# Optimization Strategies for a Java Virtual Machine Interpreter on the Cell Broadband Engine

Kevin Williams[1], Albert Noll[2], Andreas Gal[3] and David Gregg[1]

[1]Trinity College Dublin, Dublin, Ireland.

[2]ETH Zurich, Zurich, Switzerland.

[3]University of California, Irvine, CA, USA.

{ kwilliam, david.gregg } @ cs.tcd.ie

albert.noll@inf.ethz.ch

gal@uci.edu

**Abstract**

Virtual machines (VMs) such as the Java VM are a popular format for running architecture-neutral code in a managed runtime. Such VMs are typically implemented using a combination of interpretation and just-in-time compilation (JIT). A significant challenge for the portability of VM code is the growing popularity of multi-core architectures with specialized processing cores aimed at computation-intensive applications such as media processing. Such cores differ greatly in architectural design compared to traditional desktop processors. One such processing core is the Cell Broadband Engine's (Cell BE) Synergistic Processing Element (SPE). An SPE is a light weight VLIW processor core with a SIMD vector instruction set. In this paper we investigate some popular interpreter optimizations and introduce new optimizations exploiting the special hardware properties offered by the Cell BE's SPE.

## 1 Motivation

The requirements of modern computing applications demand a diverse set of processing abilities, and increasingly processors are being designed with particular classes of application in mind. An example of this phenomenon is the Cell Broadband Engine (Cell BE)[8]. The Cell BE provides one general purpose superscalar core (the Power Processing Element, or PPE), and eight identical special-purpose, VLIW vector cores known as synergistic processing elements (SPE). These SPEs are optimized for multimedia, streaming and numerical computing.

The heterogeneous design of the Cell BE architecture provides a significant challenge in supporting languages such as Java and C# which are implemented with a virtual machine.

The Java VM allows code to be distributed in an architecture-neutral format, and execute in a safe, managed runtime. Although the Java Virtual Machine (JVM) can be implemented on the general purpose PPE core using known techniques, it is desirable that Java threads could also run on the special purpose SPEs. Such a solution provides a developer with a single abstract VM as a target platform. An advantage of such a VM is that the execution details of whether to run on SPE or PPE can be decided at runtime by the VM. Efficiently implementing the JVM on Cell BE SPEs is a significant challenge since the architecture is aimed primarily at regular vector code, rather than branch heavy code with irregular data access patterns.

The Java VM is typically implemented with some combination of interpretation and just-in-time (JIT) compilation. A popular approach is to interpret code initially, and compile the code only when it has been identified as an execution hot spot. Interpreters are also commonly used where memory is limited, because an interpreter is normally much smaller and simpler than a JIT compiler.

This paper introduces a feasibility study for the interpretation of the most popular portable code, Java Virtual Machine bytecode, on the most pervasive high-performance system, the Cell BE. We evaluate a number of existing interpreter optimizations on the Cell BE SPE, and report the resulting effects on running time. We also propose a number of novel optimizations aimed specifically at the Cell BE SPE, and measure their performance.

The rest of this paper is organized as follows. Section 2 provides some background information. Section 3 discusses the architectural features of the Cell BE SPE. Section 4 addresses the design issues for the construction of our JVM interpreter. Section 5 describes the experimental setup employed for analysing and testing our optimizations. Section 6 discusses the optimizations implemented in our VM. Section 7 describes previous work related to this paper. Finally, Section 8 concludes the paper.

# 2 Background

## 2.1 JVM

The JVM [10] is an abstract stack-machine in which an operand stack is used to load values from memory and store the intermediate results of expressions during execution. Operations such as arithmetic instructions pop their operands from an *operand stack* and store their results back onto the same stack. A *local variable array* is available for loading and storing variables local to

the executing function. JVM instructions perform loads and stores of these variables from the local variable array onto the operand stack. A set of control flow instructions are available within the JVM specification for use in conditional branching and method calling. JVM instructions are referred to as bytecodes and consist of a single byte representing the individual instruction and zero or more operands attached to that bytecode.

## 2.2 Interpreters

There are a number of different techniques for interpreting Java bytecode. The simplest is a *switch-dispatch* method consisting of a large switch statement. The switch contains a case for each opcode in the instruction set. Switch dispatch can be implemented in ANSI C. An alternative approach is a *direct threaded* [1] interpreter in which bytecodes jump directly to the next instruction to be executed. Branch targets are defined using GNU C's *labels as values* whereby a computed *goto* dispatches each JVM instruction.

## 2.3 JVM Instruction Execution Cycle

The majority of JVM instructions follow a similar execution cycle of three phases:

1. Argument Fetching - Load arguments and operands either from the stack, the local variable array or the JVM instruction's operand.

2. Execution - Execute the function defined by the instruction.

3. Instruction Dispatch - Fetch and dispatch the next instruction in the programs instruction stream.

## 2.4 CellVM

CellVM [11] is a port of JamVM to the Cell BE. It consists of two distinct JVMs each running on separate cores. *ShellVM* runs on the main core and distributes work to each of the SPE cores on the Cell BE. A second JVM, *CoreVM*, runs on an SPE and executes the Java bytecode supplied by the ShellVM. A limited number of instructions in the JVM instruction set do not run on CoreVM. These include instructions that create new objects (including arrays) that must access the heap stored in main memory. A switching mechanism is employed to revert back to ShellVM processing for these instructions [11].

The CoreVM interpreter is a direct threaded intepreter. To allow efficient loading of instructions the instruction stream is 16 bytes wide and aligned to a 16 byte boundary. The first 32 bit slot of the instruction encoding acts as a padding, the second 32 bit slot is the address of the target instruction. The third and fourth slots are used as operand storage in those instructions where operand values are required, otherwise they act as padding for alignment purposes. Aligning to 16 byte boundaries is important for performance, as all loads and stores perform memory access on 16 byte boundaries and additional instructions would be required to move the value in the preferred slot. Section 6.1.1 provides more detail on operand storage and byte alignment and how the CoreVM interpreter accesses these values.

