development time of 6 years thus far outperforming Moore's Law.

High performance in mathematical (physics, graphics) calculations

The goal was to achieve a high performance with linear algorithms at the expense of technologies such as extensive branch prediction and out-of-order execution commonly found in general purpose processors. This goal also favors a single instruction, multiple data (SIMD) approach in which one instruction works with multiple data in parallel. This technique is commonly seen in vector processors but lately also in PCs with multimedia extensions such as Intel's MMX and SSE or AMD's 3DNow!

High single-precision floating-point performance

During design it was decided that a high single-precision performance is sufficient for a gaming console. This would allow to spare some complexity that comes with a full double-precision pipeline however would also result in a reduced double-precision performance.

Stream-like high-bandwidth applications

It was expected that the applications would employ algorithms which loop over vast amounts of data coming in frequently at a high rate. This allows the architecture to emphasize bandwidth over latency which is also reflected in its name, Cell Broadband Engine. This means that while data may take some time until it is available huge amounts can be transmitted in a relatively short time thus outweighing the latency once enough data is transmitted.

1.1.2 History

In 2001 IBM got together with Sony and Toshiba to achieve their goals and design a radical new computing architecture. Sony wanted to build the CPU for their PlayStation 3, IBM had decades of experience in microprocessor design while Toshiba would act as a high-volume manufacturer of consumer devices and was also interested in producing Cell-based HDTV television sets. STI set up its headquarter, the STI Design Center (STIDC) in Austin, Texas. Just on IBM's side around 400 Mill. $ were spent involving 11 IBM locations across the entire globe and several hundred employees. The main design stage lasted from 2001 to 2005 when the first prototypes of Cell/B.E. were officially presented.

1.1.3 Implementation

The first publicly available implementation of the CBEA was featured in the PlayStation 3 and in the IBM Cell Blades which were released in 2006. Details concerning the components making up Cell/B.E. are based on this first implementation which commonly runs at 3.2 GHz and features one general purpose code and eight special purpose cores. An overview of this implementation is given in [IBM]. Tests which are documented in [Hac07] have shown that even this first generation is able to achieve near-peak performance at 99.14% with specialized workloads.

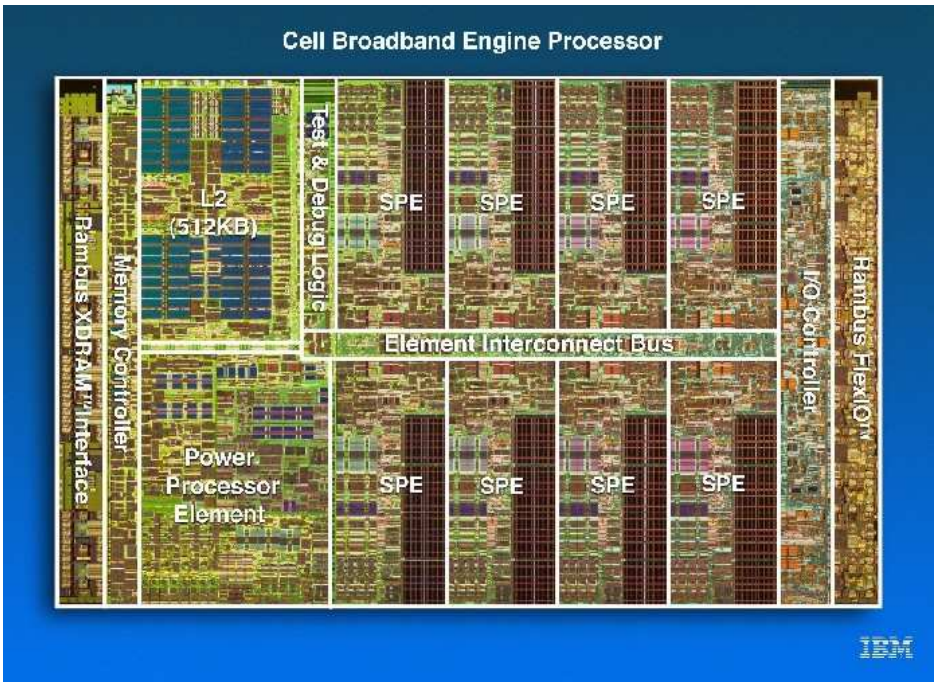


Figure 1.1: Photo of the Cell die [IBM]

A practical comparison with other architectures is given in [WSO+05]. The table comparing the raw numbers of some architectures is reproduced as Table 1.1.

	Cell		X1E	AMD64	IA64
Component	1 SPE	All 8 SPEs	(MSP)	-	-
Architecture	SIMD	Multicore SIMD	Multichip Vector	Superscalar	VLIW
Clock (GHz)	3.2	3.2	1.13	2.2	1.4
DRAM (GB/s)	25.6	25.6	34	6.4	6.4
SP Gflop/s	25.6	204.8	36	8.8	5.6
DP Gflop/s	1.83	14.63	18	4.4	5.6
Local Store	256KB	2MB	—	—	—
L2 Cache	—	512KB	2MB	1MB	256KB
L3 Cache	—	—	—	—	3MB
Power (W)	3	˜40	120	89	130
Year	2006	2006	2005	2004	2003

Table 1.1: Comparison of different processor types including Cell/B.E.

General Purpose Core

In order to maintain compatibility with legacy applications and have a good starting point an existing processor architecture was to be used as the basis for the general purpose core. IBM's

PowerPC architecture [Wik07], which has a long tradition in the field of high-performance computing and has also proven to be a very extensible and versatile architecture was destined to be this base. In the context of Cell/B.E. it is called the PowerPC Processing Unit (PPU). It is a RISC core with two hardware threads, no out-of-order execution, two pipelines and IBM's SIMD-extension VMX. It may be operated in 32-bit or 64-bit mode depending on which some semantics change. For example pointer sizes depend on the mode with 32-bit pointers in 32-bit mode and likewise for 64-bit mode. Additionally the accessible size of the registers changes in the same fashion. The register set provided consists of 32 integer, 32 floating-point and 32 vector registers. As commonly the case with RISC architectures only certain assembler instructions are able to access the memory while most instructions work only on register contents. Additionally the assembler instructions are encoded in only a few different formats with all instructions having the same fixed lengths.

When the PPU is coupled with a PowerPC Processor Storage Subsystem (PPSS) which in turn contains an L2 cache and a bus interface it is called the PowerPC Processing Element (PPE).

Special Purpose Core

The special purpose core is called the Synergistic Processing Unit (SPU) and also features a RISC instruction set with SIMD extensions, no out-of-order-execution and two pipelines. Due to the design as a gaming console only single-precision calculations are fully pipelined in the current implementation while double-precision is not and yields less performance. Theoretically each SPU can achieve a peak performance of 25.6 GFlops[1] at single precision.

The SPU contains no cache however has a large register set of 128 128-bit wide unified registers which can simultaneously store a number of integer or floating-point numbers as those are less in size than 128 bit. Therefore those 128 bit or 16 byte are also referred to as a quadword as four 32-bit words can be fit into one quadword at once. Alternatively the register contents may also be regarded as a number of 4-bit and up to 128-bit values, the exact layout and number of values stored depending on the instruction that handles the register. When the register is handled with a scalar instruction the leftmost word, bytes 0, 1, 2, and 3, are called the preferred slot. For scalar types of up to 32-bit the data is stored right aligned in the preferred slot while larger data types are stored left aligned in the entire register.

[1] This peak performance number is calculated as follows: a combined vector instruction such as multiply and add performs two operations on four values which equals eight operations each cycle. Multiplied by a clock frequency of 3.2 GHz this results in 25.6 GFlops

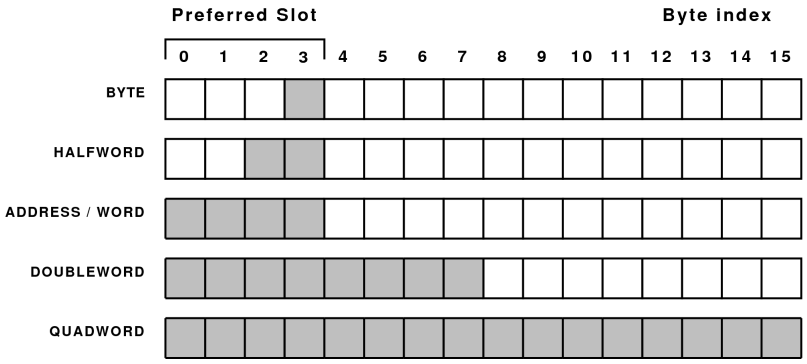


Figure 1.2: Register layout in SPU registers according to [IBM07f]

The SIMD approach taken in the SPU is pervasive meaning that all registers can contain a number of values of the same type while almost all instructions act on all values contained within one register. For example the instructions to add two registers take three registers as operands, two input and one output registers. Depending on the exact instruction used either the eight 16-bit values or halfwords from the first input register are added two their respective counterparts in the second input register and written to the correct slots in the output register or the same may be done for the four 32-bit words or for the two 64-bit doublewords.

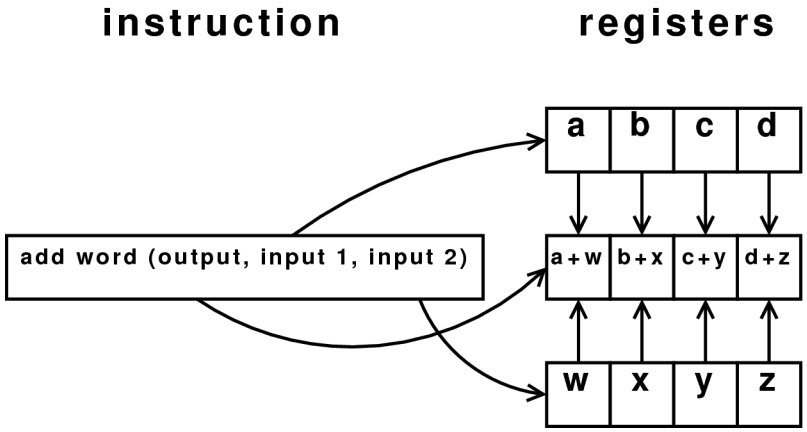


Figure 1.3: SIMD arithmetic

Currently the Cell/B.E. chip comes with 8 SPUs, each of them coupled with their own local store (LS) memory of 256 KiB. The LS is the only memory directly available to the SPU, all other memories are only accessible from the SPU via special mechanisms. The LS has to store instructions as well as data. It allows for a peak bandwidth of 51.6 GB/s. However for the SPU loading from and storing to the LS is only possible with 16 byte at once at 16-byte aligned addresses. This is ensured by having all load and store instructions in the instruction set force the four least significant bits of an address to zero resulting in a multiple of 16. Each load

and store instruction also loads or stores the contents of an entire 16-byte quadword register at once.

Exchanging data with the main memory has to be done using explicit Direct Memory Address (DMA) transfers. These transfers can be set up by the SPU or by the PPU. They are then served autonomously and asynchronously by a designated DMA controller called Memory Flow Controller (MFC) which stores the requested data in LS or main memory. The maximum bandwidth of DMA transfers is 25.6 GB/s.

The functional unit of SPU core, MFC and LS is called the Synergistic Processing Element (SPE).

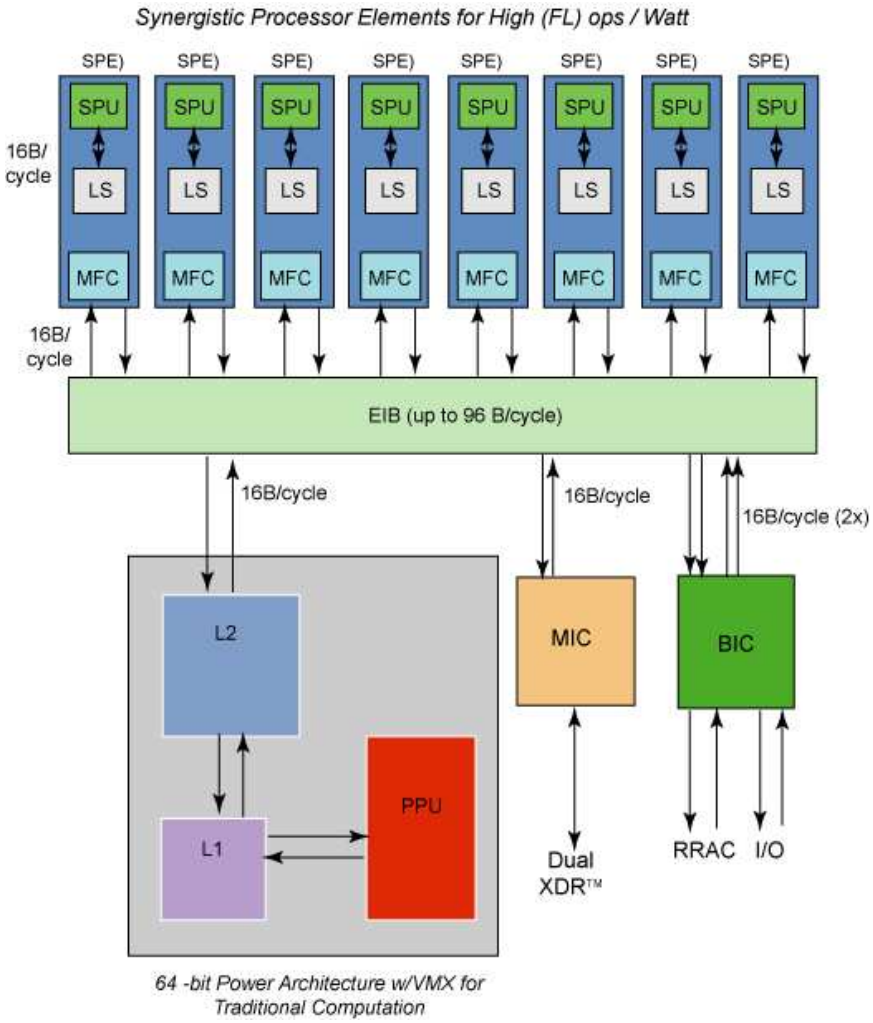


Figure 1.4: Block view of Cell/B.E.including bandwidth numbers [IBM]

Memory Flow Controller

The MFC is responsible for transferring data between LS and main memory. Once it has been instructed correctly it performs the transfers in a completely autonomous fashion in the background. Each core, PPU and SPUS, features a dedicated MFC that may transfer data both to and from the other cores.

The DMA transfers performed by the MFCs are subject to a number of restrictions

- Transfers of 1, 2, 4 or 8 byte must be naturally aligned which means that both the source and target address must be divisible by the size of the transfer. Additionally they must share the same offset to the next lower 16-byte boundary.
- Transfers of a multiple of 16 byte up to 16 KiB must be aligned to a 16-byte boundary.

Maximum performance is reached when both addresses are aligned to 128 byte and the transfer size is a multiple of 128 byte.

The MFC also supports so-called MFC lists but only on the SPU. Such a list is stored in the LS and consists of a number elements that contain a main memory address, a transfer size and a stall-and-notify bit which may be used to notify the SPU when a certain element in the list has been reached that requires some preparation. The list may contain up to 2048 elements and the restrictions for DMA transfers apply for list transfers as well. When passing this list's address to the SPU's MFC it will autonomously retrieve the elements in that list and perform the transfers. Depending on the instruction used the transfers may either load or store data from or to the main memory. While the main memory addresses in the list may be non-contiguous only one LS source or target address may be given so the area in the LS is contiguous.

The MFC also provides facilities for atomic synchronization. These allow accessing and locking an area of 128 byte in the main memory so no other unit will interfere until work on this area has been completed. Also, locking this area may be used for synchronization so as to inform other units that the locking unit is currently performing a certain work item.

Element Interconnect Bus

Connecting the PPU, SPUs, main memory and an external interface is a high speed ring bus, the Element Interconnect Bus (EIB). In order to satisfy the high bandwidth requirements of

the Cell/B.E. it provides two lanes in each direction, totaling a bandwidth of 25.6 GB/s for all connected devices.

Main Memory

The main memory is supplied by RamBus and features their Extreme Data Rate (XDR) technology and is connected to the EIB with a special Memory Interface Controller (MIC). It can provide for a bandwidth of 25.6 GBit/s but has to be soldered to the board. The first generation of IBM's Cell Blade came with just 1 GB of XDR main memory.

1.1.4 Applications

Due to the design of the CBEA a certain approach for running applications on and benefiting from the performance of Cell/B.E. is recommended. The general purpose PPU with its full instruction set is generally used to run the operating system and handle IO and networking while the specialized SPUs will only be employed for certain computationally intensive tasks. This approach usually requires careful analysis of the workload in question to determine which parts may be executed on the SPU and how they can be parallelized.

With SDK3, the latest version of the SDK which will be further described in the next section, many different forms of cooperation between the SPUs and PPU are possible. To synchronize the units a number of methods is available to them. The SPUs and PPU can send small messages via so-called mailboxes and signals among themselves or access shared data structures in main memory or in one of the SPUs' LS via DMA transfers. The most straightforward approach is to let the PPU create some SPU threads and execute an algorithm on them while controlling and synchronizing the state of the SPUs. However it is also possible to just start up a number of SPUs which can synchronize themselves. Additionally it is possible to exchange the program code while leaving the data in place thus allowing to operate on one set of data with multiple programs.

Due to the lack of a cache on the SPU most applications use part of the SPU's LS to create special buffers for DMA transfers and other regularly required data. Because of the high bandwidth of the LS this approach can be tailored to the algorithm's needs and delivers very high performance. A popular approach is to set up a double-buffering scheme where a DMA transfer is pulling data from the main memory into one buffer while the actual program is working on

the data available in the other buffer. Once this step is complete buffers are switched. This scheme is possible due to the autonomous nature of the DMA controller which allows the SPU to keep executing code while data is being fetched from the main memory.

To summarize running programs on Cell/B.E. does not automatically result in a higher performance as may be the case with traditional multi-core architectures. Yet Cell/B.E. provides a great potential for specialized applications once they have been adapted to the unique architecture.

1.1.5 Linux on Cell/B.E. and Cell SDK

The Open Source operating system *Linux*[2] has been extended by STI to be able to run on Cell/B.E. and make use of the SPUs. IBM provides a complete Software Development Kit[3] for developing SPU-accelerated applications on Linux. It contains a port of the Gnu Compiler GCC[4] which can compile C, C++, Ada and Fortran applications for the SPU, a run-time library to manage SPU access (libspe2) [IBM07d], additions to the Gnu Debugger (GDB)[5] to debug Cell/B.E. applications and a Cell/B.E. simulator which may be used on non-Cell/B.E.-hardware to simulate such a system[6]. A special C library for embedded systems with a low memory footprint, newlib[7] is used to provide SPU programs with frequently required functionality. In order to facilitate access to the SPU's LS it can be memory-mapped into the PPU's memory thus allowing the PPU to read and write the LS as if it were ordinary memory, albeit at a lower speed.

