Hardware compilation for high performance Java processors

G. HARIPRAKASH[1], R. ACHUTHARAMAN[1], AMOS R. OMONDI[2]
School of Computer Engineering,
Nanyang Technological University, Singapore.
[1]{Hariprakash.govindarajalu, Achutharaman.rangachari}@sun.com,
[2]ASAmos@ntu.edu.sg

Abstract

High performance on Java applications running on server and desktop machines requires fast execution of Java bytecodes. Such performance can be achieved by Just-In-Time (JIT) compilers, which translate the stack-based bytecodes into register-based machine code on demand. But one crucial problem in Java JIT compilation is the compilation time, which increases the total execution time of an application. So it is necessary to reduce the JIT compilation time as much as possible. In this paper we propose a front-end hardware compilation pipeline that performs the compilation of bytecodes into native machine code on-the-fly in hardware and pass the compiled code to a backend native processor for execution. The bytecodes are translated into three-address intermediate representation form, by mimicking the stack operations, before performing a series of optimizations in hardware. The optimized three-address codes are used for code generation and architectural register allocation and then placed in a cache for execution by the backend native processor. A micro-architecture of hardware compilation pipeline is presented.

1 Introduction

The Java bytecodes are generated for a stack machine, and the Java Virtual Machine (JVM) is responsible for the execution of bytecodes by interpreting them into native machine codes, or directly executing them on a hardware Java processor. Executing Java bytecodes over the JVM layer on a native platform is many times slower than the native code equivalent directly running on the same platform.

Server and desktop Java applications demand high performance and currently rely on software JVM Just-In-Time (JIT) compilers to execute the Java bytecode equivalent on a native processor. The JIT compilation time itself is a part of the applications execution time and contributes to slowness. It is therefore important to reduce the compilation time as much as possible for applications that require high performance. Hardware JVM implementations have been shown to perform several times faster than the software JVM implementations[7]. However, the current implementations of JVM in hardware, known as Java processors, do not provide high the performance demanded by server and desktop applications, and they are mostly targeted for embedded applications. This requires a trade-off between the compilation-time and execution-time of high-performance applications, which poses a challenging research problem in Java bytecode execution.

This paper introduces a hardware compilation pipeline, a front-end pipeline that takes Java bytecodes as input and generates optimised native code on-the-fly for the backend native processor. The bytecodes are translated into an intermediate representation form followed by a series of simple optimisations in hardware and then the generation of native code. The instruction-fetch unit of the native processor for execution then fetches the native code. There are three main stages in our pipeline:- translation stage, optimisation stages, and code generation stage. The translation stage converts the decoded Java bytecodes into three-address intermediate representation. Next, a series of simple machine-independent and machine-dependent optimisations are performed. The machine-independent optimisation includes common sub-expression elimination, copy propagation, constant propagation, peephole optimisations, and strength reduction. The machine-dependent optimisations include register allocation, placement of software prefetch instructions in the native code, method invocation based on calling convention of the native processor, delay-slot code scheduling etc., The optimisation algorithms are carefully chosen to be linear in space and time, so that it is feasible for implementation in hardware. The unit of fetching and compiling by the front-end pipeline is limited to a basic block of Java bytecodes, due to limitation on hardware resources as

well as timing constraints imposed by on-the-fly compilation. The front-end pipeline relies on the native processor for null pointer checking, array bound checking, and divide-by-zero exception handling.
The rest of the paper is organized as follows. Section 2 briefly reviews the related work on Java Virtual Machine implementation, in software and in hardware, and the motivation for the hardware compilation pipeline. A detailed description of the proposed hardware compilation pipeline architecture is given in Section 3. Section 4 presents the simulation environment, and Section 5 concludes with the future work.

2 Related Works

The JVM specification is flexible and permits either software or hardware implementation of the JVM, provided the execution of the bytecodes follows the Java specification [13]. The Fig. 1 shows the various ways in which the JVM may be implemented.