### 2.5 Cell BE

The Cell Broadband Engine is a multi-core processor designed for high performance computing and execution of data rich applications such as multimedia and gaming algorithms. It features in Sony's Playstation 3 games consoles and IBM's *BladeCenter QS20* servers. It contains nine cores in total. The main core, the PPE, is a 64-bit Power Architecture core. The eight Synergistic Processing Elements (SPE) are VLIW processors with SIMD instructions that operate on 128 bit vector registers. The instruction set architecture provides instructions for byte, word, long, float and double operations. The processor provides an ideal platform for computational problems such as media-processing and scientific applications [14].

## 3 Architectural Features

The architectural features of the Cell BE SPE are targeted towards large compute-intensive tasks. Hence, the execution of instructions is strictly in-order and branch-prediction logics are not present. The instruction set of the SPE provides a rich set of vector operations meeting the high numerical requirements of scientific applications. Each SPE is an individual execution unit with its own program counter, registers and local store. This section briefly discusses some of these features, the impact they have on interpreter design and the difference between the Cell BE implementation to that of a standard general purpose processor.

### 3.1 Local Store

Modern architectures provide a memory hierarchy which includes a number of hardware controlled caches located close to the processing core. The Cell BE takes a different approach. Each

```
Cycle  Pipeline Schedule      Pseudo Assembler     C Code
01     12                     addi    pc,pc,1      pc++;
01     123456                 load    op1,sp[-1]   op1 = sp[-1];
02      234567                load    op2,sp[-2]   op2 = sp[-2];
03       34                   addi    sp,sp,-1     sp--;
08        ----89              add     op2,op2,op1  op2 = op2 + op1;
10            --012345        store   op2,sp[-1]   sp[-1] = op2;
11               123456       load    next,pc[0]   next = pc[0];
17                -----7890   branch  next         goto next;
```

Figure 1: Pipeline schedule for VM instruction IADD

SPE has 256KB of software controlled *Local Store* in which it stores all instructions and data used by the SPEs. A load instruction from local store has a 6 cycle latency. This is significantly higher than the level 1 cache hit latency of many processors, which is often 2 – 3 cycles [2].

Long *dependence chains* [1] over a small number of machine instructions are common among stack-based JVM instructions. Each phase of execution depends on values resulting from the previous phase. The first phase of the code to implement a JVM instruction typically performs a load of some value(s), the second performs an operation on those values, the final phase then stores the result. With a six cycle per load latency, phase two will spend much time waiting for operands to load from the first execution phase. To demonstrate the significant costs of these loads and stores in an interpreter let us examine the schedule of a typical IADD implementation.

Figure 1 shows the profile of an IADD instruction schedule. The first column shows the cycle at which an instruction is issued. The second column displays the number of cycles an instruction requires to complete (with dashes indicating a stall in the issue of an instruction). The third column is a pseudo assembler listing for the corresponding SPE native assembler code, and column four is an equivalent C implementation.

From the profile in Figure 1 we can see 8 instructions are issued in 17 cycles resulting in 11 stalls (11 dashes). The reason for 11 stalls is due to the SPE's dual issue pipeline (see Section 3.4). The first two instructions are issued on the first cycle of execution, each down a separate pipeline. The store of the result back to memory produces two stalls, as it would have been possible to perform a dual issue here only for the data dependency between the addition and the store. The further 9 stalls are caused by one or more of the three loads contained in the code sequence. The first four stalls in the sequence preceded the addition (this is phase two of the instruction), the stalls occur because the values loaded in phase one have not yet been fully loaded. The additional five stalls occur in the loading of the program counter (PC) target

[1] A series of instructions where each instruction is dependant on results from previous instructions.

address. These memory accesses account for over half of the processing of the IADD instruction.

## 3.2 Branch Predictors

During the execution of a branch instruction the target of that branch is not known to the processor until a very late stage in the pipeline execution. Branch predictors use various techniques in an attempt to issue an accurate prediction as to which path a particular branch will follow. A correct prediction results in little or no stalls caused by branching. If a branch is predicted incorrectly a penalty approximately the length of the processors pipeline is incurred. Hardware predictors have proved enormously successful with prediction rates of greater than 90% from the best predictors [13].

Many high performance vector algorithms have few if any conditional branches. For this reason the SPE processor assumes sequential execution of software and so does not have a hardware branch predictor. Instead all branches are assumed not taken unless hinted otherwise by specialized *hint instructions* in the SPE instruction set architecture [4] . The hint instruction loads a *branch target buffer* (BTB) with two addresses. One address points to the location of the branch, the second to the target location which the branch is to be hinted towards. A correct hint allows the program to continue execution without penalty. An incorrect hint behaves just like a mispredicted branch in regular hardware, the pipeline must be flushed and the correct instruction stream fetched and executed. This flush has a penalty of 18 - 19 cycles [4]. Given that JVM instructions like IADD and ILOAD can take anything from 10 - 30 cycles to execute on a Cell BE SPE, a further stall of 18 cycles will have an impact of approximately 50% increase in the total cycle count of the instruction.

With hinting it is possible to achieve perfect prediction across all the interpreters dispatches, provided the target of the branch is known early enough. To avoid stalling the pipeline, the hint must be scheduled four instruction pairs and 11 cycles (i.e. approximately 15 cycles) before the corresponding branch [7]. If the hint is scheduled later then the pipeline will stall even if the hint is correct.