System calls from the SPU

System calls on the SPU such as opening files or printing to a terminal are handled using special PPE-assisted-calls which instruct the PPU to service an SPU's request. This mechanism is based on the stop-and-signal functionality and works by the SPU issuing a special assembler instruction called `stop`. The only argument to this instruction is a 14 bit stopcode. Once this command is executed the SPU is stopped and the PPU notified. Using the libspe2 runtime

[2] http://www.kernel.org
[3] http://www.bsc.es/projects/deepcomputing/linuxoncell/
[4] http://gcc.gnu.org/
[5] http://sourceware.org/gdb/
[6] IBM has also extended its proprietary XLC compiler family to be able to compile code for the PPU and the SPU. As they have not been used in this thesis they will not be mentioned further
[7] http://sourceware.org/newlib/

management library applications can add individual callback functions for each stopcode. By convention the stopcode values may range from 0×2100 to 0x21FF with the first 4 available values being reserved for operating systems functions such as IO operations.

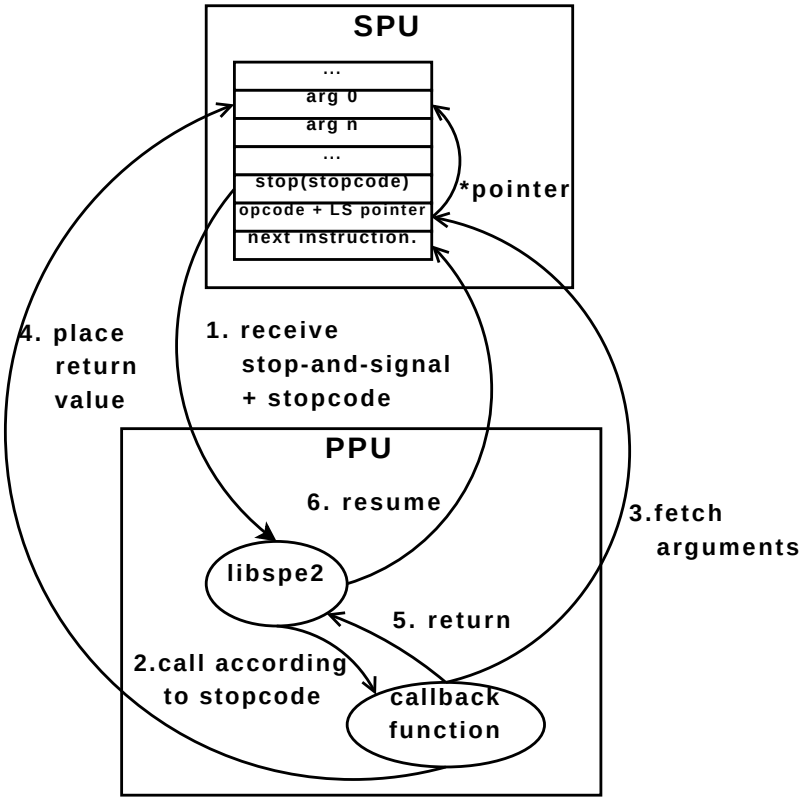


Figure 1.5: Stop-and-signal mechanism

The stop-and-signal notification is initially handled by a libspe2 function which depending on the stopcode calls the appropriate callback. The callback functions are being passed the address of the respective SPU's LS in the memory map and the offset of the originating `stop` instruction within the LS. Again by convention the next 4 bytes directly following the `stop` instruction contain further data for the callback function. This data usually consists of an opcode designating the actually requested functionality within the set of functions provided by this callback as well as a pointer to a block of arguments within the LS. Due to the SPU's 16-byte alignment restriction each argument is usually 16 byte large so that it may be handled as four 4-byte or two 8-byte elements.

Once the callback function on the PPU is done it optionally stores a return value directly in the LS in place of the first argument in the argument block. Again, the callback function may place up to 16 byte as a return value in varying granularities. Execution on the SPU is then resumed with the instruction following the LS pointer.

Linking SPU and PPU programs

In order to allow running an SPU program from a PPU program Linux on Cell/B.E. provides a mechanism to embed the SPU part within a PPU program as described in [IBM07a] Chapter 14, *Objects, Executables and SPE Loading.* For this mechanism the CBEA Embedded SPE Object Format (CESOF) has been defined.

Both the SPU and the PPU toolchain produce ELF objects with disjoint address spaces. Using the `ppu-embedspu` tool allows taking the SPU object file and wrapping it into an object file suitable for linking with the PPU program. This object file provides only one symbol of the type `spe_program_handle_t` which can be used in the PPU program and then passed to the appropriate libspe2 functions that load or run the associated SPU program.

1.2 Java

In its barely 15 years of existence Java has risen to become one of the most famous programming languages as regularly shown by the *TIOBE Programming Community Index*[8]. A good introduction to Java is given in [Ull07]. Some important aspects concerning its history and design goals as well as its current state will be reproduced here.

1.2.1 History

The origins of Java trace back to *Project Oak* which was started in 1991. It was originally envisioned to provide a virtual machine for cable TV set top boxes and other media devices and enhance them with interactive functions. This virtual machine would act as an abstraction layer and allow the code to run on a multitude of different platforms as long as an implementation of the virtual machine was available. However this market did not prove very rewarding so a change of directions was made towards the Internet. With the virtual machine it was possible to download small programs, so called applets, from the Internet and execute them in the browser. This mechanism also maintained a high level of safety due to the virtual machine acting as an intermediate layer between the applet and the computer's resources. After the great success of the applets Java was soon extended to provide functionality for stand-alone desktop programs

[8] http://www.tiobe.com/tpci.htm

as well as server applications and now has become one of the most popular and widely spread programing languages.

The fact that Java and its source were made available at no cost by Sun Microsystems also contributed to the wide adoption of the Java platform. However since it was developed in a closed-source non-free fashion the development of many free JVMs, free Java compilers and the free class library *GNU Classpath* to escape this *Java Trap* was started. Beginning in November 2006 Sun started to publish the source code of their java compiler and the Hotspot virtual machine under a free license which was soon followed by the majority of the class library.

1.2.2 Implementation

Java was originally designed with a number of goals in mind. While other programming languages have also fulfilled some of these goals before Java was the first to combine these leading to its great success. Some of the most important aspects are listed here.

Portability

Java programs should support a *compile once, run anywhere* policy. To achieve this goal a machine-independent bytecode language was defined. A Java compiler could then compile Java source code to bytecode which is interpreted by the JVM. Additionally Just-in-Time (JIT) compilers may translate the bytecode at runtime into the machine code of the current architecture the JVM runs on.

Object-Orientation with primitive types

To provide the best of both object-oriented and procedural languages Java supports both objects as well as a few primitive types. Pure object-oriented languages, those without any primitive types, such as Smalltalk had proven too be to restricting performance-wise even though they provided a clearer view on the program.

Along with object-orientation Java also implements access permissions. Classes and class members may be qualified with further keywords as shown in table 1.2 which define who may access a certain class or member.

	private	*default*	protected	public
class	-	package access	-	no restrictions
member	same object	private + same package	*default* + subclasses	no restrictions

Table 1.2: Overview of Java access permissions

Security

Due to the virtual machine as an intermediate layer between the Java program and the host system it is possible to restrict usage of the host system's resources. Fine-grained security permissions can be defined depending on the origin of the Java program. This allows to fetch bytecode via network from untrusted sources and safely execute it in the virtual machine. The most well-known case for this are the Java applets, small programs which are embedded into a website and loaded from the potentially malicious server hosting the website. Additionally a JVM also includes a bytecode verifier which checks all bytecode instructions before their execution to ensure that they do not execute potentially harmful instructions such as illegal branches and access to private data.

Stack based

The Java bytecode was designed to simulate a stack-based architecture. Since no assumption about the available register set or stack on the target architecture is done this allows the JVM to perform appropriate machine-dependent optimizations. For example CISC architectures such as x86 commonly provide only a few registers while RISC architectures such as PowerPC usually come with a larger register set. Depending on this the JVM may try to keep more or less stack elements in registers.

Multi-Threaded

Modern computer systems ranging from mobile devices over desktops to servers almost always have to perform multiple tasks concurrently. In order to accommodate this fact the Java platform supports multi-threading at the language, the library and the JVM level. This means that the language provides low-level instructions to control multi-threading, the library contains higher-level methods and the JVM must also support multi-threading usually mapping Java threads to native operating system threads.

Automatic memory management

As opposed to C++ the memory management in Java is automatic. This means that the required space on the heap for an object is allocated and released automatically. The allocation is handled transparently by the JVM while the release is performed by a garbage collector (GC). The GC is able to independently determine which objects are no longer in use and can release the space they require.

1.2.3 Multi-threading in Java

As explained multi-threading was one of the design foundations of Java. Therefore the required mechanisms for thread-safety reach through every layer of the Java platform from the language over the bytecode to the JVM. Still, the usage is kept simple with only a few additional constructs. While each thread has its own distinct stack they all share the same heap which is the main requirement for thread-safety.

The basic component is the object lock or monitor. Every object and every class provides such a monitor that can be only acquired by one thread at a time. Other threads trying to acquire the same monitor will stall until the monitor has been released by the thread holding it currently. The monitor is actually implemented as a counter where a value of 0 means that the lock is free. Acquiring the monitor increases the count by one, releasing it decrements it by one. The monitor is never accessed directly but only through the `synchronized` keyword. When encountering this keyword the Java compiler will automatically generate the correct bytecode sequence for acquiring a particular lock. `synchronized` can be used in two situations which will be explained followed by an overview of the higher-level mechanisms Java provides for thread coordination.

Synchronized functions

Any Java method except for the constructor can be qualified with the keyword `synchronized`. This means that only one thread may enter any synchronized method of a particular object or a static synchronized method of a class at a time. Since Java functions are reentrant, meaning that they can call themselves, the thread owning the monitor may enter the method again thus incrementing the count on the monitor. Once the method is left by returning or throwing an exception the counter is decremented again allowing other threads to enter synchronized methods

of this object or class.

Synchronized statements

In order to allow more fine-grained concurrency only certain blocks may be protected by a monitor. An arbitrary object reference, not necessarily the one of the object containing the block, may be used as a monitor. Again, only one thread may enter any synchronized block or method of the object or class used as the lock at a time.

```
1 ...
2 synchronized (someObject) {
3   /* critical part, only one thread may enter this at a time */
4 }
5 ...
```

Listing 1.1: Example of a synchronized Java block

Volatile keyword

As a more lightweight replacement for synchronized statements the Java language also provides the `volatile` keyword. When applied to an instance or class field the JVM ensures that every access to this field is performed in memory instead of the thread's stack or registers. This allows every thread to always see a consistent representation of the field. Additionally for architectures that do not support 64-bit types natively 64-bit operations may require separate operations on the two 32-bit parts of the type. In this case `volatile` ensures that this operation is performed atomically without interference by other threads. However, for more complex operations such as incrementing a field using `volatile` on it does not guarantee consistency. Incrementing actually requires two distinct accesses to the field, one to get and one to set its value, therefore such operations are not protected by `volatile`.

Thread coordination

A number of methods are provided by the `Object` class that allow for the coordination of threads. Since all classes are derived from `Object` these methods are available for every instance of a class. Additionally the Thread class provides one important method.

1. `wait(timeout)`: release the object's lock, enter the object's waiting set, suspend execution of the current thread, wait for the optional timeout to expire

2. `notifyAll()`: notify one or all threads from the object's set that they may continue execution, do the same as `wait()`
3. `join()`: Calling this method on an instance of `Thread` makes the current thread wait for the other thread to finish.

Except for `join()` these methods make sense only when called inside of synchronized blocks or methods. Calling `wait()` makes the current thread release the monitor allowing other methods to enter synchronized statements of the referenced object, suspend its execution and enter the waiting set of the object. It is usually used when the thread cannot perform its desired action inside a synchronized block for example because of missing data. `notify()` or `notifyAll()` performs the same actions as `wait()` while also notifying other threads in the waiting set that they may resume execution. This may be used to signify to other threads that there is now data available. `join()` provides a synchronization barrier ensuring that all participating threads have reached a certain point. When calling this method on an instance of `Thread` the current thread waits for the called thread to finish and then resumes execution.

A typical example for this mechanism is the *Producer-Consumer-Pattern.* This programming pattern consists of one shared buffer and two threads, both of which work on the same buffer. The producer thread produces or stores values in the buffer and the consumer thread consumes or removes them. The buffer provides two synchronized methods, one that removes and returns a value from the buffer and one that stores a value. When started both threads try to enter their respective method although only one may successfully do so because of the synchronized methods, the other thread has to wait. The consumer then checks if the buffer contains data in which case it consumes it and calls `notify()` to signal the producer that the buffer is now empty and new data is required. If the buffer is empty the consumer can simply call `wait()` as there is nothing to do. The producer in contrast checks if the buffer is empty. If it is it produces new data and calls `notify()` to indicate to the consumer that it may fetch data. If the buffer is not empty the consumer calls `wait()` as there is nothing for it to do.

Summary

Using the available variations of the `synchronized` keyword thread-safety in Java, that is making sure objects and their data are always in a consistent state, can be ensured. In its simplest form it is usually achieved by making fields of the object private so that they may not be directly accessed from outside the object. Synchronized public methods to get or set these fields are provided. These methods may check the validity of the new value or other conditions. Once

the method is done and the lock released the object is still in a defined state since no other threads were able to interfere with this operation. If the fields of an object can be directly accessed by multiple threads inconsistencies may occur. This comes from the well-known fact that most operations such as incrementing a value are not atomic. Incrementing for example requires fetching the current value first, incrementing it and then writing it back. If the scheduler switches to another thread after the first thread has just fetched the value the result will be incorrect.

To summarize this means that Java may provide the right tools to achieve thread-safety however the responsibility to actually ensure this clearly lies with the Java programmer. This point is very important to the concepts for shared heap access described in later chapters.

1.2.4 Performance

While these goals have helped Java to increase its outreach and appeal they came at the price of a low performance due to the virtual machine interpreting the bytecode at runtime. However in recent years the addition of Just-in-Time (JIT) compilers has greatly decreased this problem and pushed Java closer to a performance level competitive with C and C++. JIT compilers are called at runtime and dynamically translate the Java bytecode to native machine code. Using this approach allows maintaining the machine-independence of the compiled Java program. Additionally the JIT compiler is able to take into account data collected at runtime such as the flow of the program and external events and then reorganize the structure of the code for example with method inlining and branch predictions. This way it is theoretically possible to achieve a higher performance with a JIT compiler than with a traditional ahead-of-time compiler as has been shown in [BDB99].

Around the year 2000 there was a huge interest surrounding Java for high-performance computing, organizations such as Java Grande[9] still document this. However in recent years this trend has slowed down and corporate use of Java extended into the domain of server systems and large clusters.

[9] http://www.javagrande.org

1.2.5 The Java platform

For the first years the Java programing language and the Java platform, which comprises among others the JIT compiler, bytecode interpreter, garbage collector and class libraries, were perceived as one unit. Due to the intermediate language in the form of bytecode however it is possible to use any language to program for the Java platform as long as it can be compiled to bytecode. In the last years many languages other than Java have been developed for the Java platform. Most of these are dynamically typed and thus allow for faster development cycles than the statically typed Java. As a result of this Java has now expanded from providing one language for many platforms to many languages for many platforms.

Due to this paradigm shift the distinction between the Java language and the Java platform is emphasized lately. Furthermore different packages are provided for regular users and for Java developers. For regular users Java is usually delivered in the form of a Java Runtime Environment (JRE) consisting of a JVM and the compiled class libraries which is enough to run Java programs. If other development tools such as a compiler and documentation generator as well as the source code to the class libraries are added the bundle is referred to as Java Development Kit (JDK).

1.3 Java on Cell Motivation

If it is possible to enable Java programs to employ Cell/B.E. for their calculations a whole new world would open up for both Java and the Cell/B.E. Java programs could gain a performance boost from using the Cell/B.E.'s SPUs while the CBEA could profit from the vast number of Java programmers and applications already existing today and reach an even wider audience.

While Java and the CBEA share a number of attributes it currently is not possible to have Java programs benefit from the raw computational power the Cell/B.E. provides. Some of these attributes and other positive factors are discussed here.

1.3.1 Multi-threading

Multi-threading is an integral part of Java and firmly embedded into its core. Its usage spans from desktop applications using different threads to update the user interface and act on external

input to server programs using multiple threads to serve multiple requests. Similarly it is crucial for Cell/B.E. programs to subdivide tasks so they can be executed in parallel on the SPUs.

1.3.2 High-performance computing

In recent years Java has become an increasingly important force in the High-performance computer (HPC) sector thanks to huge improvements in JIT compilers and in its memory management. By now it is able to compete with C programs performance-wise while providing a much higher level of abstraction for the programmer. The CBEA has been designed from the ground up to provide extreme performance. It has already been used in a number of HPC projects, most notably the ongoing *Roadrunner* project[10] designed to become the world's fastest computer with a performance of one petaflop in 2008.

1.3.3 Provide a homogeneous environment

As the JVM acts as an intermediate layer hiding any architectural differences this fact may also be used to hide the heterogeneous nature of the CBEA from the Java programmer. MFC transfers and shared access to data across different memories can be performed implicitly and controlling the SPU threads may use the standard mechanisms available in Java. When using one of the currently supported languages for Cell/B.E. such as C/C++, Ada and Fortran the programmer has to take care of all this manually.

[10] http://www-03.ibm.com/press/us/en/pressrelease/20210.wss

2 Java on Cell

In this section a number of concepts to enable Java on Cell/B.E. will be presented and evaluated along with the software components they require to be fulfilled. Based on this evaluation the most promising concept will be further pursued and its base components will be introduced in greater detail.

2.1 Concept

Due to the uniqueness of the CBEA three different approaches to enable Java seem viable. Depending on the approach chosen the architecture can be seen as either a distributed memory or shared memory one. The goal is to extend the JVM in a way that hides the underlying architecture and thus create what is essentially a distributed shared memory system (DSM) in which the entire memory space is accessible transparently for all nodes. Such a JVM is commonly referred to as a *single system image* as it looks to the user like a single system. Further information about DSM systems and their implementation in Java is given by [Fen01].

If the SPEs are seen as independent nodes with the LS as their dedicated memory and a dedicated processing unit then the whole architecture can be regarded as a distributed memory one. Communication between the nodes is performed by message passing, however with a low latency and a high bandwidth compared to other distributed memory architectures. This is due to the fact that the nodes are not connected by a network but instead reside on the same chip. The message passing in this case would be implemented with mailbox messages and DMA transfers. In the other case the SPU is seen as an auxiliary processing unit with the LS as its cache and the main memory being the central storage area. With this model the CBEA is more closely related to a shared memory multi-core architecture where all processing units can access the entire memory space.

2.1.1 JVM running inside SPU

A more or less complete JVM is ported to run on the SPU as shown in figure 2.1. This requires the JVM to fit entirely into the 256 KiB memory available to the SPU while still leaving enough space for both the actual Java program and its runtime data, the stack and the heap. Additionally the class library has to be made available to the SPU. Due to its size, which is around 16 MiB for GNU Classpath, a free class library, this requires a concept to swap the needed code in and out of the LS as required. While the available memory does not seem much certain JVMs have been designed to work on embedded devices with similar memory constraints.