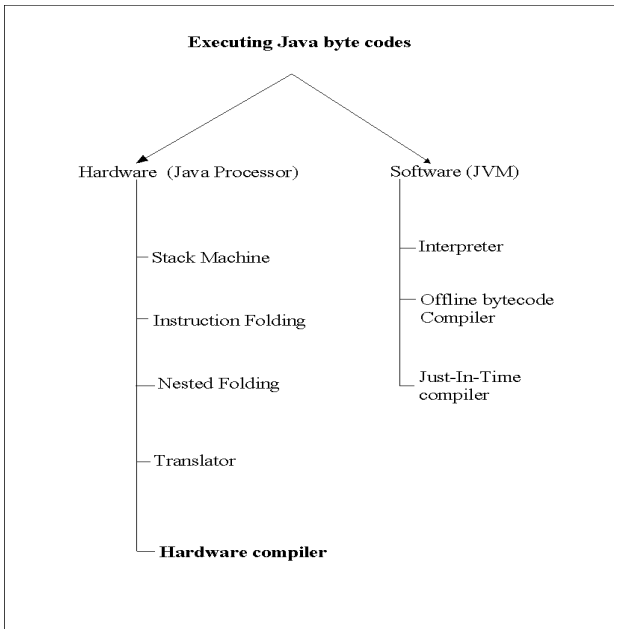

Fig. 1. Executing Java bytecodes.

2.1 Software approaches to JVM implementation

2.1.1 Interpreter

The simplest form of JVM implementation on a native platform is as an interpreter program that understands the Java bytecodes and simulates their execution on the native platform, e.g the Kaffe interpreter [15]. The Java compiler generates bytecodes for JVM that is essentially a stack machine. Writing interpreter for the code generated for a stack machine is simple, however the interpreters are severely affected by the performance due to variables need to be pushed into the stack for performing operation on them. A study in [3] reports that 48.3% of bytecodes are data-movement instructions. Hence most of the interpreter's time is spent on interpreting such instructions. Even with the latest advanced superscalar processors, the interpretation speed does not meet the performance requirements of many Java applications.

2.1.2 Offline bytecode compiler

The static compilation of Java bytecodes into native instructions can be performed prior to execution; examples of this approach are J2C[16] and Turbo J [17]. The advantage of this approach is that sophisticated code generation and optimisation techniques can be applied. However, the platform-independence feature of Java is lost when programs are compiled to native instructions. Hence this approach is not suited for network computing environment, but for homogeneous environment.

2.1.3 Just-In-Time (JIT) compiler

JIT compilers compile a Java method into native instructions on-the-fly and then cache the native code for future reference; examples of this approach are HotSpot[18], Kaffe JIT[15], IBM JIT[10] and Latte JIT[9]. The performance gain here comes from directly executing the native code on a processor, instead of simulating the bytecodes in software. The compilation itself should be fast, since the compilation time adds up to the total execution time of Java application. In order to cut down the compilation time, extensive optimisations are not performed, and the JIT compilers are limited to simple and fast optimisation algorithms.

2.2 Hardware approaches designing Java processors

The hardware implementations of JVM which are known as Java processors, perform better than the software JVM implementations, because the bytecodes are directly executed as native instructions in hardware. There are two ways of implementing Java processors; stack-based Java processors and bytecode-to-native translator or compiler Java processor pipelines.

2.2.1 Stack processor

The JVM is a virtual stack machine, and implementing it in hardware is straightforward if the JVM instruction set

forms the native language of the stack processor. Registers are not used; instead, a high-speed hardware stack is implemented over a set of registers that acts as a stack. The pipeline of a typical stack processor is similar to that of a conventional register machine, except that the register-access stage is replaced by a stack-access stage and the registers are replaced by a stack [1] [8]. The main disadvantage of the code running on a stack machine, relative to a register machine, is that, because of stack dependency, less instruction level parallelism can be exploited. Also, more data movement operations are needed to move the data from the local variables to the stack for the execution of an instruction.