## 3.3 Register Files

A Cell BE SPE has a 128-entry register file. Each register is 128 bits wide. This is a substantial register file and has been included on the SPE to mitigate the cost of replacing traditional caching for a single local storage unit. The significant size of the register file allows a Java

interpreter to cache many more variables in registers and hence reduce the number of loads and stores in the interpreter. Many of our optimizations discussed in Section 6 utilize the register file to store novel pieces of information about the interpreter. For example, the pipeline technique uses registers for program counter and operand values, stack caching stores values from the top of stack in registers and JVM branch prediction uses register values to make predictions about the direction a dispatch branch will take.

### 3.4 Dual Issue Pipeline

Each SPE has two pipelines (pipeline 0 and pipeline 1) into which it can issue up to two instructions per cycle. Each pipeline can receive a fixed set of instructions from the SPEs instruction set. The set of instructions issued to pipeline 0 include arithmetic operations and single and double precision floating point operations. Pipeline 1 receives instructions including loads, stores and branches. This technique allows, for example, an addition and a load to be performed in the same cycle. Dual issued instructions can be observed throughout the code listings of this paper.

## 4 Design Considerations

The Cell BE SPE is a memory constrained VLIW in order architecture. A successful interpreter optimization for this particular type of architecture is required to be memory efficient and sensitive to the in order nature of the execution pipeline. This requires the implementation for each opcode in the interpreter to be conscious of the data dependencies through the JVM Instruction Execution Cycle presented in Section 2.3. The Philips TriMedia VLIW Media Processor has some analogous features to the Cell BE SPE. Hoogerbrugge *et al.* [6] present a pipeline interpter design in their paper for the Trimedia Processor. We have implemented their techniques and improved on the concept by adding operand fetching to the pipeline (see Section 6.1).

There are a number of interperter optimizations presented in literature that are not feasible for implementation on the Cell BE SPE. These include Ogata *et al.'s* [12] bytecode fetch model and Ertl's [5] multi-state stack caching techniques. These optimizations store a different interperter for every state a Virtual Machine can enter. For example, Ertl has a state for each number of variables cached on the top of stack. The interpreter for each state writea to and from the stack according to the state they represent. We have implemented a stack caching technique for a number of different caching levels but only store a single interpreter per implementation. Superinstructions are a common and succesful interpter optimization. The technique

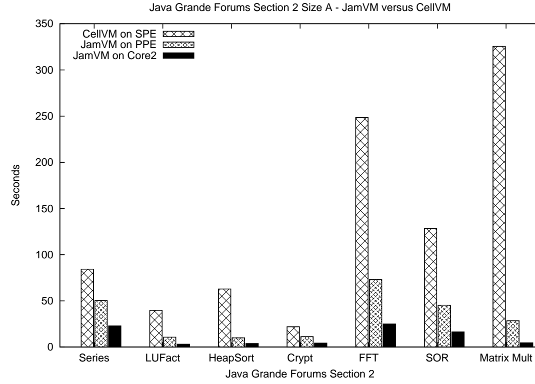


Figure 2: PPE interpreter versus SPE interpreter running times (smaller is better).

invlovles constructing new 'superinstructions' which are optimized concatenations of more than one Java instruction. These superinstructions lead to increased memory usage and so we have implemented a small set of possible instructions as an evaluation of the technique on the Cell BE SPE.

# 5 Experimental Setup

The experiments performed for this study were carried out on an IBM BladeCenter QS20 server running RedHat Linux (kernel 2.6.20-CBE). The PPE and SPE compiler used were *ppu-gcc* and *spu-gcc* (GCC) version 4.1.1 along with IBM Cell SDK 2.1. The benchmark suite used is the Java Grande Forums Version 2 Section 2 Size A. Figure 2 plots timing results for JamVM running on the Cell BE PPE, Intel Core2 (2.4GHz) and CellVM running on a single SPE. Both JamVM running on the Cell BE PPE and Intel Core2 significantly outperforms CellVM, its port to the Cell BE SPE. This graph outlines how difficult the interpretation of Java bytecode is on the Cell BE SPE.

The interpreter was written in C to allow testing of different variations, hence the performance of the interpreter is dependent on the compiler used. Observations about bottlenecks and latencies were made from assembler listings and performance results provided by the IBM *spu_timing* tool in IBM Cell SDK 2.1. Code listings from this tool are provided throughout Section 6 of this paper in order to illustrate design decisions and performance results. For readability purposes assembler code is listed in a pseudo assembler format and commented with the equivalent C code listings.

```
Cycle Pipeline Schedule Pseudo Assembler       C Code
01    12                move   pc_next,pc_pref pc_next = pc_pref;
01    123456            load   op1,sp[-1]      op1 = sp[-1];
02     234567           load   op2,sp[-2]      op2 = sp[-2];
03      34              addi   sp,sp,-1        sp--;
03      345678          hint   LABEL,pc_next
04       456789         load   pc_pref,pc[2]   pc_pref = pc[2];
05        56            addi   pc,pc,1         pc++;
08         --89         add    op2,op2,op1     op2 = op2 + op1;
10            -012345   store  op2,sp[-1]      sp[-1] = op2;
                        LABEL:                 LABEL:
11              1234    branch pc_next         goto pc_next;
```

Figure 3: Pipeline schedule for VM instruction IADD with PC fetch pipelining

# 6 Optimizations

This section is split into five different sections each detailing a different set of optimization strategies. Section 6.1 investigates a technique first proposed by Hoogerbrugge *et al.* [6] called interpreter pipelining. Section 6.2 presents some unique techniques for the dispatch of JVM instructions. Section 6.3 introduces stack caching by Ertl [5]. Section 6.4 presents some code merging techniques and finally a set of superinstructions are investigated in Section 6.5. Performance results are presented for each optimization throughout the rest of this section.