This approach makes it possible to run both interpreted and JIT-compiled code or a mixture thereof on the SPU. However it seems doubtful that a Java interpreter on the SPU is able to achieve any significant performance as it has to do a lot of expensive branches and memory lookups. A JIT compiler promises much higher performance however requires additional space for the compiled code. Once a method has been compiled and executed on the SPU its machine code can be swapped out to the main memory so only the currently needed functions have to be kept in the LS. This also erases the time needed to compile a function on subsequent calls as it is still available from the code cache in the main memory.

As the SPUs act as mostly independent nodes in this model a message passing scheme or a technology such as remote method invocation (RMI), the Java standard for distributed Java systems, is required to allow them to cooperate. Other problems of this domain such as distributed garbage collection due to each SPUs' local heap also apply.

2.1.2 Compile Java code to native SPU code

A second concept is based on the idea of using a compiler which can compile Java source code into native machine code for the SPU ahead-of-time like a traditional compiler. This approach promises a high performance due to the use of native code without the penalty of compilation during runtime. It could also be extended to work in a similar fashion as the currently supported languages for the Cell/B.E. For these, compilers are provided which can generate native code for both the SPU and the PPU with additional runtime libraries to start the compiled SPU program from the PPU. The basic concept is depicted in figure 2.2. A Java compiler would also face the same problems as the existing compilers which include synchronization between SPUs and PPU and code size often being larger than the available LS size. This is especially problematic for Java since dynamic libraries are not available on the SPU thus requiring to link the class

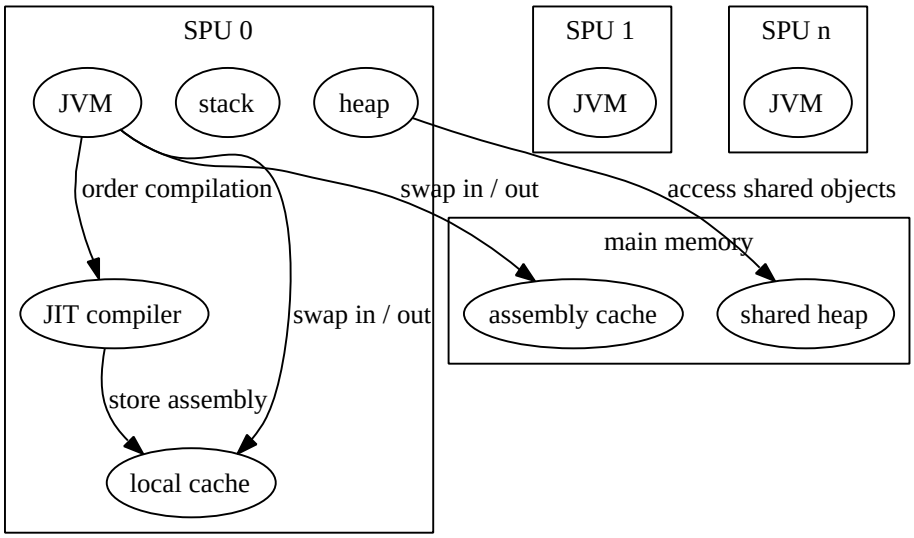


Figure 2.1: Run an entire JVM on the SPU

libraries statically into the SPU program.

2.1.3 Use a JIT compiler to offloads functions to the SPU

In the third concept a host JVM will be run on the PPU which offloads computationally intensive methods to the SPU. The PPU executes common functions and service the SPUs' request to central JVM infrastructure components such as the classloader. Employing the SPUs is done by having the JVM's JIT compiler generate native machine code for the SPU which can be executed right away without additional performance penalties. The JVM can act as a central synchronization point to exploit the multi-threading capabilities of Cell/B.E. As an additional benefit the main memory can be used as a cache to store compiled functions and provide them to the SPUs as needed. A mechanism to swap compiled code in and out of the SPUs LS has to be developed. Furthermore objects on the heap have to be mutually shared and accessed by all SPU and PPU threads. Thus the common problem of synchronized access to objects and functions also has to be taken care of with the addition that objects might have to be temporarily moved to an SPU's LS to work on them. A rough overview of this system is shown in figure 2.3.

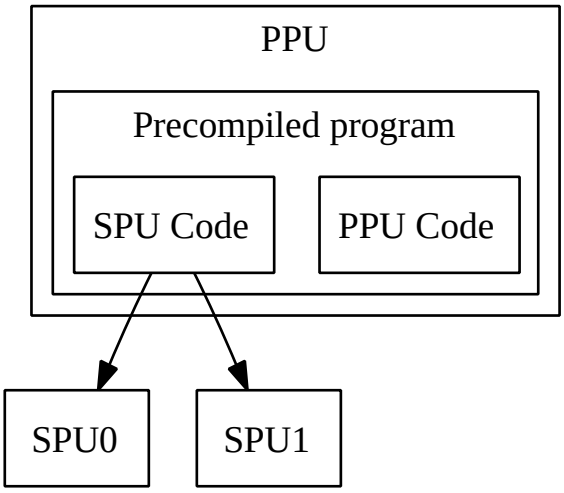


Figure 2.2: Compile Java code to native SPU code

2.2 Feasibility study

Each of these three approaches comes with its own advantages and disadvantages which determine its feasibility and will be discussed here.

1. As discussed before the first approach will be best fulfilled by a JVM which has been designed for use in embedded systems. Due to the similarities in the SPU and the PPU instruction set it is desirable to choose a JVM which already provides an interpreter or JIT compiler for PowerPC. One example of this is *JamVM*[1] which comes with a binary size of around 160 KiB and includes a PowerPC interpreter. However due to their nature interpreters commonly achieve only performance levels at least an order of magnitude lower than those of JIT compilers as shown by the performance comparisons on [Shu04]. Therefore this approach does not look very promising. Another free JVM which provides a PowerPC JIT compiler is *Cacao*[2]. It can be stripped down to a binary size of around 600 KiBs which is still too large for the SPU's LS. Additionally the thread synchronization overhead in this concept is larger. Either way this concept has many problems as explained above. As even the basic idea does not sound promising in terms of performance this concept will not be further pursued.
2. The second concept is based on *GCJ*[3], part of the Gnu compiler collection, which can

[1] http://jamvm.sourceforge.net/
[2] http://www.cacaojvm.org
[3] http://gcc.gnu.org/java/

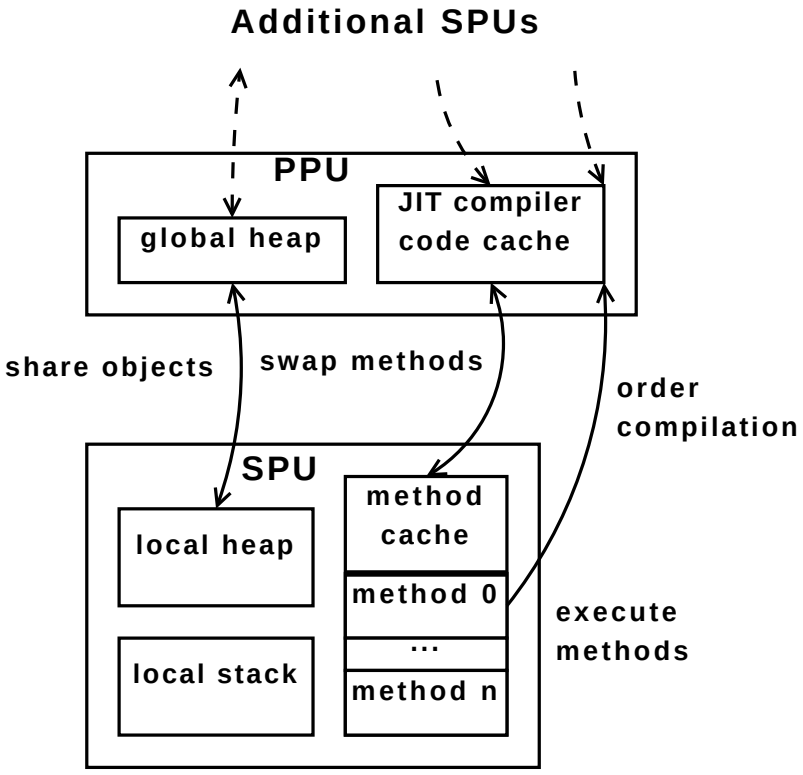


Figure 2.3: Use a JIT compiler to offload functions to the SPU

compile Java code into native machine code for many platforms. As both a frontend for Java as well as a backend for the SPU, the respective GCC terms for the component supporting a certain language and the one supporting a platform, already exist these two have to be coupled together. With this it will be possible to compile a Java program into native SPU machine code before runtime and execute it on the SPU. This however means that the program is only executed on a single SPU, allowing multiple SPUs and the PPU to cooperate requires additional work. In principle a GCJ for the SPU should profit from the large number of optimizations built into GCC and therefore in GCJ. However GCJ's performance is still sub-par [Shu04]. Additionally GCJ only conforms to the Java 1.1 standard.

3. As with the first approach a JVM with support for PowerPC provides a good starting point. The only free JVMs with PowerPC JIT compilers still being actively developed are Cacao and *Kaffe*, however the Kaffe JIT compiler for PowerPC is broken as of this writing. Cacao on the other hand has been developed from the ground up to support JIT compilers for RISC processors and thus promises a high performance. Since no other compilation technology so far could rival JIT compilers for the Java platform this match seems to be suitable. Additionally the possibilities of JIT compilers such as dynamic optimizations provide interesting opportunities. These could for example be applied to

> determine how to partition workloads between the different Cell/B.E. cores or anticipate data transfers based on previous and repeated occurrences.

Due to the reasons discussed the third approach promises the most potential. It will therefore be further pursued in this thesis.

2.3 Components

Summarizing from the feasibility study two main components will make up Java on Cell. These components will be briefly introduced in this section while the chosen JVM will be described in greater detail in chapter 3.1.

2.3.1 Cacao

From the Cacao manual [Kea04]:

> CACAO is a research Java virtual machine. From the beginning it was designed for 64 bit architectures and was based on a just-in-time compiler. To avoid two different stack frame formats no interpreter is included. The compiler is so fast that it does not matter to compile code which is just executed once. One of the aims of CACAO is to keep the system small and simple which makes CACAO also well suited for embedded systems. It has been used to explore new just-in-time compilation techniques, fast program analyses and improvements for run time systems. Many of our developments turned out to be useful and had been included in the version of CACAO which we now distribute under the GNU general public license.

The Cacao project was originally started in 1996 and is still being actively developed by a small community following its release under the GPL in 2004. By now it features an interpreter and has been extended to support a number of different RISC and CISC 32- and 64-Bit platforms including Alpha, x86 and x86_64, ARM, PowerPC and MIPS with varying degrees of support for the interpreter and the JIT compiler. The JIT compiler has proven to create efficient code however still has room for improvements for certain workloads as the performance comparison on [Gar07] shows. Cacao uses GNU Classpath for its class libraries.

2.3.2 GNU Classpath

GNU Classpath was originally created as a response to the Java Trap which describes the fact that Sun provides its class libraries at no cost including access to the source code while forcing a non-free license on developers. It was started in 1998 along with its own JVM which was dropped as the project matured. Development was then focused on providing a stable base of class libraries for other JVMs to use. This move was highly successful and Classpath was adopted by many free JVMs. With the release of the Sun JDK including the class libraries and the *HotSpot* JVM under the free GPL v2 license starting in November 2006, the future development of Classpath is unclear as Sun's JDK provides a more complete implementation. Classpath will most likely be used to fill in the gaps left in Sun's JDK by the remaining proprietary parts. One Open Source project to attempt this is IcedTea. As Classpath provides a prerequisite for Cacao it requires no further work and thus will not appear in subsequent chapters.

3 Architecture and Design

Due to the reasons described in section 2.2 the following work will be based on the Cacao JVM and GNU Classpath. This section will give an overview of the Cacao architecture and the resulting necessary steps to port Cacao to the Cell/B.E. It will begin with a rough overview of the steps the JIT compiler performs and will describe the code generator that outputs the assembler instructions in greater detail. Afterwards the GC and its integration in Cacao will be explained followed by some mechanisms used in Cacao for lazy resolving and the different schemes for calling a method. Finally some interesting points in the file hierarchy of the Cacao source code will be pointed out.

3.1 Cacao architecture

The Cacao handbook gives a first rough overview of the Cacao architecture and especially the JIT compiler whose port to Cell/B.E. will take the most manual work. It describes the way classes and its members, fields, methods and nested classes, are loaded by the JVM when requested by the Java program, how the required data structures are organized during runtime, how methods are invoked and how exceptions are handled. It then goes on to describe the architecture of the JIT compiler which will be summarized here. Further architectural parts will be introduced as required.

3.1.1 Compiler steps

The Cacao handbook also describes the challenges of porting the JIT compiler to different platforms, among them PowerPC in 32-Bit mode. Due to the fact that Cacao was originally started with 64-Bit RISC processors in mind and had already been ported to the 32-Bit x86 architecture most of the upcoming problems had already been solved. Two specific parts required further work, 64-bit arithmetic and the calling conventions for native functions. 64-bit arithmetic is

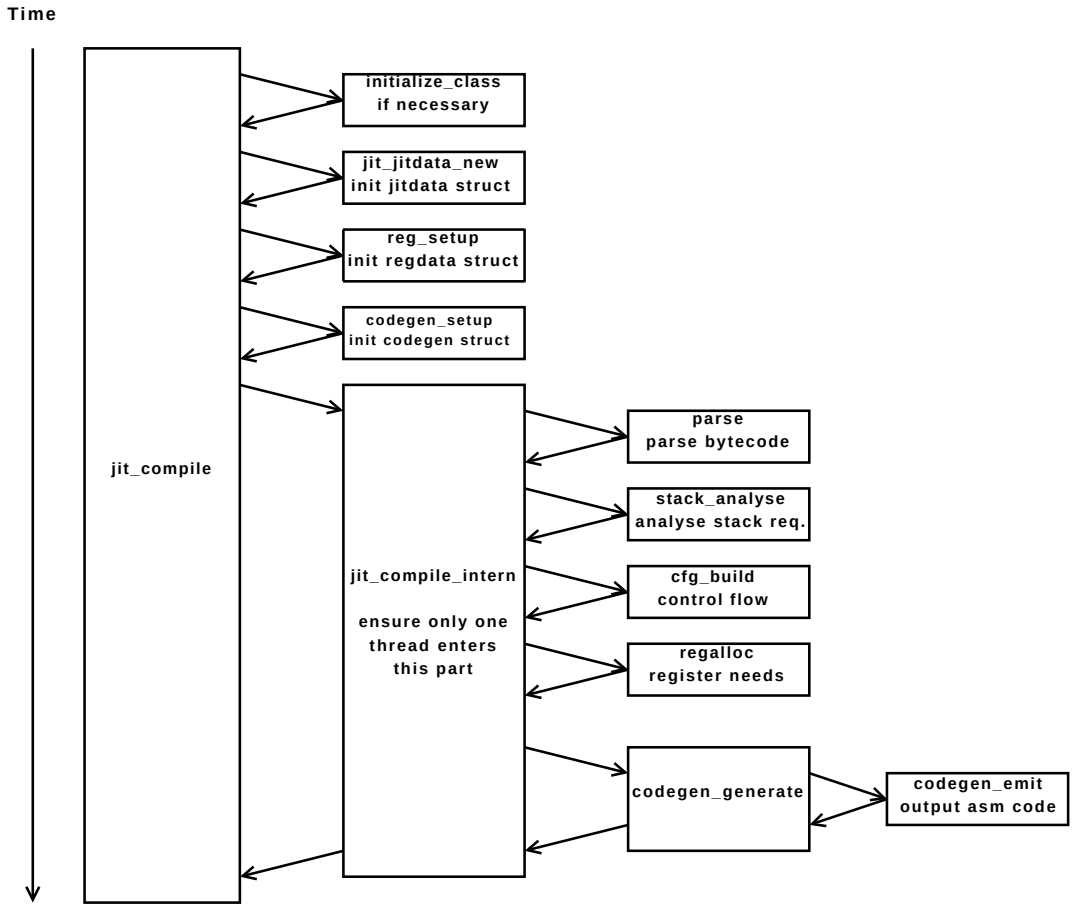


Figure 3.1: Control flow of the JIT compiler

performed by doing more or less the same operation on two 32-Bit registers. Calling native functions required integrating the respective platform's application binary interface (ABI). The ABI for a platform which is the sum of an architecture and an operating system defines how the available registers are to be used so as to ensure interoperability between code compiled by different vendors and technologies such as a JIT compiler or interpreter. This includes defining the link register, the stack pointer register, registers containing arguments and return value, the callee and caller saved and the temporary registers as well as the setup of the stack frame.

The final goal of the compiler is to translate the stack-based Java bytecode into register-based native machine code for the target machine. In order to avoid unnecessary register or memory copy instructions the compilation is done in three stages and spends a considerable amount of time analyzing the bytecode. These three steps will be described in the following.

Basic block determination and internal representation

The smallest unit apart from individual bytecode instructions that the compiler works with is the *basic block* which always consists of one or more bytecode instructions. During basic block determination the compiler iterates over the respective method's bytecode and evaluates for each instruction if it marks the end of the current basic block. Basic block ends are most instructions that represent a conditional branch such as `if-else`, `switch`, throwing an exception or simply returning from a method. The instruction that marks the end of a basic block will not be included in the current block but will be the first instruction in the next block. Method invocation however does not necessarily end a basic block which in principle allows for method inlining, that is including the instructions for a submethod in the calling function's code. This allows saving the overhead of a method call such as setting up the stack frame. This basic block determination is performed to make the calculations of branch targets for the JIT compiler easier as each basic block marks a potential branch target. Additionally the information about blocks can be used for optimizations for example determining unused code or recompilation for frequently used blocks.

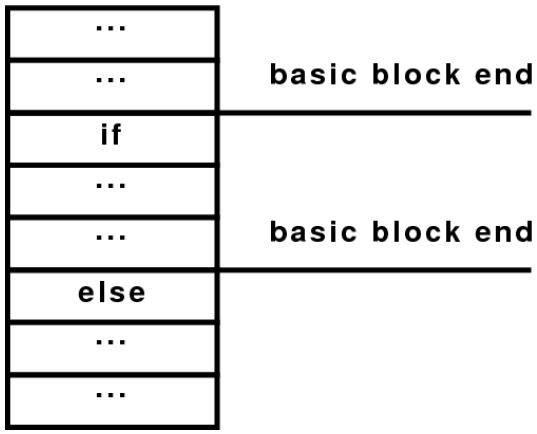


Figure 3.2: Basic block layout example

In order to simplify the transition from one basic block to another the stack is mapped to the registers with a fixed interface so the following basic block can always expect a certain stack slot at a certain register. This approach was chosen by the Cacao developers after empirical research on their part have shown that that the stack depth at a basic block boundary is rarely greater than six. As most architectures provide more than six registers the stack can be entirely mapped to registers.