## 6.1 Interpreter Pipelining

The instruction dispatch phase in the JVM instruction's life cycle has a dependence chain that includes the program counter increment (addition — 2 cycle latency), the program counter fetch (load from memory — 6 cycle latency) and the instruction dispatch (indirect branch — 4 cycle latency and 18 - 19 cycle mispredict penalty). The location of the program counter fetch can be moved around the interpreter to allow software pipelining of the JVM instructions execution cycle. By performing this rescheduling, the dependence chains that produce many of the latencies identified in the JVM Instruction IADD are broken (see Figure 1). Thus, the compiler is given room to produce improved scheduled code and the dispatch phase of the instruction cycle can be issued without a data dependant delay.

Hoogerbrugge *et al.* [6] presented a pipelined interpreter for the TriMedia VLIW Media Processor. They showed that their techniques could decrease VLIW instruction count by up to 19.4% and cycle count by up to 14.4%. The TriMedia VLIW Media Processor has an issue width of five and a jump latency of four cycles allowing fifteen operations to be scheduled in

three delayed slots after the jump and another four in parallel with the jump. Cell BE SPE has a different approach to branches. Its hinting scheme requires four instruction pairs and eleven cycles to be executed between a hint and a branch. This hint to branch latency requires the interpreter to access the dispatch value very early in the execution of a JVM instruction, pipelining the instruction dispatch phase enables this access.

In the code sample of Figure 1 the program counter for the next instruction is loaded immediately before the indirect branch dispatching the next instruction. This causes a large penalty due to memory latency and a further penalty for a mispredicted branch. The first instruction in Figure 3 defines the program counter for the dispatch phase of the current JVM instruction. The value is copied from a prefetch variable defined in the previous JVM instruction. This value is subsequently used to throw a correct hint to the dispatch instruction, *branch*. Figure 4 shows the speed-ups for this technique over all our benchmarks.

### 6.1.1 Operand Pipelining

To allow for efficient loading of bytecodes, the program's instruction stream is aligned to a 16 byte boundary on the Cell BE SPE local store. The instruction's bytecode is stored in the *'prefered slot'* (the second value in a 32 bit vector on which scalar operations are performed) of the 16 bytes while the operand is stored in the third slot of the vector. To access the operand the 16 bytes must be loaded into memory (6 cycles), a rotate instruction must align the third slot into the prefered slot (4 cycles) and for the purposes of the JVM instruction ILOAD, a shift instruction must scale the operand for offsetting the local variable array (4 cycles). This dependence chain of 14 cycles is illustrated in Figure 6.1 (a) where there are 4, 4 and 6 stalls preceding each dependence in this chain.

Operand pipelining removes this dependence in its entirety. Extending program counter pipelining the 16 bytes of the instruction stream are pre-loaded in registers removing the need to perform a load for the operand value. The rotate stage to align the operand is performed in the previous instruction. The benefits of this schedule are illustrated in Figure 6.1 (b) where the stall cycles have been removed and the total cycle count of ILOAD is reduced from 23 to 14.

The final optimization of the ILOAD instruction is to pre-scale the operand value in the code preparation stage of the JVM. As illustrated in Figure 6.1 (c) this allows the prefetched operand value to be immediately used to offset the local variable array and perform the load from memory of the necessary value.

Figure 4 graphs the results for operand fetch pipeling. As ILOAD is the most commonly

```
Cycle Pipeline Schedule     Pseudo Assembler       C Code
01    12                    move   pc_next,pc_pref pc_next = pc_pref;
02     234567               load   pc_cur,pc[0]    pc_cur = pc[0];
03      345678              load   pc_pref = pc[2] pc_pref = pc[2];
04       456789             hint   LABEL,pc_next
05        56                addi   pc,pc,1         pc++;
08         --8901           rotate op,pc_cur,8     op = pc_cur->opr;
12            ---2345       shift  op,op,4         op = op << 4;
16                ---678901 load   local,op,lvars  local = lvars[op];
22    234567          ----- store  local,sp[0]     sp[0] = local;
23     34                   addi   sp,sp,1         sp++;
                          LABEL:
23     3456                 branch pc_next         goto pc_next;
```

(a) ILOAD with Program Counter Pipelining

```
Cycle Pipeline Schedule  Pseudo Assembler           C Code
01    12                 move   pc_next,pc_pref     pc_next = pc_pref;
02     234567            load   pc_temp = pc[2]     pc_temp = pc[2];
03      3456             shift  opr,opr_fetch,4     opr = opr_fetch << 4;
04       4567            rotate opr_fetch,pc_temp,8 opr_fetch = pc_temp->opr;
05        56             addi   pc,pc,1             pc++;
06         678901        hint   LABEL,pc_next
07          789012       load   local,opr,lvars     local = lvars[opr];
08           89          move   pc_pref,pc_temp     pc_pref = pc_temp;
13           -----345678 store  local,sp[0]         sp[0] = local;
14                 45    addi   sp,sp,1             sp++;
                       LABEL:
14                 4567  branch pc_next             goto pc_next;
```

(b) ILOAD with Program Counter and Operand Pipelining}

```
Cycle Pipeline Schedule Pseudo Assembler           C Code
01    12                move   pc_next,pc_pref     pc_next = pc_pref;
02     234567           load   pc_temp = pc[2]     pc_temp = pc[2];
03      34              addi   pc,pc,1             pc++;
04       456789         load   local,opfetch,lvars local = lvars[opfetch];
05        567890        hint   LABEL,pc_next
06         6789         rotate opfetch,pc_temp,8   opfetch = pc_temp->opr;
08          -89         move   pc_pref,pc_temp     pc_pref = pc_temp;
10            -012345   store  local,sp[0]         sp[0] = local;
11              12      addi   sp,sp,1             sp++;
                      LABEL:
12               2345   branch pc_next             goto pc_next;
```

(C) ILOAD with Program Counter and Operand Pipelining removing the shift

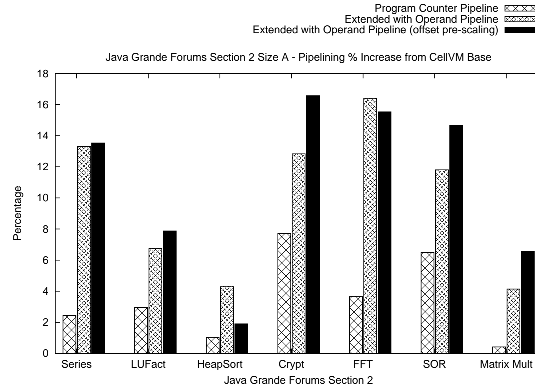


Figure 4: Pipelining speed increases (bigger is better)

executed JVM instruction [9] a substantial increase in speed is observed when operand pipelining optimizes both it and other instructions with an operand value.

## 6.2 JVM Branch Prediction

A number of JVM instructions compute their branch target from operands (e.g. JSR, GOTO) or from conditional statements comparing values from the operand stack (e.g. IFEQ, IF_ICMPLT). In these cases the pipeline chain fetching the program counter in the previous instruction is broken and an accurate hint can be issued only at a very late stage of execution. In this schedule the hint latency illustrated in Section 3.2 will reduce the effectiveness of a branch hint by introducing stalls to the issue of the indirect branch at the dispatch stage of execution.

If a prediction can be made earlier on in the execution cycle a saving of the full 18 - 19 cycles can be made. Regular processors have solved these very issues in their pipelines by issuing predictions using hardware implemented branch predictors. This project has implemented a simplified predictor commonly referred to as a one bit predictor inside the Java VM interpreter.

One bit predictors issue predictions based on a heuristic giving a single piece of information (e.g. direction of the last branch or the type of branch - loop, conditional statement). A number of different heuristics were analysed with the results graphed in Figure 5 (a), they include:

1. Taken - All branches are hinted as taken. The hint value is the operand value of the JVM instruction.

2. Not Taken - All branches are hinted as not taken. The hint value is the value of the next instruction in the instruction stream.

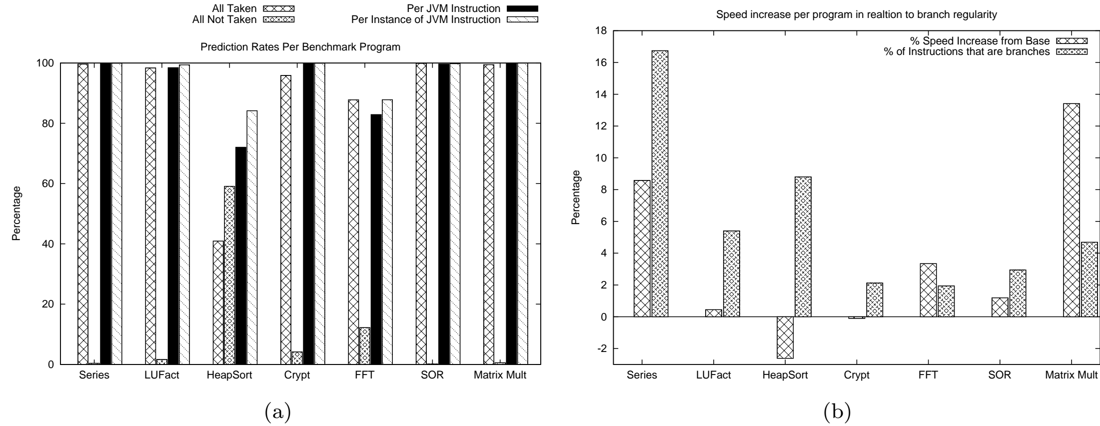


Figure 5: (a) Prediction rates over all branches for a given prediction technique. (b) 'Predict Per JVM Instruction': the left bar graphs the percentage increase in run timing for each benchmark. The right bar charts the percentage of instructions in a given benchmark that are branch instructions.

3. Predict Per JVM Instruction - Each JVM instruction has a register value which it uses to statically hint at the beginning of each instruction. On each execution the register value is updated with the target address computed for that iteration.

4. Predict Per Instance of JVM Instruction - Each instance of a JVM branch instruction in a given program stores its own hint value in an operand. Each iteration of the instruction updates the operand with the tartget address for the current instruction.

One bit predictors will achieve on average a 50% success rate in a program with completely random branch behaviour. However, most programs contain branches that control loop iterations and other flow control that are highly predictable. In our final two implementations, the branch at the end of a standard non-nested loop will be miss-predicted twice, in the first iteration and the final iteration. All other phases of the loop will be correctly predicted.

Both implementations for (3) and (4) achieve extremely similar high prediction rates on the Java Grande Section 2 benchmarks (see Figure 5 (a)). Because of the penalty the final implementation incurs accessing its second operand value (prefetching of two operand values is not implemented in interpreter pipelining) the performance results for it are lower than that of the first implementation. As a result, performance figures for the third heuristic are presented in Figure 5 (b) which charts the percentage of instructions that are branches alongside the percentage speed up achieved by the predictor. By examining the Heapsort and Matrix Multiplication benchmarks we can see a relation between the number of branches, the prediction rate and the performance of a program. A program with a large number of branches has a speed increase

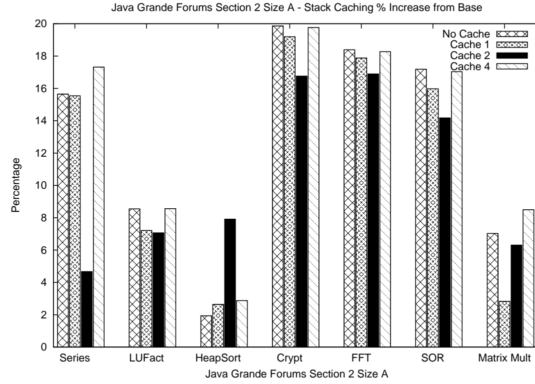


Figure 6: Stack caching speed increases

directly related to the predictability of the branches. JVM branch prediction has a negative impact on unpredictable programs like Heapsort. Heapsort has many branches and as seen in Figure 5 (a) these branches are less predictable than other benchmarks. The overhead taken to produce a prediction is larger than the improvement given by predicting branches correctly. We therefore have a negative speed up for heapsort. In Matrix Multiplication on the other hand we have few branches but as they are highly predictable we see a large speed increase.