At this stage the bytecode instructions are also translated to an internal representation consisting of one operator, two operands and a pointer to a stack structure. Certain bytecode instructions

such as those folding the operand into the opcode eg. `ICONST_X` which pushes a constant value on the stack have to be split up into the generic operator and the respective operand to fit the fixed representation. This approach allows for faster execution of the subsequent stages.

Stack determination and instruction combining

The next stage analyzes the used values and tracks where they are loaded and where they are consumed in order to avoid unnecessary copies and loads. For this purpose the stack is represented as a linked list allowing the compiler to look up if and where in the stack an operand resides. Based on this information the register requirements of the basic block can be calculated.

In this stage some sequences of instructions are combined into internal instructions that can be executed more efficiently by the hardware. One example is loading a constant that is a power of two and then using it for multiplication or division. This sequence is combined into one instruction which can be efficiently handled at the machine code level with byte-shifting.

Register allocation

The register allocator determines for each instruction which registers its machine code translation may use for input and output. Due to the fact that Cacao was originally designed for register-rich RISC architectures such as Alpha and MIPS a simple first-come-first-serve approach is used for the register allocation of the machine instructions. This means that registers will be reserved as long as the value they contain is required after which they can be allocated again. If too many values have to be kept available old values may be swapped out to the stack.

3.1.2 Code generator

The actual code generator for the JIT compiler consists of two major parts, macros to output the machine code translation of the bytecode of a method and functions and data structures to manage the data segment. The two components will be described in this section.

The code generator macros

The code generator macros are defined in the architecture-specific file `codegen.h`. They mostly consist of a mapping of assembler commands and very few higher level functions to macros which in turn call functions or macros that output the correct binary sequence to encode the requested instruction. As the PowerPC instruction set only has a few different syntax forms for its instructions the binary output macros as shown in 3.1 provide a concise view of the binary instruction stream. The formats vary in the number of operands which can be registers or immediate numbers. These binary output macros are used by further macros which are mapped to the instruction set of the respective architecture. A sample of these macros is shown in 3.2. These macros will eventually be used by the JIT compiler as it decides for each bytecode how to translate it to native machine code.

```
1  /* macros to create code ************************************************/
2
3  #define M_OP3(opcode,y,oe,rc,d,a,b) \
4      do { \
5          *((u4 *) cd->mcodeptr) = (((opcode) << 26) | ((d) << 21) | ((a) << 16) | ((b) << 11)↩
               | ((oe) << 10) | ((y) << 1) | (rc)); \
6          cd->mcodeptr += 4; \
7      } while (0)
8
9  #define M_OP3_GET_A(x)                (((x) >> 16) & 0x1f  )
10 #define M_OP3_GET_B(x)                (((x) >> 11) & 0x1f  )
11
12 #define M_OP4(x,y,rc,d,a,b,c) \
13     do { \
14         *((u4 *) cd->mcodeptr) = (((x) << 26) | ((d) << 21) | ((a) << 16) | ((b) << 11) | ((↩
               c) << 6) | ((y) << 1) | (rc)); \
15         cd->mcodeptr += 4; \
16     } while (0)
17
18 #define M_OP2_IMM(x,d,a,i) \
19     do { \
20         *((u4 *) cd->mcodeptr) = (((x) << 26) | ((d) << 21) | ((a) << 16) | ((i) & 0xffff));↩
               \
21         cd->mcodeptr += 4; \
22     } while (0)
23
24 #define M_INSTR_OP2_IMM_D(x)           (((x) >> 21) & 0x1f  )
25 #define M_INSTR_OP2_IMM_A(x)           (((x) >> 16) & 0x1f  )
26 #define M_INSTR_OP2_IMM_I(x)           ( (x)        & 0xffff)
```

Listing 3.1: PowerPC binary output macros for the JIT compiler

```
1  #define M_IADD(a,b,c)                  M_OP3(31, 266, 0, 0, c, a, b)
2  #define M_IADD_IMM(a,b,c)              M_OP2_IMM(14, c, a, b)
3  #define M_ADDC(a,b,c)                  M_OP3(31, 10, 0, 0, c, a, b)
4  #define M_ADDIC(a,b,c)                 M_OP2_IMM(12, c, a, b)
5  #define M_ADDICTST(a,b,c)              M_OP2_IMM(13, c, a, b)
6  #define M_ADDE(a,b,c)                  M_OP3(31, 138, 0, 0, c, a, b)
```

```
7  #define M_ADDZE(a,b)                  M_OP3(31, 202, 0, 0, b, a, 0)
8  #define M_ADDME(a,b)                  M_OP3(31, 234, 0, 0, b, a, 0)
9  #define M_ISUB(a,b,c)                 M_OP3(31, 40, 0, 0, c, b, a)
10 #define M_ISUBTST(a,b,c)              M_OP3(31, 40, 0, 1, c, b, a)
11 #define M_SUBC(a,b,c)                 M_OP3(31, 8, 0, 0, c, b, a)
12 #define M_SUBIC(a,b,c)                M_OP2_IMM(8, c, b, a)
13 #define M_SUBE(a,b,c)                 M_OP3(31, 136, 0, 0, c, b, a)
14 #define M_SUBZE(a,b)                  M_OP3(31, 200, 0, 0, b, a, 0)
15 #define M_SUBME(a,b)                  M_OP3(31, 232, 0, 0, b, a, 0)
```

Listing 3.2: PowerPC assembler instruction macros for the JIT compiler

The data segment

The data segment is handled as a linked list by the JIT compiler during compilation. It is used to store values at JIT-compile time which may have to be retrieved during runtime of the method such as locks, pointers to data structures as well as method and field references. Additionally placeholders in the data segment may be created that are set at runtime. Each entry in the list specifies a type such as integer, double, address, and a value and links to the next entry. The list can be modified with a number of functions that add a value of a certain size or find the location of a previously added value. That way existing entries may be reused.

A number of data segment entries such as those designating the start of the exception table and a pointer to an informational structure have to exist for each method. They are the first entries to be added to the data segment and thus have a fixed offset. To simplify access to these entries their offset is also given in precompiler definitions.

When adding an entry to the list an offset is returned. This offset points to the location of the entry relative to the code entrypoint during runtime which also marks the beginning of the data segment in the opposite direction of the code. The location of the code entrypoint is always kept in a register called the *procedure vector*, so each value can be accessed by loading the data stored at the sum of the procedure vector and the offset.

Once compilation of the method's code has been finished the JIT compiler iterates over the data segment list and writes out the values in the list according to their storage size to the actual memory location of the data segment. It starts with the first element in the list which is stored right below the procedure vector and moves on towards lower addresses.

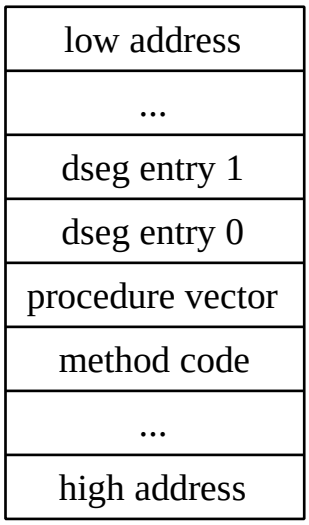


Figure 3.3: Schematic view of the data segment

3.1.3 Garbage collector

Like many other free implementations of object-oriented programming languages Cacao uses the *Boehm-Demers-Weiser garbage collector*(Boehm GC). As this GC has originally been developed as a replacement for the C/C++ memory management functions it may also be used for automatic memory management for other programming languages when the actual implementation is done in C/C++. The Boehm GC implements a *mark-and-sweep* algorithm. This type of GC performs its work in two stages:

1. Starting from the *roots*, in Cacao's case the method stack which may partially reside in registers, follow the chain of pointers and mark every object in this chain as *alive*
2. Free all memory belonging to objects not marked as *alive*

For the purpose of following the pointer chain the Boehm GC considers every bit pattern that may represent a pointer as such. This is called a *conservative GC* as opposed to an *exact GC* that can differentiate between arbitrary bit patterns and actual pointers. The Boehm GC uses a *stop-the-world* mechanism during garbage collection which means that when the GC is started all threads are stopped. While this avoids any concurrency issues it marks a certain performance barrier since no actual code can be executed in the program using the GC while it is running.

At the time of this writing the Cacao team is also working on their own implementation of an exact GC, however it was not ready for productive use.

3.1.4 Lazy resolving and compilation

Since Cacao is using a JIT compiler not all references to the members of a class are resolved at compile time of a method but during runtime. This is called lazy resolving since the resolving only takes place once the member is actually accessed. The mechanism for lazy resolving in Cacao is based on a so-called patcher. Whenever an unresolved member is encountered during compilation a temporary data structure containing only the essential information for resolving that member is created. The instruction trying to access the member is replaced with a stub method. Once this instruction is reached during runtime the inserted stub method is called. The stub method then calls the patcher with the data structure and other required information for resolving the member. Depending on the type of member, such as a static or virtual method or field, a more specialized patcher subfunction is called and the affected references in the method's code and data segment are adjusted. Execution then continues with the resolved member and the original instruction at the point where it previously branched to the patcher stub.

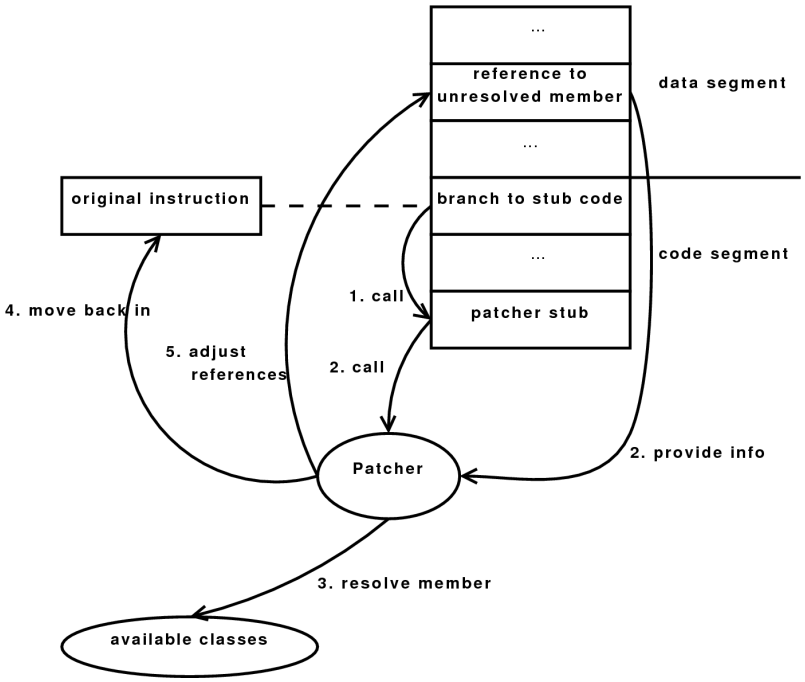


Figure 3.4: Patcher mechanism

Closely related to lazy resolving is also the way the JIT compiler is called. Only when a class is first required during runtime it will be initialized by the lazy resolving mechanism and each method it provides is filled with a stub compiler method. This stub method is only a few instructions long and all it does is call a glue function which in turn calls the JIT compiler. The stub method also contains a small data segment which provides only two entries, the branch

target for the glue function and a reference to a structure which provides all required information for compiling the method. Once the method is called for the first time the stub method is actually executed. After successful compilation the JIT compiler adjusts the references to the method so that subsequent calls will branch to the newly compiled code and finally it calls this code as well.

3.1.5 Method calling

Several different ways for calling a method in Java are available each of which requires a certain calling scheme as well as appropriate handling by the patcher.

Static methods

Static methods are those methods that are shared among every instance of a class and do not belong to a particular object. This means that the code can always be found in a fixed location once it has been compiled. The implementation for this calling scheme in Cacao consists of adding a reference to the entrypoint to the data segment, loading the reference and branching to it. The patcher therefore only has to adjust the data segment entry once the method has been compiled.

Instance methods

Instance methods are object members so they are always associated with a currently existing object. In order to resolve the address of an instance method's code each object in Cacao is associated with a *virtual function table* (vftbl) which contains pointers to all instance methods of an object. Using the address of the object header and the index within the vftbl the code address can be resolved. This requires two memory accesses, dereferencing the address of the vftbl from the object header and then dereferencing the entrypoint from the vftbl. For unresolved methods the index in the vftbl is not known so it is being set correctly by the patcher once required.

Interface methods

A Java class may implement any number of interfaces. An interface defines which methods a class has to implement however does not provide an implementation of its own. Similar to instance methods each object header in Cacao also provides an interface table which contains all interfaces the object's class implements. In order to resolve the code address two offsets are required. The first one is the index in the interface table, the second offset provides the index in the function table of that interface. Resolving the code location and patching interface methods is similar to instance methods only with two instead of one offset.

Builtin methods

A number of methods are implemented in C in the Cacao source code and are therefore called builtin methods. They are mainly used for two cases:

1. Interactions between Java methods and the JVM such as instantiating new objects along with the required memory management
2. Implementing translations for bytecode instructions that require a complex and long sequence of assembler instructions

The addresses to these methods are known at JIT-compile time so the appropriate branch instructions do not have to be resolved at runtime and can be correctly generated right away. No patching is required. Depending on the architecture some calling conventions from the ABI have to observed, for PowerPC however the ABI used by Java methods and builtin methods is the same thus requiring no additional work.

3.1.6 File hierarchy

A short overview of the file hierarchy used by the Cacao sources reveals mainly two interesting points. The JIT compiler source directory contains an individual directory for each architecture it supports which in turn contains another set of directories for each operating system supported on that architecture. While the architecture directories contain the machine-dependent parts of components such as the code generator and the register allocator the OS-specific directories contain for example definitions from the ABI which may differ from OS to OS even on the same architecture.

3.2 Design decisions and steps for the port

As a conclusion from the overview of the Cacao architecture the port of Cacao to Cell/B.E. will be based on the following design decisions and steps. The individual steps will be briefly discussed along with potential issues that will have to be addressed.

3.2.1 Porting the JIT compiler to emit SPU code

In order to execute any methods at all on the SPU the code generator for the JIT compiler will have to be ported to support the SPU instruction set. The PowerPC JIT compiler can be taken as a base since the instruction sets share a certain similarity. They are both RISC-based and use only a small number of different formats for the assembler instructions. Special care will have to be taken with the 16-Byte aligned memory accesses of the SPU's LS as well as efficiency of the code which should be low on branches and make use of the SPU's unique instructions.

3.2.2 Shared heap access

Due to the multi-threaded nature of both Java and the Cell/B.E. a solution will have to be developed for shared access of objects on the heap from all PPU and SPU threads. This may include the SPU accessing objects in the main memory, maintaining a local heap in the LS and creating objects there as well as transferring entire objects between the different memories and caching them for faster access. However, as Cacao implements the multi-threading capabilities of Java some of these problems such as concurrent thread-safe access to objects have already been solved when the Cell/B.E. is regarded as a shared-memory system. Still, in order to achieve the desired functionality many changes to the Cacao architecture will be required as support for multiple heaps has never been regarded in Cacao before.

Along with shared heap access also comes the problem of extending the GC. Depending on the approach for shared heap access objects or object references may reside on both the PPU and on the SPE including its registers and LS. Since the SPU's LS can be mapped into the PPU's address space a GC running on the PPU can also access and clean a heap on the SPU at a slightly decreased performance. It can also include object pointers on the SPU in its calculations of whether an object is still alive.

3.2.3 Swap code in and out of the LS

As the LS is very limited in size it will most likely be required to keep only those compiled methods in it that are needed currently or soon. The compiled code of the other methods will be cached in and made available from the main memory. A method to swap code in and out of the LS as required will have to be developed. A starting point could be provided by the SPU Software Managed Cache Library [IBM07b] that is included in the Cell SDK 3.0. This library allows the SPU to use parts of the LS as a cache for the main memory so that the developer does not have to manually take care of the synchronization and transfer of the data. Additionally a method's code may be too large for the entire available space in the LS requiring to split up the code into multiple segments or not being able to execute this method on the SPU.

3.2.4 Selective execution on the SPU

Due to the SPU's highly-specialized capabilities only those methods that can make use of its functionality should be executed on it while most other methods should be run on the PPU. In fact this applies for most of the methods from the class library, especially parts such as the classloader and I/O functionality. That way the PPU would mostly service requests from the SPUs that they cannot complete themselves. In order to achieve this separation it will be required to mark methods for execution on the SPU. This could be done for example by giving the method or class a special name or package or tag it with an annotation.

3.2.5 Communication and branching between SPUs and PPU

As only certain methods will be executed on the SPU it will be necessary to develop a concept which allows branching from an SPU to a PPU method and vice versa. This requires passing arguments between the methods which in turn means moving them between main memory and LS. Additional facilities to support the multi-threading capabilities of Java including thread coordination and shared synchronized object access have to be provided. Since most of these functions are realized by using mutexes in main memory they can be handled by atomic DMA transfers.

4 Implementation variations

In order to fulfill the design decisions a number of different tasks have to be performed for each step. A more thorough discussion of these steps along with possible issues and a number of feasible solutions will be given in this chapter. Not all solutions presented have actually been realized in the prototype implementation. An overview of the implemented parts will be given in the chapter 5 along with a review of what the prototype is capable of.

4.1 Build process and additional files

Besides changes to existing files or using existing files as a base a number of new files had to be added in order to access the SPUs. Additionally the build process had to be altered to include the newly added files.

4.1.1 Cell port

As a starting point in order to maintain the sources in a state that can be compiled the PowerPC port of the JIT compiler was simply duplicated and moved to a directory called `cell/`. Any references to files from the `powerpc/` directory had to be adjusted to point to `cell/` instead. Furthermore a number of files in this port which will be explained in greater detail in section 4.2 were also duplicated to provide one copy for the PPU and one for the SPU.

4.1.2 C-code entrypoint for the SPU

In order to facilitate use of the SPUs a central entrypoint for them is developed in C from which the first Java method on the SPU will be called. While the SPUs could be loaded with the

first Java method they have to execute and then be set to run, this intermediate step greatly eases development. The necessary steps for setting up and starting the SPU can be handled by libspe2, all required C functions including builtin functions are automatically loaded to the correct address in the LS and debugging statements such as printing out values can be easily inserted.

The exact flow for starting up the SPU as shown in figure 4.1 begins with the PPU transferring the SPU binary to the SPU and then starting up the SPU. This binary includes all necessary C and assembler functions. This is performed by first calling `spe_program_load()` with the pointer to the SPU binary that is linked in with the PPU binary and then calling `spe_context_run()` which starts the SPU. `spe_context_run()` accepts a number of arguments which are passed on to the `main()` function of the SPU binary. The `main()` function initializes the software cache for storing code and then waits for a number of mailbox messages from the PPU. The PPU currently sends these messages once it is ready to execute the `main()` Java method so that this method is executed on the SPU. The individual messages contain the starting address of the data segment in main memory, the combined size of the data segment and the code and the offset to the beginning of the code. Once the SPU has received all three parameters it retrieves the code and starts to execute the Java method. More details on this process will be given in later sections.

Linked in with the C part is also an assembler part. It currently contains only a small number of functions used to branch between Java methods, explained in more detail in section 4.6.2.