### 6.3 Top Of Stack Caching

The discussion of Cell BE SPE's local store in Section 3.1 illustrates the expense of loading values from the stack into registers. Stack caching is an interpreter optimization designed to negate the cost of these loads and stores by reducing the number required. This is achieved by storing (caching) the top few values of the stack in registers. Ertl [5] presented two techniques for stack caching. The first, dynamic, requires the interpreter to keep track of its state and replicate instructions for each possible state. This technique results in code explosion as each instruction requires a number of different replicated versions. The second, static, requires the stack caching level to be defined at compile time. Since the local store of the SPE is limited (256kB) and is used for both code and data we choose the static design for our interpreter. Three different levels of caching were measured – the top one, two and four variables of the stack.

Figure 7 illustrates the effect of stack caching one variable on the cycle count and pipeline stalls for the JVM instruction IADD. In Figure, 1 preceding the addition of the two values are two load instructions. These loads move operand values from the stack into registers. A further six cycles are then required before issuing the addition. In total ten cycles pass before the addition

```
Cycle  Pipeline Schedule  Pseudo Assembler       C Code
01     12                 move   pc_next,pc_pref pc_next = pc_pref;
01     1234               load   op2,sp[-1]      op2 = sp[-1];
02      23                move   op1,top1        op1 = top1;
02      234567            load   pc_temp,pc[2]   pc_temp = pc[2];
03       34               addi   sp,sp,-1        sp--;
03       345678           hint   LABEL,pc_next
04        45              addi   pc,pc,1         pc++;
04        4567            rotate op,pc_pref,8    op = pc_pref->opr;
07         --78           add    op1,op1,op2     op1 = op1 + op2;
08            89          move   op,opfetch      op = opfetch;
09             90         move   top1,op1        top1 = op1;
09             9012       move   pc_pref,pc_temp pc_pref = pc_temp;
                          LABEL:
10              0123      branch  pc_next        goto pc_next;
```

Figure 7: A profile of instruction IADD with stack caching of one variable

instruction is executed. In Figure 7 the top value of the stack is cached in a register – *top1*. The second value is loaded from the top of the stack in local store. The result of this addition is stored back into the top of stack register, *top1*, removing the need for a further store instruction to push the value back onto the top of stack.

In total the stack cached IADD operation takes 10 cycles compared to 21 in the original IADD. This reduction in cycles causes scheduling of the hint to become more difficult for the compiler. There are fewer cycles to schedule a hint in and the required approximation of 15 cycles for a hint to branch are not present. The effect of this scheduling challenge on run time is graphed in Figure 6. For the majority of benchmarks stack caching produces a small variance in performance. The variance in time depends on the level of caching performed. For example, caching one value reduces the instruction count and the number of cycles available for scheduling hints. This produces a less efficient hint schedule and hence a slight decrease in runnning time over many benchmarks. In comparison caching four variables increases the instruction count as the interpreter has an increased number of varaibles to manage. This is the most effective caching level across all our benchmarks as the increased instruction count allows for more efficient scheduling of the branch to hint.

## 6.4 Merged Operators

The bottleneck of execution for the majority of JVM instructions in the pipelined interpreter presented in Section 6.1 is the hint to branch latency. This latency requires 4 instruction pairs and 11 cycles of processing between a hint and a corresponding branch. This bottleneck allows

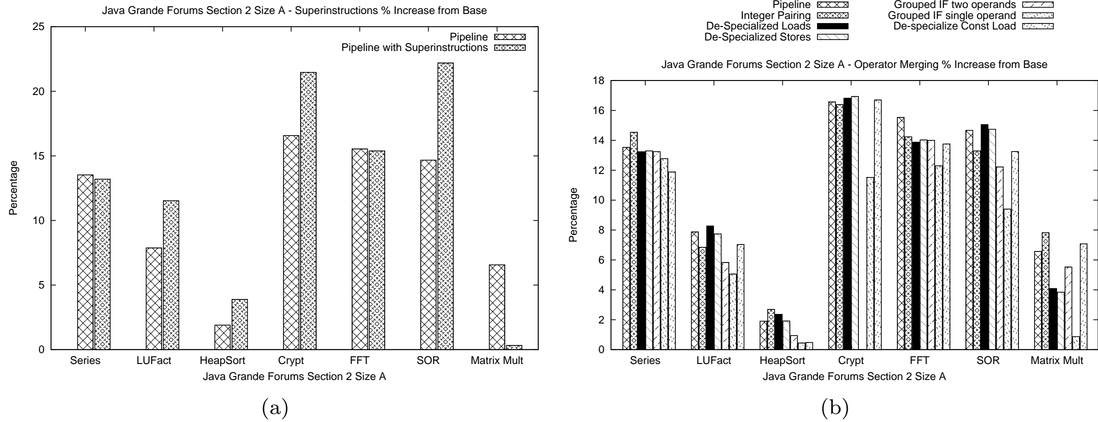


Figure 8: Superinstructions speed increases: pairs include combinations of integer load and stores, constant loads and integer operations and their de-specialized and merged equivalents.

for more work to be performed in this *stall window* than otherwise possible. A traditional way to increase the work load of instructions per dispatch, as proposed here, is to introduce superinstructions. Superinstructions are groups of the most common instructions concatenated together to form a larger instruction that can execute multiple instructions with a single dispatch. This process increases the code size as extra instructions are added.