4.1.3 Build process

Cacao is using the *GNU Autotools* for its build process. This set of tools from the GNU project allows the automatic creation of Makefiles for large projects by defining only a relatively small number of build options and dependencies. Additionally it supports a portable build mechanism across multiple platforms. This includes automatic configuration of the build process which checks for the available tools and libraries as well as using the correct platform-specific mechanism for dynamic linking.

Unfortunately the Autotools currently do not support the Cell/B.E. toolchain which in fact uses two separate toolchains, one for the PPU and one for the SPU. The compiler to be used can be passed to the configuration script however this compiler will be used for all files. For that reason the commands to build, link and embed the SPU specific parts as described in section 1.1.5 were hardcoded in the specific Makefiles.

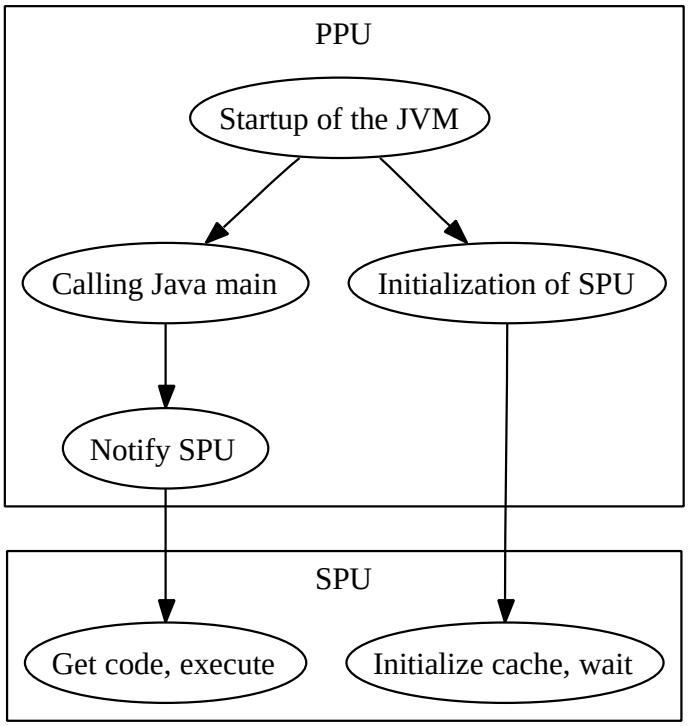


Figure 4.1: Startup of the SPU

Build script

In order to support building Java on Cell/B.E. on other systems on which the modified sources are not available a build script is provided. This build script is packaged with a number of patches and the unmodified Cacao sources. When it is run it automatically unpacks the Cacao sources, copies the PowerPC port to the new Cell/B.E. port, applies the patches which contain all modifications performed for this project and starts the automatic configuration for the build process. The user may then build and run Java on Cell/B.E.

4.2 Porting the JIT compiler to emit SPU code

The majority of work with porting the JIT compiler was split between two of its components, the register allocator and the code generator as these two parts are the most machine-dependent ones. One of the biggest and most pervasive issues with porting the JIT compiler comes from the fact that it is expected to compile code for only one type of processor and thus expects to use

the same register set and instruction set for all methods. Based on this reasonable assumption precompiler definitions are used in Cacao to conform to the ABI and other fixed items including the number and size of registers available, dedicated registers like the stack pointer, pointer size and stack slot size. As these definitions are handled at compile time this still allows for the flexibility to build the JVM to generate code for different architectures however once it has been compiled it supports only one type of architecture. Since extending Cacao with support for Cell/B.E. required the ability to JIT-compile for two different types of processors a new approach for this problem was needed.

Three possible solutions to this issue seemed realistic and have also been discussed with the Cacao development team:

1. Duplicate the precompiler definitions and machine-dependent functions for both the PPU and the SPU, call the correct function for the target processor type of the currently compiling Java method

2. Replace the precompiler definitions with a conditional statement that returns the correct value depending on the target processor type

3. Define a structure containing the machine dependent parameters and pass the correct instance to the compiler stages thus adding an intermediate target-specific layer

Item two could quickly be ruled out since precompiler definitions should define exactly one value. Item three seemed most promising since it provides great flexibility with little redundancy however would require a major code rework. This is also the solution chosen by other projects such as GDB that have also encountered the problem of precompiler definitions for architecture-dependent parameters. Due to time constraints the first solution was chosen since it requires the least effort however at the cost of redundant code.

As explained above the main parts concerned by this change are the register allocator and the code generator. However, as for the code generator it has to be duplicated and adapted for the SPU anyway due to major changes from its PowerPC base. Therefore it will not contain much redundant code in the end. The implemented changes for these two components are described in the following sections. As porting the code generator included a number of separate subtasks their descriptions have also been split into additional subsections.

4.2.1 Porting the register allocator

The precompiler definitions for the register allocator mostly consist of values from the ABI as shown in 4.1. As the SPU ABI [IBM07e] provides 128 128-bit wide unified registers compared to the 32 integer and 32 floating-point registers provided by the PowerPC a modification of the register allocator definitions for the SPU was required. Since the different register allocator functions simply use the existing definitions both the functions and the definitions had to be duplicated. With a duplicate set of functions and definitions the PPU functions use the PPU definitions while the SPU functions use the SPU definitions. Depending on what processor type a method is JIT-compiled for the respective version of a function can then be called. To differentiate from the PPU versions the SPU versions of the definitions and functions are simply suffixed with `_SPU`.

```
1  /* integer registers */
2
3  #define REG_RESULT        3   /* to deliver method results                    */
4  #define REG_RESULT2       4   /* to deliver long method results               */
5
6  #define REG_PV           13   /* procedure vector, must be provided by caller */
7  #define REG_METHODPTR    12   /* pointer to the place from where the procedure */
8                                /* vector has been fetched                      */
9  #define REG_ITMP1        11   /* temporary register                           */
10
11 /* ... */
12
13 #define REG_SP            1   /* stack pointer                                */
14 #define REG_ZERO          0   /* almost always zero: only in address calc.    */
15
16 #define REG_A0            3   /* define some argument registers               */
17 #define REG_A1            4
18
19 /* ... */
20
21 #define INT_REG_CNT      32   /* number of integer registers                  */
22 #define INT_SAV_CNT      10   /* number of int callee saved registers         */
23 #define INT_ARG_CNT       8   /* number of int argument registers             */
24 #define INT_TMP_CNT       7   /* number of integer temporary registers        */
25 #define INT_RES_CNT       7   /* number of integer reserved registers         */
26
27 /* ... */
28
29 /* ABI defines ****************************************************************/
30
31 #define LA_SIZE              8  /* linkage area size                           */
32 #define LA_SIZE_ALIGNED     16  /* linkage area size aligned to 16-byte        */
33 #define LA_SIZE_IN_POINTERS LA_SIZE / SIZEOF_VOID_P
34
35 #define LA_LR_OFFSET         4  /* link register offset in linkage area        */
```

Listing 4.1: PowerPC ABI definition for the JIT compiler

Due to the first-come-first-serve approach of the register allocator it should profit from the vast number of available SPU registers as values will only rarely have to be swapped out from the registers to the stack.

4.2.2 Porting the code generator

As layed out in the introduction to this section the required steps for porting the code generator can be split into a number of relatively disconnected subtasks which will be discussed in the following.

Supporting bytecode translations for the SPU

Supporting the correct bytecode translations for the SPU is mostly a manual and repetitive task that involves going over all Java bytecode instructions and finding the correct set of assembler instructions to perform this operation on the SPU. Three different parts have to be implemented for this as explained in section 3.1.2:

1. The macros to support the different assembler instruction formats
2. The macros that map the actual assembler instruction to the right instruction format
3. Appropriate machine code translations for the bytecode instructions

Using the SPU Instruction Set Architecture (ISA) [IBM07f] step one was easy to perform especially since the SPU instruction formats are very similar to those used on the PowerPC. This way the PowerPC binary output macros as presented in listing 3.1 could be used as a base with only few modifications.

Step two could be greatly simplified and automated. Using the tool `pdftotext` the SPU ISA could be converted to a flat text file and subsequently parsed using a Perl script which is reprinted in 6.1. This script correctly outputs most of the required assembler macros except for some special cases with unusual syntax such as branch hints which have to be added or corrected manually.

As for the third step, translating the bytecode instruction to assembler instructions, the easiest way to find out an appropriate translation was to look at how the SPU-GCC translates an instruction. If applicable a short `C`-program that performs the `C`-pendant of the bytecode instruc-

tion in question was written and compiled and the resulting assembler code examined using `spu-objdump`. This tool is used to disassemble a binary file and print out the raw contents of the file along with the assembler instructions encoded. The relevant assembler instructions could then be incorporated into the code generator by inserting the corresponding assembler macros.

This approach is especially useful for instructions for which no equivalent single or small number of assembler instruction exists. An example for this is division[1]. For these the compiler has to generate a rather long stream of assembler instructions which would be hard to figure out from the ISA alone.

Builtin functions

As explained in section 3.1.5 builtin functions may be used to implement bytecode instructions that are too complex to realize in the code generator due to the long sequence of assembler instructions their translation requires. A good example for this on the SPU is the aforementioned division especially with 64-bit values. In fact according to the SPU-GCC the instruction sequence for double-precision floating-point division consists of 175 instructions not counting subfunctions called in the process.

Calling builtin functions comes with the overhead of branching from Java code to C code. This includes tasks such as setting up the stack frame according to the ABI and copying the operands to the argument registers while the called function might have to save additional registers in order to restore them when returning. These tasks would not be required if the appropriate instructions were generated directly by the code generator. However, especially in the case of the discussed 64-bit instructions using builtin functions greatly eases porting the code generator.

In order to efficiently use builtin functions on the SPU it has to be ensured that they are directly available in the SPU's LS. This can be done by appropriately linking them to the SPU binary that provides the entrypoint for Cacao on the SPU.

IEEE 754 compliance

Since Java 2 the keyword `strictfp` has been added to the language specification as discussed in [LY99] chapter 2.18, *FP-strict Expressions.* It can be applied to classes, interfaces and methods.

[1] On the assembler level division is performed by multiplying with the reciprocal value

When encountering this keyword the JVM is forced to perform all floating-point calculations in this part under strict IEEE 754 conformance meaning that the resulting values must match exactly those expected by following the IEEE 754 standard.

The IEEE 754 standard [IEE85] defines how computer architectures have to handle floating-point calculations. This includes the different precisions available, 32 bit for single-precision and 64 bit for double-precision, rounding, handling of special values such as infinity and not-a-number (NaN) and the binary representations of ordinary and special values. It also defines how and when floating-point exceptions are thrown when some of these values and special cases such as underflow are encountered during a calculation. In addition to the standard-precision formats IEEE 754 also describes optional extended-precision formats which define the minimum width of data types with 43 bit for single- and 79 bit for double-precision. The exact size is implementation-dependent.

When the `strictfp` keyword is omitted the JVM may use whatever floating-point support the current architecture provides. In many cases such as for x86 this includes the extended-precision formats. Since these formats are not supported by all architectures and vary between implementations omitting the keyword may result in the same program yielding different results on different architectures. Regardless of the keyword however, catching floating-point exceptions is not supported by the Java language.

Since all compliant JVMs have to implement the `strictfp` keyword this is also true for Cacao. While both the PPU and the SPU support floating-point numbers with 32 bits for single- and 64 bits for double-precision only the PPU fully conforms to the IEEE 754 standard. As explained in [IBM07a] chapter 3.1.4, *Floating-Point support* on the SPU the range of normalized numbers is extended compared to IEEE 754 at the expense of certain special cases that cannot be represented according to the standard. When a calculation yields a result not conforming to the standard a flag in a special floating-point status register called *DIFF* is set indicating a deviation.

Using this flag implementing the IEEE 754 standard as required by Java is possible. Cacao has to be extended so that after every `strictfp` calculation on the SPU the DIFF flag is checked. When a non-conforming result is encountered two ways to rectify this are possible.

1. The calculation is repeated on the PPU which supports the full IEEE 754 standard. The easiest way to do this is by issuing a stop-and-signal notification and pass the operands and the operator to the PPU which performs the calculations and returns the accurate result.

2. The correct IEEE 754 floating-point semantics are emulated in software on the SPU. This requires developing builtin functions that manually work on the bit patterns provided by the operands according to the standard.

While both these workarounds will induce a performance hit non-IEEE 754 compliant results should only occur for rare corner cases and are only relevant with `strictfp` calculations making this workaround tolerable.

Enabling 16-byte stack and data segment slots

As discussed in section 1.1.3 the SPU can only load and store an entire quadword at a 16-byte aligned address at a time. A number of methods to account for this fact in relation to the stack and the data segment will be discussed here.

The SPU instruction set provides special commands to work around the 16-byte alignment restrictions. For storing data to unaligned addresses a special bitmask can be generated based on the offset of the target address to the next lower 16-byte boundary by the family of *Generate Controls for* {*Half,Double*}*Word Insertion* instructions. The only requirement is natural alignment of the address meaning that for example a 4-byte value must be stored to a 4-byte address. The data to be stored is loaded to the preferred slot of one register while the 16-byte aligned quadword surrounding the target address is loaded into another register. The *Shuffle Bytes* instruction then picks values from one of the two data registers according to the bitmask and places them in the appropriate slot of the target register. The target register's contents can then be written back to memory. For reading data the surrounding quadword is also loaded entirely into a register. A shift count is calculated based on the 16-byte offset of the actually requested data within the quadword. Using this count the contents of the register are then shifted to the left so that the requested data is moved to the preferred slot for the data type. This mechanism is shown in figure 4.2.

Another approach is to simply align all data to 16-byte borders regardless of their actual data size and store them within this quadword the same way they are also stored in a register. This means that they can be immediately stored and loaded back into a register without any further operation. This is the solution used by the SPU-GCC for the stack as well as for the data segment and also documented in the SPU ABI which specifies that the stack must always maintain 16-byte alignment. As the stack contains many scalar values such as the stack pointer and link register this means that memory is wasted with the benefit of simpler operation on the stack and the data segment values. In order to maintain compatibility with the native SPU ABI and keep

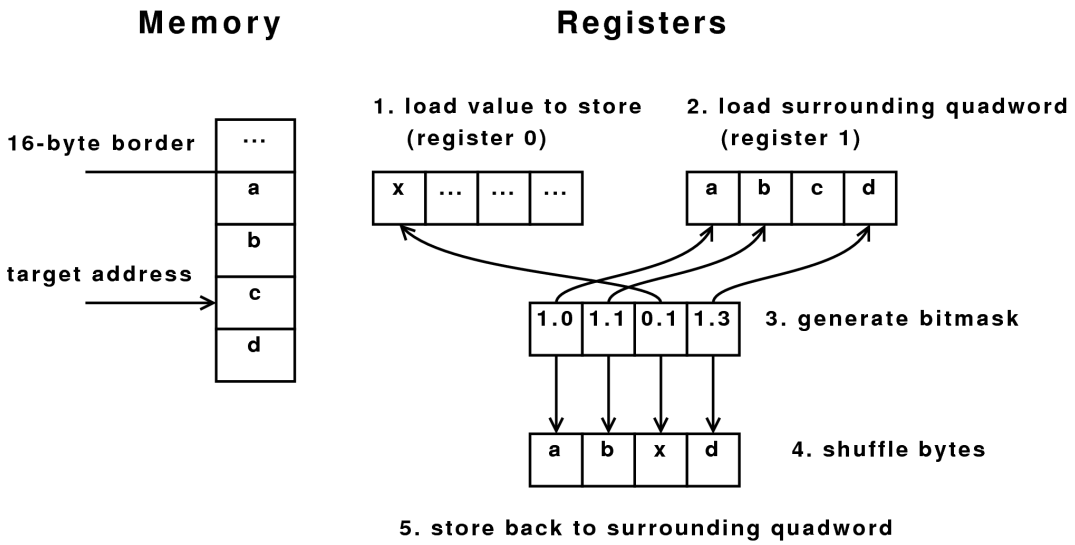


Figure 4.2: Simplified mechanism for unaligned storing

the code efficient and simple this solution will be further pursued.

Implementing the 16-byte alignment requires a number of modifications. As the stack is controlled entirely by the code generator part of the SPU the necessary changes were fairly localized. They simply consisted of creating a stack frame of sufficient size so that for every stack slot 16 bytes were allocated. The data in the stack such as the stack register, link register and the registers stored by the calling method are then offset from the stack pointer by a multiple of 16 according to the stack layout from the SPU ABI.

Adapting the data segment to 16-byte alignment was more complicated. In order to simplify the process both the data segment generation for PPU as well as for SPU methods was modified to observe 16-byte alignment even though this is not a requirement for the PPU. However differentiating between the PPU and SPU data segments would have required an additional number of conditionals. Based on the introduction of the data segment in section 3.1.2 the actual steps performed were:

1. adjusting the fixed offsets for the method-specific properties to the next 16-byte aligned number
2. aligning the current data segment lengths to the next multiple of 16 when adding a new element
3. fixing some offsets in the stub compiler method which works with fixed offsets

Once the correct locations to modify were identified the changes were fairly easy. However,

especially for the stub compiler method finding the required changes was non-trivial as it uses numerical values to describe the offsets instead of symbolic ones.

Accessing SPU symbols in the PPU code generator

For some functions it might be necessary for the code generator to emit a branch to an SPU function that is implemented in the C-part. This includes internal helper functions as well as builtin functions. As the code generator runs on the PPU it does not have direct access to the SPU functions or more generally symbols included in the SPU binary which are required to calculate the branch target to call such a function. As the SPU binary that is linked in with the PPU binary is also an ordinary ELF file it is possible to read in the required symbols manually. For this purpose the symbol designating the SPU part which is of the type `spe_program_handle_t` provides a pointer to its ELF structure which contains all the required information. The exact implementation of this is structure is reprocuded in 4.2.

```
1  /** SPE program handle
2   * Structure spe_program_handle per CESOF specification
3   * libspe2 applications usually only keep a pointer
4   * to the program handle and do not use the structure
5   * directly.
6   */
7  typedef struct spe_program_handle {
8    /*
9     * handle_size allows for future extensions of the spe_program_handle
10    * struct by new fields, without breaking compatibility with existing users.
11    * Users of the new field would check whether the size is large enough.
12    */
13   unsigned int  handle_size;
14   void    *elf_image;
15   void    *toe_shadow;
16 } spe_program_handle_t;
```

Listing 4.2: Implementation of the SPE program handle

A starting point for the code is taken from the libspe2 sources, specifically the included ELF loader. Once it has loaded the required files into memory the pointer `*elf_image` provides a reference to the runtime representation of the SPU binary. As an ELF binary consists of a number of sections which may include symbols the loader has to parse all sections included in the ELF image looking for a `.toe` section. Once it has found this section it iterates over all symbols looking for a few specific ones. Using this code as a base writing a function that looks for a given symbol in all sections was fairly easy. Associated with the symbol is the offset at which it will later be found in the SPU's LS. This offset can then be used with the *branch absolute* instruction which branches to a fixed address.

In its current implementation the code goes through the entire SPU binary every time an SPU symbol is requested. As an improvement the symbol could be loaded once and then be saved in a table allowing for a faster lookup on subsequent runs.