We observe that an alternative processes can be performed where-by different JVM instructions with similar characteristics in operand fetching and instruction dispatch are co-located in a single interpreter case. This process has two distinct advantages:

1. JVM Instruction Reduction: The number of instructions requiring individual implementations in the interpreter is reduced. For example, arithmetic instructions like *iadd* and *isub* can be grouped together, reducing the number of arithmetic operation implementations required from 14 to 8.

2. Options for Superinstructions: The number of schedules that can take advantage of superinstructions is increased as options become more widely available. With the merging of instruction implementations sequences of these instructions become more common.

As the latency costs of loading and operating on JVM operands has been removed from the interpreter with our pipline techniques, the merged and de-specialized instructions tested in our implementation have shown minor changes in performance across all benchmarks (see Figure 8). The following subsections discuss the individual groupings tested in our implementation.

```
Cycle Pipeline Schedule  Pseudo Assembler          C Code
01    12                 move    pc_next,pc_pref   pc_next = pc_pref;
01    123456             load    op1,sp[-1]        op1 = sp[-1];
02     23                ceqi    cond,opfetch,96   cond = (opfetch == 96);
02     234567            load    op2,sp[-2]        op2 = sp[-2];
03      34               addi    sp,sp,-1          sp--;
03      345678           hint    LABEL,pc_next
04       4567            rotate  opfetch,pc_pref,8 opfetch = pc_pref->op;
05        567890         load    pc_pref,pc[2]     pc_pref = pc[2];
06         67            addi    pc,pc,1           pc++;
08          -89          add     r1,op1,op2        r1 = op1 + op2;
09            90         sub     r2,op1,op2        r2 = op1 - op2;
11             -12       select  r1,r1,r2,cond     r1 = cond ? r1 : r2;
13               -345678 store   r1,sp[-1]         r1 = sp[-1];
                         LABEL:
14                  4567 branch  pc_next           goto pc_next;
```

Figure 9: A profile of merged-instruction IADD_ISUB with program counter and operand pipelining

### 6.4.1 Integer Operation Pairing

There are 14 integer operations in the Java virtual machine. Figure 9 is an example of one of these pairings. Here the execution phase of the instruction calculates both the integer addition and subtraction result for the two operands on the top of stack. A single specialized instruction from the SPE Instruction Set Architecture allows a conditional assignment to store the desired result back onto the top of stack. The condition for this assignment is the operand value of the virtual machine instruction. This is set to either the opcode value for IADD or ISUB depending on what operation is required.

### 6.4.2 Conditional Instruction Grouping

A similar scheme to integer operation pairing can be used to group conditional statements together. Each conditional instruction firstly reads either one or two operands from the operand stack. In the case of one operand, it is compared to zero using an operator specific to the operation. Similarly for two operands, the operands are compared to each other using a specific operation for that VM instruction. Next, the condition is examined to be true or false and assignment to a program counter is performed, either from the operand of the instruction (the target) when the condition is true, or the next program counter in the instruction stream (the fall through path) when the condition is false.

In total there are twelve conditional VM instructions. This grouping technique forms these JVM instructions into two groups, one for single stack value operand instructions, the other for

double stack value operand instructions. For each of the six possible conditions to be compared a result for that comparison is defined. To choose the correct condition a series of conditional assignments using the second operand to the VM instruction as a condition. This second operand is an integer representing the instructions byte code value. This series of conditional assignments will result in a correct condition value. The program counter can now be defined as in a standard implementation.

### 6.4.3 De-specializing LOAD / STORES

A common technique for reducing the total number of JVM instructions is to reduce the number of 'specialized' JVM instructions. Specialized instructions are instructions that have had their operand value incorporated into the VM instruction, for example ILOAD_0 has had its operand '0' defined as part of the instruction. De-specializing is the process of removing these types of instructions and replacing them with an equivalent VM instruction and its correct operand. In total there are eight integer load and stores with a number of long and double loads and stores too. For every different group we can reduce the number of JVM instructions required for it down to one.

### 6.4.4 Constant Load Grouping

In total there are fourteen instructions in the CellVM interpreter that simply load a constant value to the top of stack. Ten of these fourteen instructions are specialized instructions, for example the VM instruction ICONST_0 loads the constant value 0 to the top of the stack. The others load the constant value defined in their operand to the top of stack. Constant load grouping removes all these instructions and replaces them with a single instruction that loads a constant defined by the JVM instructions operand to the top of stack. The specialized instructions in the group are de-specialized and their load values are defined in their operand.

## 6.5 Superinstructions

Superinstructions are implementations of concatenated Java instructions. For example, an instruction sequence of an ILOAD followed by an IADD could be concatenated into the single instruction ILOAD_IADD. The compiler of the interpreter can now perform more powerful optmizations, in the case of the ILOAD_IADD superinstruction, two stack operations can be optimized away as the local variable accessed in the ILOAD instruction can be accessed within the instruction.

Further a whole instuction dispatch is made redundant as the two instructions now form a single implementation in the interpreter.

Casey *et al.* [3] have shown the benefits of superinstructions and also demonstrated how the elimination of specialized instructions makes superinstructions more applicable. Our interpreter has taken a modified approach by merging unrelated instructions and hence creating an even more common pool of instructions to choose from. To demonstrate the performance of superinstructions on our interpreter a number of instructions were grouped in pairs as superinstructions (see Figure 8(a))). Matrix multiplication achieved a negative speed up from our pipelined version, this negative speed is a result of the combined impact of negative speed up from the merged instructions in Figure 8(b). Merging of conditional instructions and de-specialized loads and stores have had a greater impact on matrix multiplication than the savings found through the superinstructions implemented that were applicable to matrix multiplication. Other benchmarks such as *Crypt* and *LU Factorization* achieve in the order of 5-7% increase over the pipelined interpreter, giving a total performance increase of approximately 23% for both benchmarks.