4.3 Shared heap access

Apart from essential steps such as porting the code generator to the SPU an efficient access to shared data on the heap is crucial to the performance of the entire system. Unfortunately providing a transparent solution, one that does not require special programming constructs or mandatory modifications of the source code, has not been given any effort by the Cacao team so far. This is understandable as the concept of Java on Cell/B.E. presents a novel approach.

As discussed in section 3.2.2 shared heap access requires a number of unforeseen and therefore large modifications to Cacao. For now the heap and therefore all objects reside in only one memory and may be directly accessed the same way by all threads which share the same memory space. With the SPUs and the PPU using different memories a mechanism will have to be developed to access single members or entire objects efficiently from both processor types and across all memories.

In order to understand the challenges this step poses it is necessary to understand the multi-threading support provided by Java and the consistency guarantees it makes. This has been discussed in section 1.2.3. A more formal overview of the Java memory model is given in [GJS05] chapter 17.4, *Memory model.* It defines how different actions such as reads, writes and thread events may be reordered by the JVM and which storage they must affect at which point. Based on these bounds a number of concepts to support shared heap access along with their advantages and downsides will be presented in this section.

4.3.1 Individual non-cached access of fields

In the solution used for the prototype implemented as part of this thesis the heap and all objects will only be kept in main memory. Object fields, the data members of a class, will be accessed individually via DMA transfers. The steps required to implement this solution will be described in this section.

The major disadvantage with this simple solution lies in the fact that every heap access requires

at least one DMA transfer and thus will pose a major performance barrier. Since fields are at most eight byte large the latency of DMA transfers will greatly outweigh the possible bandwidth. Additionally the GC requires an extension. Due to the fields being transferred to the SPU references to objects in the main memory may be stored in the SPU stack and in the SPU registers in which the stack may partially reside. The GC would have to include these in its calculations about whether an object is still alive. In order to not cope with this issue at this point the GC will be turned off. This is supported by Cacao as a compilation switch and results in heap space not being freed. For small testcases this is acceptable.

This first simple solution also has to take the alignment restrictions of the MFC into account, mainly the fact that the 16-byte offset for both the target and the source address must match. Therefore a 16-byte entry is added to the data segment of each SPU method which serves as a temporary storage for DMA transfers[2]. Due to its size every 16-byte offset of a field in main memory can be matched in the LS. As the alignment of the field in main memory is not known at JIT-compile-time the exact address within the temporary storage to or from which it will be transferred has to be calculated at runtime so both 16-byte offsets match.

The required code sequence for storing a field can be found at 4.3 while a graphical explanation is given in figure 4.3. The actual steps required are:

1. Load the value to be stored into a register
2. Shuffle the value to the correct slot according to the offset of the target address
3. Store the register contents to the temporary storage
4. Calculate the matching source address according to the offset of the target address
5. Perform the transfer

For load operations this sequence is mostly reversed.

```
1  /* sample code for storing a static field */
2  /* s1 contains the value to be store, REG_ITMP3_SPU the target address */
3
4  /* generate the bitmask for moving the actual value to the right slot according to the ↩
       target
5  address */
6  M_CWD(0, REG_ITMP1_SPU, REG_ITMP3_SPU);
7  /* use the bitmask to shuffle the register accordingly */
8  M_SHUFB(s1, s1, s1, REG_ITMP3_SPU);
9  /* store the register contents to the temporary storage */
10 M_IST(s1, REG_PV_SPU, MfcTemp);
11
```

[2]This is not thread-safe as the data segment is shared across all simultaneous executions of a method. However executing several threads on one SPU is not possible so this issue is negligible

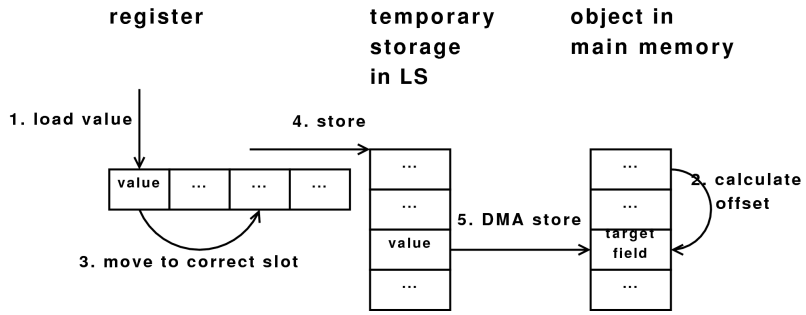


Figure 4.3: DMA access to individual object members, storing a field

```
12 /* calulate the 16-byte offset of the target address by ANDing it with 0xF */
13 M_ANDI(0xF, REG_ITMP1_SPU, REG_ITMP3_SPU);
14 /* add the offset to the procedure vector */
15 M_A(REG_ITMP3_SPU, REG_PV_SPU, REG_ITMP3_SPU);
16 /* add the offset of the temporary storage, this gives the LS target address */
17 M_AI(MfcTemp, REG_ITMP3_SPU, REG_ITMP3_SPU);
18
19 /* emit the correct code sequence for storing the value in the main memory */
20 emit_mfc_spu(jd, MFC_PUT_CMD, REG_ITMP3_SPU, REG_ITMP1_SPU, 4, 5);
```

Listing 4.3: Code sequence required to store an unaligned field via DMA

Since the method to get or set a value will be executed on the processor where the object resides ie. on the PPU all threads regardless of where they are being executed will access the same monitor. Therefore synchronized access can be realized as required by the standard provided the Java program is thread-safe. Accessing the monitor from the SPU can be done by using atomic DMA transfers since the monitor is just an address in memory. Using atomic DMA transfers ensures that no other thread may modify the monitor while it is being locked.

4.3.2 Sharing objects between processors

One efficient and transparent solution is based on the fact that it is the Java programmer's responsibility to ensure thread-safety. This means in turn that non-thread-safe parts may be executed in an incoherent manner while still being compliant to the Java specification. The solution introduced in this section exploits this concept accessing only those fields in main memory that are used in synchronized statements while locally caching the other fields. The individual steps required to implement this solution will be detailed in additional subsections.

One of the main problems with efficient shared and thread-safe access lies in the fact that Java does not support locking individual members of objects and thread-safety being the responsi-

bility of the Java programmer. Even when all fields of an object are private and it provides synchronized getters and setters there is no guarantee that some other method of the object does not modify these fields directly. Therefore the JVM can only make assumptions about which members are actually accessible directly or indirectly by only one thread at a time. However, in order to achieve thread-safety for shared objects the programmer must ensure that this object is accessed only from within a properly synchronized block.

Access objects locally when possible

One approach to achieve efficient object access is to copy the entire object that the SPU will work on to the LS. This must also include the object's methods like the getters and setters so that they can be executed on the SPU. Otherwise the benefit of having local access to the object's fields would be negated by having to execute the access methods on the PPU. As long as the object includes non-synchronized methods these may be executed directly on the SPU working with the available local copy without observing coherence with the original object in the main memory. This is possible since the semantics of Java do not guarantee a consistent state in this case. As for synchronized methods, these may be executed on the SPU if locally available however all operations on object members within a synchronized block or method must be executed on the original object so as to ensure consistency.

Being able to execute an object's methods locally also requires compiling them for both the PPU and the SPU. In principle this is not a problem as each methodinfo structure points to a codeinfo structure which in turn contains the important information about the code such as its location and size. The methodinfo could thus simply be extended to contain one codeinfo structure for both processor types. When the respective codeinfo for a processor type is a NULL pointer this indicates that the method must be compiled.

Track object references

In order for this method to work it is required to track the object references when they are passed between methods. This is possible since the Java language differentiates between actual object references and primitive types featuring the same bit pattern. Additionally, casting a primitive type to an object is not directly supported by Java and requires the creation of a new object. This way the JVM can then at first create new objects in the memory space of the creating method. Once the object reference is passed to another method the object may have to be moved to the

main memory for shared access or can be kept in its current memory if the method is executed on the same processor.

As an optimization the JVM could intercept the creation of new objects to which no local reference is kept when they are being passed to a method on a different processor. Since the calling method does not store a reference to the object it is no longer accessible from that processor meaning that it can be moved without any conflicting access issues. An example for this is passing a copy of an object using `clone()`, which duplicates an entire object, directly in the argument list of the method. Additionally the JVM may generate the required instructions for moving the object before the method is actually called. While this must be done with care as values in the object may still change it could hide the latency of the required DMA transfer.

Transfer scattered objects and references

One issue with moving objects between different memories is the fact that objects often reference other objects by internally, on the JVM level, using pointers. Due to this mechanism an object with all its further referenced objects may be fragmented across memory. This is making it hard to efficiently use DMA transfers to move the object to another memory since DMA transfers usually operate on contiguous memory areas. Two remedies for this problem may be used:

1. The SPU requests an address list for the object in question by notifying the PPU with a mailbox message or a stop-and-signal call and passing the object reference. The PPU then resolves the chain of objects referenced by the given object, possibly only up to a certain depth, and transmits the generated address list to the SPU. Based on this address list the SPU then initiates an MFC list transfer which copies the entire object and its references to the LS in one step. After adjusting the object references to the LS addresses the object can then be used.
2. The SPU only transfers the first level of an object of which both the address as well as the size is known. This includes only primitive types, object references will not be resolved but their values transmitted. As an additional improvement the program flow could be analyzed as to whether the referenced objects are actually needed. If they are, they can be transferred, possibly some time before they are actually needed in order to hide the latency of the DMA transfer. The object's methods will also only be transferred once they are actually required as resolving their code location and size requires a number of indirections.

Conclusion

This solution will almost certainly produce inconsistent states when not enough care is taken. Due to the latencies of the DMA transfers and the disjoint memories of the processors the chance for conflicting accesses to fields is higher than with a traditional Java multi-threading implementation on a shared-memory architecture. However, the Java specification permits these inconsistencies if they do not violate the Java memory model. While great care has to be taken to operate within these constraints the discussed solution provides for great flexibility. Without any further modifications multiple threads may each work without the latency associated with DMA transfers on the uncritical parts of their local copy of an object. Critical parts may be protected as usual by synchronized statements resulting in thread-safe execution at the cost of reduced performance. Due to multiple threads competing for a lock and having to wait at a certain point however synchronized statements in Java always come with a certain performance hit. While this impact has been significantly reduced over the last few years it still is noticeable so a slowdown coming from this solution is acceptable.

As with the first solution presented object references may also be stored in the SPU stack and in the SPU registers. The GC will have to include these references in its calculations. As the GC represents a completely different and complex field yet may conveniently be turned off during development extending the GC in this fashion is out of the scope of this thesis. However, a principle idea how to realize a solution to these challenges is presented in section 6.3.

4.3.3 Read-only copies on the SPU

Another solution includes a few simplifications that are however not transparent to the Java programmer. If objects passed to methods executed on the SPU are seen as read-only copies of objects in the main heap this would allow the GC to ignore the SPU stack and registers in its calculations. The objects in the SPU heap could simply be discarded without writing them back to the main memory once the SPU method has finished. If the results of methods executed on the SPU are to be kept they have to be explicitly written back. This simplification is especially rewarding since access to the SPU registers from the PPU as required by a GC for Java on Cell/B.E. is very expensive performance-wise. Accessing the registers requires the kernel to load a small program to the SPU that writes out the contents of the registers to main memory. This is almost as expensive as an entire context-switch which occurs when a new program is moved to a used SPU.

4.4 Swap code in and out of the LS

In order to provide fast execution of methods on the SPU it is desirable to keep frequently used methods and those that may be executed soon in the LS. As the LS provides only limited space a mechanism providing a cache of methods has to be developed. After introducing the solution chosen by the SPU-GCC an approach to this problem for Java on Cell/B.E. will be discussed in this section.

4.4.1 SPU-GCC overlays

The SPU-GCC contains a mechanism that can be used to split SPU programs if they are too large to fit entirely into the LS. The different parts will then automatically be transferred to the LS or cached as required. A detailed introduction to this mechanism is given in [IBM07c] chapter 4, *SPU code overlay*.

In the approach used for the SPU-GCC code overlays single function represents the smallest unit that can be cached. Functions can be combined into segments which are always transferred entirely to the LS. Multiple segments form a region of which exclusively one segment is kept in the LS at a time. So if another segment is needed the currently cached segment in this region if any is overwritten. When branching between segments the branch is actually redirected to a stub function by the compiler. This stub function for which one exists for each real function fetches the target segment from main memory if necessary and then jumps to the real branch target. As the runtime location of all functions is known at compile-time all these branches can be generated at compile-time even though the target function may not actually be in the LS at the time of the branch.

When using the SPU-GCC the distribution of functions into segments and regions has to be defined manually in a linker script. Apart from preparing the binary for overlays the linker also automatically inserts the stub and other required functions and tables to maintain an overview of currently cached segments. This manual creation of a linker script allows for a reduction of swap operations when careful manual analysis of the calling dependencies between functions is performed. Ideally the calling and the called function reside in different regions so the containing segments do not have to be overwritten when calling the method and again when returning. With this approach the developer has to manually ensure that the linker script always matches the actual program.

The SPU-GCC overlay support imposes two size restrictions: Under ideal circumstances concerning segmentation and runtime data the total possible size of the entire SPU program is 512 MiB and a single function cannot be larger than the entire LS as cannot be further segmented.

4.4.2 Caching Java methods

As the prototype built for this project uses the SPU Software Managed Cache library this library and its usage will first be introduced in this section. Afterwards its implementation in the prototype will be explained.

The SPU Software Managed Cache library

The SPU Software Managed Cache Library [IBM07b] is commonly used to cache frequently required data in the LS. As the cache is managed entirely by software it can be customized for the needs of the program that uses it. This includes for example the size of the cache lines, the number of sets, the associativity of the cache and whether it provides read only access or read and write. Including the cache in a program requires setting these parameters and other using precompiler definitions and then including a header file provided by the library. With these few prerequisites the compiler automatically reserves the required space for the cache and provides the methods to access it. The precompiler definitions can be re-set and the header file re-included several times which allows the creation of multiple caches for different needs. The flexibility of a software managed cache however comes at the price that it is not coherent in respect to changes in main memory. When a value cached in the LS is changed in main memory this change will not necessarily be visible in the cache.

The SPU Software Managed Cache library provides different interfaces, an external and an internal one. The external or safe interface acts only on the values included in the cache, not the way they are stored. This means that a read from the cache using `cache_rd(...)`given an address in main memory returns the value at that address. When the value at this address has already been cached it will be returned from the cache otherwise it will be transferred to the LS. Similarly the write command `cache_wr(...)` writes out a value to a given main memory address. The internal or unsafe interface allows direct access of the cache lines. When the program notifies the cache that it wants to read or write an address in main memory by calling `cache_rw(...)` an LS pointer to the cache line containing the data is returned once it is available. As the data in that line may be cast out between accesses it must be locked and after-

wards unlocked using `cache_lock(...)` and `cache_unlock(...)` respectively. Additionally the internal interface provides asynchronous access. For this case two commands are used. `cache_touch(...)` which makes the cache prefetch the requested main memory contents and `cache_wait()` which waits until this data is actually available.

Implementation in the prototype

As explained the SPU Software Managed Cache library is commonly used to cache data in the LS. However, as the machine code translation for the Java methods is generated at runtime it can be regarded as such. Therefore this library provides a suitable tool to cache frequently required functions in the LS.

The two main benefits it provides are:

1. No additional code to setup and control DMA transfers is required as the library provides all necessary functions
2. 4-way set associativity of the cache ensures a low risk of cache-thrashing even for calling chains that occupy the same set

N-way set associativity determines in which cache lines or set the data for a given main memory address may be stored. The possible cache lines are usually determined by calculating a hash value of the target address. The higher the *N* the more possible cache lines the data may be stored in and the lower the risk that certain access patterns accessing data from addresses with the same hash value constantly cast each other out of the cache. Applied to this project this would be the case with a calling chain of methods that occupy the same set. With 4-way set associativity 4 different methods of the same set can be called before the first one has to be cast out of the cache.

A survey on the distribution of the size of compiled methods has been conducted for *ECJ*, the Java compiler used by the Eclipse platform, itself a fairly complex Java program. It has shown that the vast majority of Java methods is small in size, in fact around 95% of all methods fall below the 4 KiB border. This result is expected as Java with its object-oriented approach encourages the use of many small methods that perform only a limited set of operations on an object. 4 KiB for a cache size line also marks the arbitrary limit the cache library supports. For most data caching needs this size will likely suffice however some methods may exceed this limit as shown. The limitation can be circumvented by modifying the header files for the library and use a private copy. While this is not a clean solution it works well enough for a prototype.

However, with the solution for partitioning a method into fixed-size blocks presented in the next section this may not be necessary. Alternatively, it could be decided that methods that are too large are instead executed on the PPU.

Including the prerequisites for the cache can be conveniently done in the C-part of the SPU binary so that the SPU-GCC can set up all required functions and memory areas. The exact semantics for calling the cache largely depend on the mechanism to branch between multiple methods on the SPU. Therefore further details of its implementation in the prototype and the way it is called will be given in section 4.6.2.

4.4.3 Segmentation of functions

Even without the arbitrary limitation of cache size lines a fixed-size must be set for them. As the machine code translation for a Java method may exceed this limit as well a mechanism to break the code into smaller blocks must be developed if larger should methods be executable on the SPU. The issues of implementing such a mechanism will be briefly discussed in this section.

As this mechanism will work on the assembler code level only two kinds of instructions will have to be adjusted, load and store instructions and branches. Since the space for a method's code is allocated at runtime and its final location not even known when this method is JIT-compiled all of these instructions work almost exclusively with offsets. Most of them such as accesses to the data segment are relative to the procedure vector which is kept in a register. This way as long as the data segment fits into one block an instruction in another block may access the data segment by accessing the address in the procedure vector plus an offset. This should be the common case with an appropriate block size. However, data segment accesses will be problematic when the data segment is split across multiple blocks. In this case the procedure vector may not point to the same block as the data segment element requested so the relative load or store does not work. Additionally relative branches, those that branch to the current instruction plus an offset, across a block boundary are problematic. Since the blocks are cached as requested the current LS address of a block is not known.

The mechanism should ensure not to break constructs such as loops into different blocks. Whatever solution is chosen for this problem branching within the same block will certainly be less expensive performance-wise. Observing the structure of such constructs can be achieved by using the information generated in the first step of the JIT compilation, the basic block determination. This step breaks up the method after every bytecode instruction that may result in a jump so that every basic block marks the potential target of a jump. With this information

available it will be easier to limit the number of jumps across segments.

4.5 Selective execution on the SPU

In order to give the Java developer a way to select which methods should be executed on the SPU three different solutions seem feasible. They will be discussed in individual subsections and afterwards evaluated. A fourth option, leaving it to the JVM to decide which methods to offload to the SPU will be briefly presented in section 6.3.1.

4.5.1 Create a special class or interface to mark SPU threads

In Java a thread is created by deriving a class from the superclass `Thread` or implementing the interface `Runnable`. The entry point for the thread is the method `run()` which has to be implemented by the subclass. In order to start the thread its method `start()` is called. Similarly to this a thread designated to run on the SPU could extend `SpuThread` or implement `SpuRunnable` and the required methods. These special cases would have to be intercepted by the JVM and then turned into an SPU thread.