# 7 Related Work

The most closely related work to ours is from Hoogerbrugge and Augusteijn who developed a Java interpreter for the Philips Trimedia VLIW processor [6]. This processor uses delayed branches, which are analogous to the hinted branches found in the Cell BE SPE architecture. Their interpreter prefetches operands up to two instructions ahead to break the dependencies between fetching and dispatching the next instruction. They also use static stack caching to eliminate memory traffic to the JVM operand stack. Overall, their optimizations yielded a speedup of 19% over a simple interpreter that uses a switch statement to dispatch JVM instructions.

Ogata *et al.* [12] addressed the problem of bytecode prefetching for the IBM POWER3 processor. This is a conventional out-of-order superscalar processor, except that rather than predicting indirect branches, it uses a special branch target register. If the target is placed in the branch target register sufficiently early, then indirect branches can be executed without pipeline stalls. This is also analogous to the branch hints found in the Cell processor. Ogata *et al.* use opcode prefetching to make the target available earlier, and replicate the interpreter to allow multiple bytecodes to be loaded using a single 4-byte load instruction.

Ertl [5] proposed a large number of variations of stack caching, including the static stack caching scheme used in this paper. Although Ertl provided statistics on the number of memory

accesses eliminated by using stack caching, no running times were provided. Stack caching was subsequently implemented in various JVM interpreters including Sun's Hotspot VM, Cacao VM, and Jam VM.

Casey *et al.* [3] investigated the use of superinstructions in a Java interpreter as one way to reduce indirect branch mispredictions on a conventional processor. They eliminated specialized versions local variable load and store JVM instructions, as well as specialized constant load JVM instructions to make superinstructions more widely applicable. They did not, however, attempt to merge unrelated JVM instructions as we have done in Section 6.4.

# 8 Conclusions

This paper has presented analysis of existing interpreter optimization techniques on the Cell BE Processor and introduced novel optimizations made possible by the architectural features of the Cell BE SPE. Firstly, interpreter pipeline techniques first published by Hoogerbrugge *et al.* and tested on the Trimedia VLIW processor were tested and shown to provide increases in performance on the Cell BE. Further, these techniques were extended by introducing an optimization referred to as operand pipelining in which the existing pipeline structure was extended by performing a decode stage in which operands were fetched in advance. This technique proved extremely successful providing a speed increase of just under 20% on our best performing benchmark. Secondly, a novel optimization for the dispatch of conditional JVM instructions was described. This technique used the hint feature of the Cell BE to improve the dispatch performance of JVM instructions that compute their control flow target. Finally, the limitations of the Cell BE's abilities to interpret JVM bytecodes efficiently were identified. They include the large hint to branch stall (approximately 15 cycles) and the penalty incurred by expensive load and stores (6 cycles). The opportunities of the first limitation were demonstrated through a useful technique that eliminated the number of JVM instruction implementations required by performing a merging and pairing optimization co-locating similar instructions in the same implementation. These merged instructions were subsequently used to provide a rich environment for producing superinstructions combinations that produced an increase in speed over many benchmarks. Through the use of software pipeling an efficient pipelined interpreter was designed that removed the load and store limitation of program counter fetch and decode as well as prefetching operands.

## References


[1] J. R. Bell. Threaded Code. *Commun. ACM*, 16(6):370–372, 1973.

[2] B. Brock and M. Exerman. Cache Latencies of the PowerPC MPC7451. *Freescale Semiconductor Application Note*, AN2180, 2006.

[3] K. Casey, M. A. Ertl, and D. Gregg. Optimizing Indirect Branch Prediction Accuracy in Virtual Machine Interpreters. *ACM Trans. Program. Lang. Syst.*, 29(6):37, 2007.

[4] A. E. Eichenberger, K. O'Brien, K. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, and M. Gschwind. Optimizing Compiler for the Cell Processor. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 161–172, Washington, DC, USA, 2005. IEEE Computer Society.

[5] M. A. Ertl. Stack Caching for Interpreters. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 315–327, New York, NY, USA, 1995. ACM.

[6] J. Hoogerbrugge and L. Augusteijn. Pipelined Java Virtual Machine Interpreters. *9th International Conference, CC 2000*, 1781/2000:35–49, 2000.

[7] IBM. Cell Broadband Engine Programming Handbook, 4 2007.

[8] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell Multiprocessor. *IBM Journal of Research and Development*, 49(4/5), 2005.

[9] I. H. Kazi, H. H. Chen, B. Stanley, and D. J. Lilja. Techniques for Obtaining High Performance in Java Programs. *ACM Comput. Surv.*, 32(3):213–240, 2000.

[10] T. Lindholm and F. Yellin. *Java Virtual Machine Specification.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[11] A. Noll, A. Gal, and M. Franz. CellVM: A Homogeneous Virtual Machine Runtime System for a Heterogeneous Single-Chip Multiprocessor, 2006.

[12] K. Ogata, H. Komatsu, and T. Nakatani. Bytecode Fetch Optimization for a Java Interpreter. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 58–67, New York, NY, USA, 2002. ACM.

[13] A. K. Uht, V. Sindagi, and S. Somanathan. Branch Effect Reduction Techniques. *Computer*, 30(5):71–81, 1997.

[14] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The Potential of the Cell Processor for Scientific Computing. In *CF '06: Proceedings of the 3rd Conference on Computing frontiers*, pages 9–20, New York, NY, USA, 2006. ACM.