The advantage of this approach is its similarity to the current C programming model for Linux on the Cell/B.E. as well as the Java multi-thread programming model. Both usually feature a single-threaded main program that at some point branches into a number of threads. As for the C programming model, the main program is executed on the PPU while the threads run on the SPU. Java programs usually execute the main program and the threads on the same CPU. With Java on Cell/B.E., the main program would still be run on the PPU while the threads are executed on the SPUs.

4.5.2 Mark SPU methods and classes with annotations

With the recent addition of *annotations* to the Java programming language another possibility has opened up. Annotations represent a way to add meta-data to functions and classes that can be processed at compile-time, at run-time or by external tools such as test frameworks. With these it will be possible to mark certain methods or entire classes for execution on the SPU. When the JVM encounters such an annotation it will create an SPU thread if necessary and

execute the method in that thread. Another benefit of this approach lies in the fact that SPU methods can be executed at arbitrary times in the main program by simply calling them. The downside however is that the methods would be called synchronously meaning that the caller will have to wait for the called method to finish.

4.5.3 Group SPU methods and classes in a special package

Sharing the benefits and downsides with the annotations approach is another solution where the classes and thus the methods designated for execution on the SPU will be grouped together into a package such as `com.ibm.spu.math`. In the Java programing language classes with similar functionality can be combined into a so-called package. This package along with the access modifiers such as `private` and `public` also governs object access in the way that certain classes and methods may only be available to classes and methods in the same package. Apart from the lack of meta-data this solution also provides for a clear view of which methods are to be offloaded to the SPU. While it grants full flexibility about the point they are executed at the methods are also executed synchronously. Again, the JVM will have to intercept calls to methods in this package and execute them on the SPU.

4.5.4 Conclusion

While the two latter solutions share some interesting features they would require a change in the semantics of a Java program. An interesting application of these approaches lies in a scenario that is closer to a distributed-memory model with message passing. Calling these methods could trigger execution of a method on the SPU. This method is then executed asynchronously and a later call to another SPU method fetches the results calculated in the meantime.

However, since the Thread approach most closely resembles the familiar Java and C programming models and allows full exploitation of the CBEA's multi-threaded nature it presents the most favorable solution and the one that will be further pursued.

4.6 Communication and branching between SPUs and PPU

The JVM has to support a number of interactions between the PPU and the SPUs. These interactions will be discussed in this section. These include internal JVM functions such as resolving of class members at runtime and branching from one processor to another. Since especially the branching is complex the different cases, SPU to PPU and SPU to SPU, will be further detailed in additional subsections. Of the remaining cases PPU to PPU is already implemented in Cacao and PPU to SPU is supported as described in section 4.1.2. This calling scheme only allows passing information about the method to execute without any further arguments which is sufficient for starting a thread or the main method on the SPU.

4.6.1 Lazy resolving on the SPU

In order to allow lazy resolving of class members from the SPU the patcher as described in section 3.1.4 has to be extended. These extensions are relatively straightforward and can be realized by using the stop-and-signal mechanism. The standard Cacao patcher stub sets up its own stack and stores the data that is to be passed to the patcher in there. This mechanism can also be used in a similar fashion on the SPU, requiring as the only modification that the stack pointer be stored right after the `stop` instruction. This way the stop-and-signal callback function can resolve the LS pointer, reach the information stored in the patcher stub stack and pass it onwards.

Due to the SPU's alignment restrictions this requires a number of steps. First, a gap for the stack pointer has to be left in the instruction stream right after the `stop` instruction. Then the right code sequence for loading the quadword surrounding the stack pointer position, generating a bitmask and inserting the actual stack pointer at the correct offset in the quadword and storing the prepared quadword again has to be generated. A graphical representation of this process is given by figure 4.4 and the required code sequence is listed in 4.4.

```
1  /* set up the opcode + the LS pointer = opdata for the PPE-assisted call */
2  M_ILHU(((opcode << SPU_CALL_PATCHER) & 0xFF00), REG_ITMP4_SPU);
3  M_MOVE(REG_SP_SPU, REG_ITMP5_SPU);
4  M_OR(REG_ITMP4_SPU, REG_ITMP5_SPU, REG_ITMP4_SPU);
5
6  /* load the quadword surrounding the gap for the opdata */
7  M_LQR(5, REG_ITMP5_SPU);
8
9  /* calculate offset and move the opdata to the right slot */
10 disp = ((u4 *) cd->mcodeptr ) - ((u4 *) cd->mcodebase);
11 M_CWD((4 * (disp + 4)) % 16, REG_SP_SPU, REG_ITMP6_SPU);
12 M_SHUFB(REG_ITMP4_SPU, REG_ITMP5_SPU, REG_ITMP4_SPU, REG_ITMP6_SPU);
```

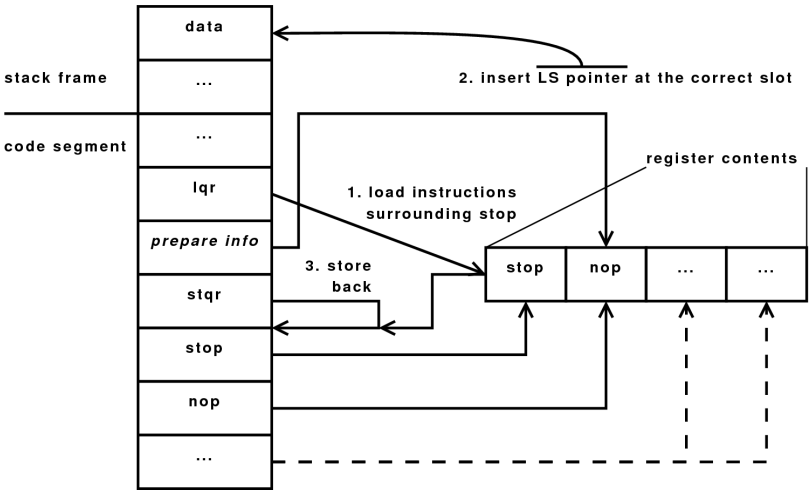


Figure 4.4: Storing an LS pointer at runtime

```
13
14 /* store the prepared quadword right after the STOP */
15 M_STQR(2, REG_ITMP4_SPU);
16 M_STOP(0x2107);
17
18 /* leave gap for the LS pointer */
19 cd->mcodeptr += 4;
```

Listing 4.4: Code sequence to store an LS pointer at runtime

The callback function only has to transfer the SPU stack into a format suitable for the Cacao patcher and can then call it. The patcher itself can run almost unmodified as it has direct access to the LS of the SPU that called the patcher. As methods may be swapped out of the LS the original version of the compiled method in the main memory has to be patched as well. This way the modifications are not lost when this method is cast out from the LS. Resolving the address of the code in main memory is possible through the information provided to the patcher which also indicates the method that has called the member. Additionally multiple SPU threads may try to concurrently resolve the same member. As Cacao already supports multi-threading entry to the patcher is protected with a lock on the object that the member belongs to. Acquiring this lock blocks until it has been released and exits if it detects that the member has already been resolved and the calling method been patched. In order to ensure that the method gets patched correctly on all SPUs this second exit condition for the patcher has to be removed. Subsequent calls to the patcher with the same member will still be faster since only the references on the SPU have to be modified, the actual resolving of the member is already done.

Once the patcher and the callback function have returned execution is automatically resumed on the SPU right after the `stop` instruction plus four bytes so that the stack pointer is skipped.

4.6.2 Branching from the SPU

Depending on the type of method called different information is required to resolve the location of the target method's code. As described in section 3.1.5 this may just be the address of the method's information structure or the object header and a number of offsets. The ways to resolve the location of the code when branching from the SPU to either a method on the PPU or another method on the SPU will be explained in this section.

As a first solution for the prototype all required information to resolve the code is passed to the PPU for every method call regardless of where it will be executed. The way this data is handled on the PPU depends on where the called method is to be executed and will be further detailed in the appropriate subsection. The easiest way of passing information between these two processors is the stop-and-signal mechanism. The memory area used for passing the arguments is realized as an additional reduced stub stack frame containing only the stack chain pointer, link register and arguments. The LS pointer to the first argument is then passed to the PPU. The Java method will set up this stub stack frame itself and insert the LS pointer to it in the instruction stream before issuing the `stop` instruction. The deallocation of this frame depends on whether the method is executed on the PPU or the SPU.

SPU to PPU

When calling a method on the PPU from the SPU the call is synchronous so the SPU will have to wait for the method on the PPU to finish before it can resume execution. Therefore using the slow but simple to use stop-and-signal mechanism is adequate for this purpose. Using the information the PPU is being passed via the stop-and-signal call it can determine which method it has to execute.

An additional requirement for calling a method on the PPU is to store all arguments to this method on the stack so they can be accessed by the PPU. The code generator already supports reading the given arguments from the memory area into the argument registers when entering a method as well as writing out the arguments from registers to memory when calling another function. Thus, it was only required to enforce that when calling a method on the PPU all arguments would be stored in memory. The callback function then transforms the data into a structure suitable for use with a special transition functions that allows calling a Java method from C code. The return value of that method, if any, is then written back to the LS by the callback function and is picked up by the Java method once execution is resumed. The stub

stack frame is then deallocated by the Java method.

SPU to SPU

Branching between two methods on the same SPU requires a number of modifications. Two points have to be ensured for this to work:

1. A method has to be able to return to the point from which it was called in the calling method
2. The information for transferring the calling method's code back into the LS if necessary has to be provided

Item one is a common requirement and is already ensured by using the link register. Using a variation of the branch instructions the address of the instruction following the branch is stored to the link register. For *leaf methods*, those methods that do not branch to further methods, the method can simply branch to the address stored in the link register once it is done and thus resume execution of the caller. For non-leaf methods the link register has to be stored in the stack frame of the calling function as defined by the SPU ABI. Once a non-leaf method is done it loads the original link register from the stack frame and can return to the calling method as well.

The required information for item two consists of the address of the data segment in main memory, the combined size of data segment and code and the entrypoint into the method. The entrypoint marks the end of the data segment and the beginning of the actual code. This information is obtained by passing the required arguments as described above to the PPU in a stop-and-signal call. The PPU then resolves the requested information and stores it directly in the LS in place of the methodinfo or objectinfo. This saves the SPU from issuing a number of small DMA transfers to resolve the pointers itself. With this info available the SPU then branches from the Java method to a short assembler function that calls `cache_rw(...)` from the cache library which transfers the method's code into the LS and returns an LS pointer to the respective cache line. The assembler function stores the return address to the calling Java method, then adjusts the procedure vector and branches to the newly transferred method while setting the link register. The connection between the different stack frames and methods can be seen in figure 4.5.

As seen in figure 4.6 when the called Java method returns it deallocates its stack frame if it is not a leaf method and branches to where it was called from, in this case the assembler function.

The stack pointer then points to the called method's stub stack frame (callee stub). Since the required information for resuming execution of the calling method is available from the calling method's stub stack frame (caller stub) the callee stub is deallocated after retrieving the link register for the calling method from it, leaving the stack pointer to point to the calling method's stack frame. The stack pointer is saved and the stack chain traced one more step back to the caller stub. The calling method is transferred back into the LS using the cache library, its stack pointer restored and execution resumed where it had previously been stopped. The stub stack frame is deallocated by the assembler part right before returning to the Java method.

Special care has to be taken with the first Java method that is called. With just the mechanism described above the SPU would try to locate a non-existing stub stack frame for the method that called the first Java method. In order to prevent this the link register is adjusted for the first Java method so that it does not return to the DMA transfer stub but instead to an exit function that will safely move on to the C-part on the SPU from which the first Java method was called.

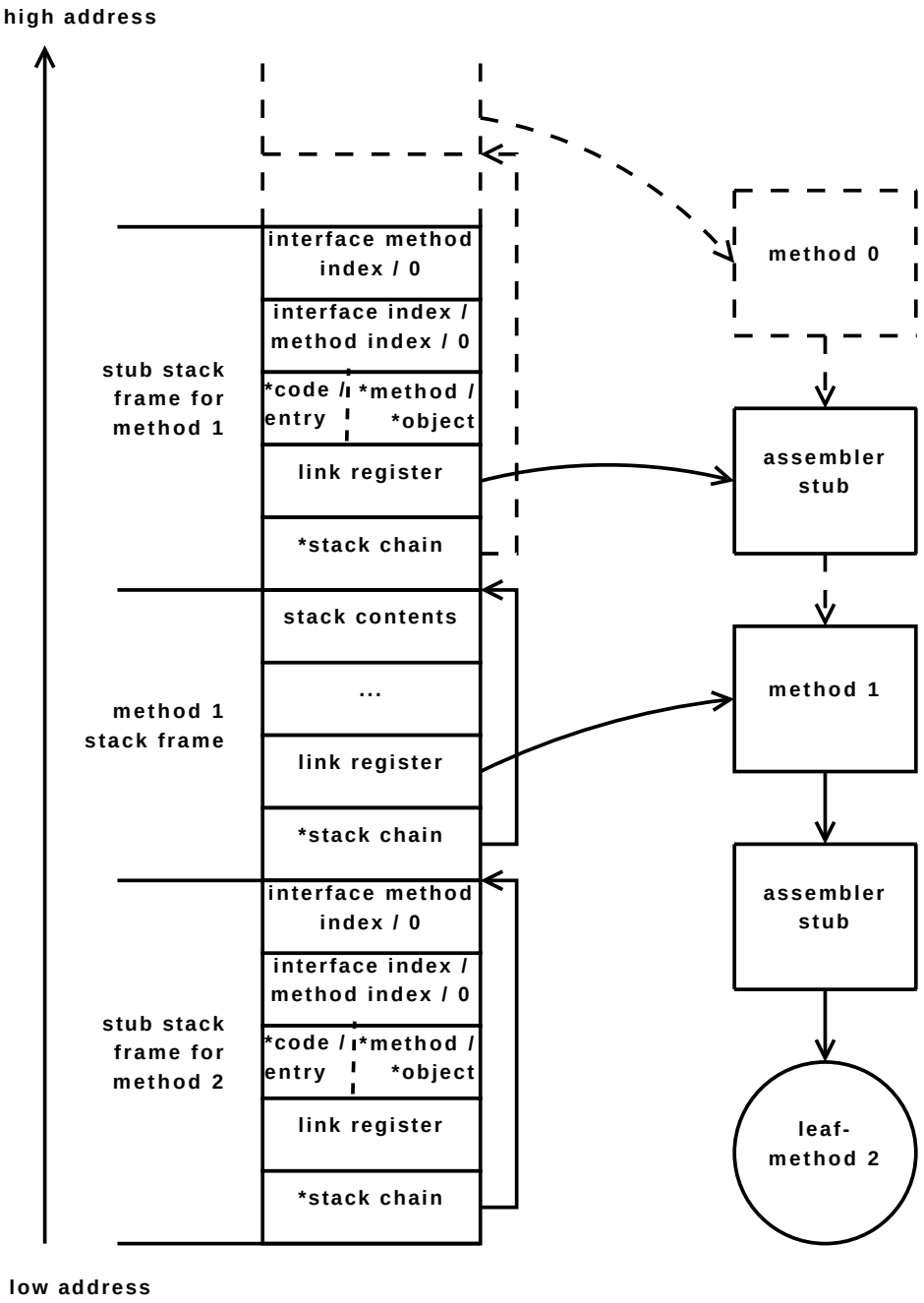


Figure 4.5: Layout of the real and stub stack frames

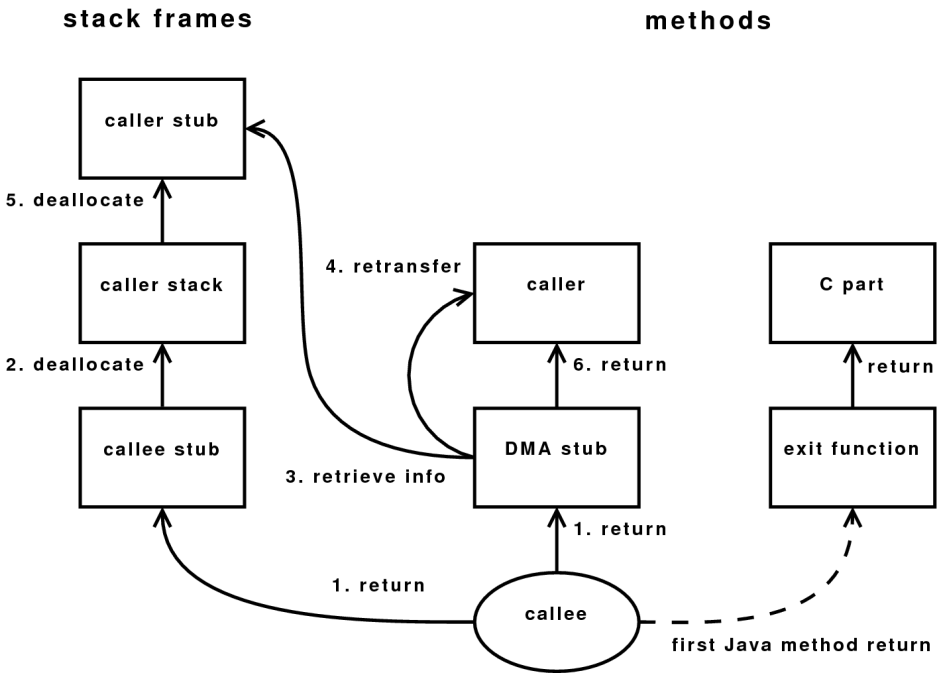


Figure 4.6: Returning from a Java method

5 Evaluation

As part of this thesis a prototype has been developed in which some of the concepts described have been implemented. As a prototype it provides only a proof of concept and is not yet capable of running full programs. However it provided good use in testing out some ideas. The capabilities of this prototype will be examined in one subsection while some rough guidelines and advice for a programing style that may benefit Java on Cell/B.E. will be given in another.

5.1 Capabilities of the prototype implementation

The task of porting a JVM is in itself already a very complex and time-consuming one. However, especially for such a unique architecture as Cell/B.E. many new challenges have to be solved and new concepts developed. Therefore not all solutions presented in chapter 4 could be successfully implemented during the limited available time for this thesis. Currently the prototype is capable of doing the following things:

- It can set up a single SPU to await mailbox notifications from the PPU. The SPU then independently fetches the method's code using the SPU Software Managed Cache Library and executes it.
- All submethods that are part of a class called `Spu` will also be executed on the SPU. Using the branch mechanism described in section 4.6.2 the code for the submethod will also be fetched using the cache library. All required information for returning to the calling method will be stored in stub stack frames. This solution was chosen during development instead of one of the ideas presented in section 4.5 as it was easier to implement and provided a simple way to test the different branch semantics between processors.
- All other methods will be executed on the PPU to which the SPU may pass arguments as explained in section 4.6.2. This way the SPU is for example able to print out values to the console by calling the IO methods on the PPU.

- Most types of members can be lazily resolved from the SPU using the mechanism discussed in section 4.6.1 which includes passing the required information to the PPU using a stop-and-signal notification.
- Object access follows the principles described in section 4.3.1. This means that each access to a field from the SPU requires a single DMA transfer, no data will be cached. For the prototype no thread-safety mechanisms are observed and the GC has not been extended to account for object references on the SPU.
- Apart from object accesses and method calls the code generator for the SPU mostly supports arithmetic operations. These were implemented early for two reasons. First, they could be used to test if other bytecode instructions that load and store data for example from the stack as well as the handling of arguments and registers were correctly implemented. Second, they can be used to create load and perform measurements on the SPU when calling them in a loop. Therefore the necessary branch instructions for setting up simple loops were also implemented.

The current prototype heavily relies on stop-and-signal calls for many different situations. As they provide an easy way to pass information back and forth between PPU and SPU they were used in many cases during implementation to facilitate the different steps. For some situations such as calling the patcher this mechanism makes sense as the SPU has to wait for the PPU to finish its part. However for others such as obtaining the required information for calling a method other ways such as an asynchronous DMA transfer may be more suitable. This has the benefit of enabling the SPU to fetch the information in the background ahead of time and it eases the load on the PPU which may become a bottle neck when a greater number of SPUs constantly call methods on it. Depending on the situation it may also improve performance on the SPU. While the overhead of a stop-and-signal call has been measured to be at around 1.3×10^{-5} seconds obtaining the required information via DMA may be faster depending on how many pointers have to be resolved.

5.2 Programming advice

Due to the fact that only very limited Java programs can be run as intended by the prototype in its current stage it is hard to give any programming advice. However with the current non-caching model in use and even more so with the later shared heap access model with a local cache it will be very important to be careful with synchronized statements. They already provide a performance barrier for traditional JVMs where all threads access the same memory space. However

with the disjoint model of Cell/B.E the penalty will be much higher due to the increased latency that comes with multi-threading actions such as synchronization across different memories and cores. When a program manages to keep these barriers low while providing and handling data in a way that allows threads to work largely unsynchronized this program could achieve a good performance increase by employing the SPUs.

6 Ending

6.1 Experiences during porting

6.1.1 Infinite recursion with debug flags

One of the gravest barriers in the early stage was getting the official Cacao release to run on just the PPU of a Cell/B.E. based system. An unfortunate combination of compile flags led to an infinite loop. A certain Java method in Classpath responsible for providing interactions between Classpath and the JVM called the `println()` Java method in its constructor. At this point the class providing `println()` was not yet initialized so this step had to be performed first. Through a number of further methods required for initialization the originating Java method was called again as I/O functions such as `println()` require the JVM interaction. At this point the loop started anew. The obvious result of this was a stack overflow however without any indication of its cause. While Cacao provides a very verbose debug option, `-verbose:call` that prints a message upon each method's beginning and end this option produces data in excess of 2 GiB which is too much when not knowing what to look for. The key to finding the root cause was using GDB's backtrace option once Cacao had caused the stack overflow. Using the available backtrace `-verbose:call` could be set incrementally at methods lower in the stack ie. methods that were called earlier. Setting this debug switch manually resulted in only those methods generating verbose output that were compiled after the switch had been set and in turn generating less output with more relevance. Incrementally setting the switch at earlier times soon showed a certain sequence of the same methods being call over and over again. Once the method calling `println()` in its constructor had been identified solving this bug was simply a matter of turning off the `debug` flag for classpath.

6.2 Related work

Due to the perceived notion that only lower-level complex programing languages such as C and Fortran can exploit the full potential of the Cell/B.E. very little work has been done in order to enable Java on Cell. The few known approaches will be discussed in the following.

6.2.1 CellVM

CellVM [NGF06] takes an approach similar to one discussed in this thesis but was not further pursued. A JVM, in this case JamVM, is modified to run on the SPU. The final goal is to provide a single system image so that a homogeneous view on Cell/B.E. is provided. The architecture built by CellVM consists of an *CellVM Abstraction Layer* that receives and redirects all requests accordingly to the different JVMs working in the background. These include one *CoreVM* based on JamVM on every SPU and the *ShellVM* on the PPE. The most common bytecode instructions can be directly executed by the CoreVMs and only for more complex function is the assistance of the ShellVM required. These mostly include functions concerned with access to system parts and object and array creation. As the entire heap in this architecture resides in main memory CellVM implements a software-controlled cache mechanism for the SPUs so previously fetched data may be reused from the LS again if possible.

The authors also provide a performance evaluation in which they demonstrate cache hit rates between right below 80% and up to 100%. Additionally an almost linear speed-up from using 1 up to 8 threads is observed.

6.2.2 SPU-accelerated parallel JIT-compilation of methods

A completely different approach is pursued by [Hoy07]. In this master thesis project Cacao will be accelerated not by using the SPUs for highly-parallel execution of methods but by using them for parallel compilation of methods. The benefit of this approach is the fact that the speed-up does not depend on the Java program executed and how effectively it exploits the parallelism. If this way is in fact fast enough methods can be compiled speculatively in advance so that the overhead of JIT-compilation should almost cease to exist. Unfortunately no further information about the progress of this thesis could be obtained. However it will be interesting to see for which part the SPU's provide more benefit, for execution of methods or for their compilation.

6.3 Outlook

Unfortunately the outlook for Java on Cell/B.E. is currently hard to determine. IBM as well as other third-party vendors providing development tools and frameworks for Cell/B.E. focus their current efforts on mostly C/C++ based solutions. As these are still the most important languages in the HPC sector this direction is understandable. Even though Java still cannot quite live up to C/C++'s performance level there is a reason why there is such a huge movement behind it. So if Cell/B.E. may not be the right choice to compete in the HPC sector it could still be used to accelerate computationally intensive Java applications such as financial ones or other business software.

For the future development of Java on Cell/B.E. many different directions are suggested by this thesis. Apart from the suggested implementation variations presented in chapter 4 that have not been realized yet a number of further-reaching optimizations may be feasible which will be discussed in the next subsection.

6.3.1 Optimizations

Auto-vectorization

A rather obvious yet highly complex optimization is auto-vectorization which means the automatic generation of SIMD code from the scalar code provided. Potential areas for this optimization can be found mostly in loop unrolling. If the body of the loop performs work on arrays multiple iterations may be replaced with a few SIMD instructions. While SIMD exploitation is essential to fully utilize the power of the SPU this is a non-trivial task with lots of restrictions and pitfalls to observe. Even with traditional ahead-of-time compilers such as GCC which can spend a lot more time inspecting the structure and data dependencies of the code the auto-vectorization is still a work in progress[1]. Auto-vectorizing Java code is especially problematic since code that may throw exceptions cannot be optimized and each access to an array may throw an out-of-bounds exception that occurs when trying to access an element outside the bounds of the array.

[1] http://gcc.gnu.org/projects/tree-ssa/vectorization.html

Automatic offloading

A JIT compiler's ability to optimize methods at runtime can be used to further advantage. Once the JVM identifies a method as being numerically intensive it may recompile the method for execution on the SPU. Methods that qualify for this optimization should additionally require only little external data from other methods and run long enough to compensate for the overhead of executing a method on the SPU. A simple way to measure the numerical intensity of a method may be just to count the number of arithmetic bytecodes and weigh them against other instructions. This optimization would be especially beneficial in combination with auto-vectorization.

Extending the GC

As discussed several times before for example in section 4.3.1 when references to objects in the main memory are used on the SPU this requires an extension of the GC. In this case it also has to include the various places where the SPU may store object references in its calculations whether an object is still reachable. As long as the references reside on the stack in the LS the problem is fairly easy as the PPU and thus the GC has direct access to the LS. However as parts of the Java stack for an SPU method may be mapped to SPU registers the object references may also reside there while still being valid. Accessing the SPU registers from the PPU requires saving the current SPU context and loading a new program that writes out the registers to main memory. This step is prohibitively performance-expensive.

However, a more efficient solution can be based on the fact that the PPU can cause interrupts on the SPU as described in [IBM07a] chapter 18, *SPE Events.* When the SPU has been configured to receive interrupts it will automatically branch to LS address 0 and execute the code there which may be a user-defined interrupt handler. Used in conjunction with the GC the GC could always cause an interrupt on the SPU whenever it is run. The interrupt handler which has direct access to the SPU registers may write out their values to main memory from where they could be used by the GC. Once the GC has finished work the interrupt handler has to rebuilt the execution state of the SPU thread before the interrupt and can then return back to where it was called from.

6.4 Conclusion

Java on Cell/B.E. provides an interesting combination of two relevant modern-day technologies. The multi-threaded nature of both show great potential for a combined work and the additional layer of the JVM allows hiding the heterogeneous architecture of the Cell/B.E. so that Java programmers do not have to adjust considerably to this platform.

The work done in this thesis has shown that it is indeed possible to run JIT-compiled code on the SPUs and transparently access objects across the different memories. The Java programs that can be run so far are very limited and do not run efficiently. However, a significant number of optimizations has been discussed which show good potential for performance increases. While Java on Cell/B.E. will certainly not provide performance on a level with the programming languages currently supported on Cell/B.E. such as C and C++ even a single-digit speed-up for suitable programs over traditional JVMs would mean a valuable result.

Unfortunately no JVM except for the other related work items listed has been enabled to run on a heterogeneous architecture such as the CBEA. Providing this support requires a number of major modifications most of which were too time-expensive to be completed during the limited time available for this thesis. This also means that no comprehensive evaluation or estimation of the performance of Java on Cell/B.E. could be conducted.

Further work on this topic is required and most likely rewarding and this paper gives enough directions which may be pursued in the future.

6.5 Appendix

6.5.1 Tools

Perl script to convert the SPU ISA to Cacao codegen macros

```
1 #!/usr/bin/perl -w
2 open (ISA, "SPU_ISA.txt");
3 $i = 0;
4 @lines = <ISA>;
5 while ($line = $lines[$i++]) {
6   if ($line =~
7 /^\s*((0|1)(0|1|\s)+)\s*(((I\d+)|([SRC][ABCT])|(\/{3})|C)\s+(([SRC][ABCT])|(\/{3})|(\s*))+)↩
    /) {
```

```
8   $opcode = $1;
9   @args = split(/\s+/, $4);
10  $opcode =~ s/\s//g;
11  $opname = uc((split(/\s+/, $lines[$i-3]))[0]);
12  $opcode = unpack("N", pack("B32", substr("0" x 32 . $opcode, -32)));
13  $num = scalar(@args);
14  unless ($line =~ /\//) {
15    next;
16  }
17  if ($args[0] =~ /I(\d+)/) {
18    $imm = $1;
19    shift(@args);
20    $num--;
21    $regs1 = $regs2 = lc(join(",", @args));
22    $regs1 =~ s/\/{3}//g;
23    $regs2 =~ s/\/{3}/0/g;
24    print substr("#define M_$opname(imm,$regs1)" . " " x 40, 0, 40) .
25 "M_OP${num}_IMM${imm}($opcode,imm,$regs2)\n";
26  }
27  else {
28    $regs1 = $regs2 = lc(join(',', @args));
29    $regs1 =~ s/\/{3},?//g;
30    $regs2 =~ s/\/{3}/0/g;
31    print substr("#define M_$opname($regs1)" . " " x 40, 0, 40) . "M_OP$num($opcode,$regs2)\↩
          n";
32  }
33 }
34 }
35 close (ISA);
```

Listing 6.1: Perl script to help convert the SPU ISA

6.5.2 Trademarks

- Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc.
- PS3 and PLAYSTATION are registered trademarks of Sony Computer Entertainment Inc
- Cell Broadband Engine and Cell/B.E. are trademarks of Sony Computer Entertainment Inc.
- PowerPC is a trademark of IBM Corp.
- XDR is a trademark of Rambus Inc.
- All other trade names are the service marks, trademarks, or registered trademarks of their respective owners

6.5.3 Glossary

Term	Meaning
ABI	Application Binary Interface, defines assembler-level interface including stack frame layout and dedicated registers
API	Application Programming Interface, defines high-level source code interfaces
Bytecode	The intermediate machine-independent language used by Java
CBEA	Cell Broadband Engine Architecture, the architectural definition on which implementations will be based
Cell/B.E.	Cell Broadband Engine, the first implementation of the →CBEA
CESOF	CBEA Embedded SPE Object Format, the object format for embedding →SPU programs within →PPU programs
CISC	Complex Instruction Set Computer, a computing architecture that supports complex and powerful instructions
Doubleword	Defined as a 64-bit type on →Cell/B.E.
EIB	Element Interconnect Bus, the bus connecting the different components of the →CBEA
ELF	Executable and Linkable Format, the common file format for executables used in Linux and many variations of Unix
Field	In Java a data member of a class, unlike a method member
GC	Garbage Collector, responsible for freeing unused heap space
GCC	The Gnu Compiler Collection, a number of different compilers provided by the GNU project
GCJ	The Gnu Compiler for Java, an ahead-of-time compiler that can compile Java source code to native machine code
GiB	Gibibyte, 1024 →MiB
Halfword	Defined as a 16-bit type on Cell/B.E.
HPC	High-Performance Computing
IEEE 754	IEEE standard, defines floating-point calculations and semantics
JDK	Java Development Kit, consists of a →JRE and additional developer tools such as a Java compiler
JIT	Just-in-time compiler, dynamically compiles methods to machine code when they are actually called
JRE	Java Runtime Environment, consists of a JVM and the compiled class libraries, allows running Java programs
JVM	Java Virtual Machine, the platform for executing Java bytecode

Continued on the next page

Term	Meaning
KiB	Kibibyte, 1024 bytes
libspe2	The runtime management library for managing the SPUs
LS	Local Store, the small memory directly accessible by the →SPU
Member	In Java all fields and methods of a class
MiB	Mibibyte, 1024 →KiB
MFC	Memory Flow Controller, autonomous unit that transfers data between →LS and main memory
PPE	PowerPC Processing Unit, consists of →PPU, L1 cache and →PPSS
PPSS	PowerPC Processor Storage Subsystem, includes the →PPU L2 cache and a Bus Interface Unit
PPU	PowerPC Processing Element, the actual processing core
Procedure Vector	In Cacao terms the entrypoint to a method and the base for the data segment
Quadword	A 128-bit type, containing four (quad) 32-bit →words
RISC	Reduced Instruction Set Computer, a computing architecture that supports only simple and easy to decode instructions
RMI	Remote Method Invocation, the Java standard for message passing in distributed systems
SIMD	Single-Instruction-Multiple-Data, class of assembler instructions which can work on multiple data at once
SPE	Synergistic Processing Unit, consists of →MFC, →LS and →SPU
SPU	Synergistic Processing Element, the actual processing core
STI	Sony, Toshiba and IBM, the consortium that developed the →CBEA
Word	Defined as a 32-Bit type on →Cell/B.E.
XDR	Extreme Data Rate memory, the kind of memory used for the main memory of the →Cell/B.E.

List of Figures

List of Tables

List of Listings

Bibliography

[BDB99] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. The Design and Implementation of Dynamo. http://www.hpl.hp.com/techreports/1999/HPL-1999-78.html, June 1999. Fetched September 2007.

[Fen01] Kevin Fenwick. A Performance Analysis of Java Distributed Shared Memory Implementations. http://www.dhpc.adelaide.edu.au/reports/107/dhpc-107.pdfhttp://www.dhpc.adelaide.edu.au/reports/107/dhpc-107.pdf, October 2001. Fetched September 2007.

[Gar07] Robin Garner. DaCapo Benchmarks. http://cs.anu.edu.au/~Robin.Garner/dacapo/regression/, September 2007. Fetched September 2007.

[GJS05] James Gosling, Bill Joy, and Guy L. Steele. The Java Language Specification. Addison-Wesley Longman, 3rd edition edition, June 2005.

[Hac07] Daniel Hackenberg. Fast Matrix Multiplication on Cell (SMP) Systems. http://tu-dresden.de/die_tu_dresden/zentrale_einrichtungen/zih/forschung/architektur_und_leistungsanalyse_von_hochleistungsrechnern/cell/index_html, 2007. Fetched September 2007.

[Hoy07] Francisco Hoyos. SPU-accelerated parallel JIT-compilation of methods. http://c1.complang.tuwien.ac.at/pipermail/cacao/2007-August/000391.html, 2007. Discussed on a mailing list in August 2007.

[IBM] IBM. The Cell project at IBM Research. http://www.research.ibm.com/cell/home.html. Fetched on September 2007.

[IBM06] IBM. Cell Broadband Engine Architecture. http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/1AEEE1270EA2776387257060006E61BA, October 2006. Fetched September 2007.

[IBM07a] IBM. Cell Broadband Engine Programming Handbook. http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/

9F820A5FFA3ECE8C8725716A0062585F, April 2007. Fetched September 2007.

[IBM07b] IBM. Cell Broadband Engine SDK Example Libraries. http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/3B6ED257EE6235D900257353006E0F6A/, September 2007. Fetched September 2007.

[IBM07c] IBM. Software Development Kit 2.1 Programmer's Guide. ftp://ftp.software.ibm.com/systems/support/bladecenter/cpbprg00.pdf, March 2007. Fetched September 2007.

[IBM07d] IBM. SPE Runtime Management Library. http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/771EC60D862C5857872571A8006A206B, March 2007. Fetched September 2007.

[IBM07e] IBM. SPU Application Binary Interface Specification. http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/02E544E65760B0BF87257060006F8F20, March 2007. Fetched September 2007.

[IBM07f] IBM. Synergistic Processor Unit Instruction Set Architecture. http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/76CA6C7304210F3987257060006F2C44/, January 2007. Fetched September 2007.

[IEE85] IEEE. IEEE 754: Standard for Binary Floating-Point Arithmetic. http://grouper.ieee.org/groups/754/, 1985. Fetched September 2007.

[Kea04] Andreas Krall and et al. The CACAO Java Virtual Machine. Only available in source form from http://www.cacaojvm.org, April 2004. Built September 2007.

[LY99] Tim Lindholm and Frank Yellin. The Java Virtual Machine Specification. Prentice Hall PTR, 2nd edition edition, April 1999.

[NGF06] Albert Noll, Andreas Gal, and Michael Franze. CellVM: A Homogeneous Virtual Machine Runtime System for a Heterogeneous Single-Chip Multiprocessor. http://www.ics.uci.edu/~franz/Site/pubs-pdf/ICS-TR-06-17.pdf, December 2006. Fetched September 2007.

[Shu04] Kazuyuki Shudo. Performance Comparison of Java/.NET Runtimes. http://www.shudo.net/jit/perf/, October 2004.

[Ull07] Christian Ullenboom. Java ist auch eine Insel. Galileo Computing, 6., aktualisierte und erweiterte auflage edition, 2007.

[Wik07] Wikipedia. PowerPC Implementations. http://en.wikipedia.org/w/index.php?title=PowerPC_Implementations&oldid=154400705, 2007. Fetched September 2007.

[WSO+05] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. The Potential of the Cell Processor for Scientific Computing. http://www.lbl.gov/Science-Articles/Archive/sabl/2006/Jul/CellProcessorPotential.pdf, 2005. Fetched September 2007.