2.2.2 Instruction Folding

A group of Java bytecodes can be combined into a single operation and then issued by a Java processor for execution, a process known as folding. Contiguous Java bytecode instructions that have true data dependencies are grouped into a single compound instruction. This is done by appropriate folding hardware. Sun Microsystems introduced such a folding stage in its picoJava-II processor pipeline [1].

2.2.3 Nested Folding

An improved version of folding, which can detect nested folding patterns, complex state machine that can recognize the such patterns and then issue folded instructions, is proposed in [4].

2.2.4 Translator

When the bytecodes are translated into two-address or three-address instructions and executed by the RISC pipeline, the performance will be higher than the folding stack processor or JIT compiler running on a native processor [2]. This approach has the advantage of eliminating the need to design a complete processor from scratch. Also, legacy applications and the operating-system support already existing for the native processor is readily available, with the additional capability of supporting Java language directly in hardware. Such a translation processor was proposed in [2], the register allocation and the optimisations are not emphasized. JEDI technologies JSTAR accelerator and FaRM Java co-processor are translators which share the resources of the back-end native processor.

2.2.5 Hardware compiler

In [5] - the authors proposes an instruction-level-parallel architecture for Java processor in which the hardware translates the bytecodes to RISC-like primitives during the instruction cache miss or a page miss, stores these primitives in a cache or memory, and later optimises and issues them to a superscalar pipeline. This is similar to the AMD K5 processor, which translates the x86 instructions to RISC primitives and then issues them out-of-order. The optimisation and scheduling are carried out in software by JIT. A complete hardware compilation in hardware is feasible, as mentioned in many papers [6] [5], but there no known implementation or a complete design of such a Java processor exists.

2.3 Motivation

The code generated for the JVM lacks many possible optimisations, because of the postfix representation form of bytecodes. The virtual stack dependency created by the stack operations does not allow the exploitation of instruction-level-parallelism available in the code. The JIT compilers depend on simple and fast algorithms for optimisation and code generation to cut down the compilation time. Typically, a JIT compiler that optimises the bytecode has to access various internal data structures of the JIT compiler to generate the native code; this will take hundreds of processor cycles; however, these algorithms when implemented in hardware will consume only a few processor cycles to generate native code but will boost the performance of bytecode execution in hardware. Most of the compilation algorithms are fairly simple and can be implemented in hardware. The postfix representation of bytecodes can be easily translated into three-address representation by the use of semantic stack (known as the Mimic Stack) [9] [11]. Most of the optimisations on the three-address intermediate representation form require searching for a pattern. The searching can be performed in hardware, by an associative search operation, on the three-address intermediate representations stored in a content-addressable memory (CAM). The searching operation when performed in hardware takes O(n) due to parallel search possible in CAM, but when the same search is performed in software it takes $O(n^2)$, because of the sequential search. The CAM size plays an important role in the hardware optimisations. To determine the approximate size of CAM we studied the average size of basic blocks in SpecJVM98 applications. This revealed that 90% of the basic blocks are less than 32 bytecodes. All of the above points and the need for a processor which can execute the legacy code as well as Java bytecodes motivated us to design a high performance Java pipeline which compiles the bytecodes into native code on-the-fly

3 Architecture of Hardware compilation pipeline

The translation of Java bytecodes to native codes in hardware can accelerate the compilation time by reducing from few hundreds to few tenths of cycles. The main reasons for the speedup are the possible optimisations and their implementation in hardware. The architecture of a hardware-compilation pipeline requires very little software support and greatly differs from the conventional pipeline, since the purpose of the pipeline is to compile the bytecodes rather than execute them. The compilation pipeline acts as a front-end pipeline that compiles the bytecodes into native code and passes the result it to a native backend pipeline.

3.1 Pipeline Architecture

The proposed hardware compilation pipeline essentially consists of five stages, as shown in Fig. 2.

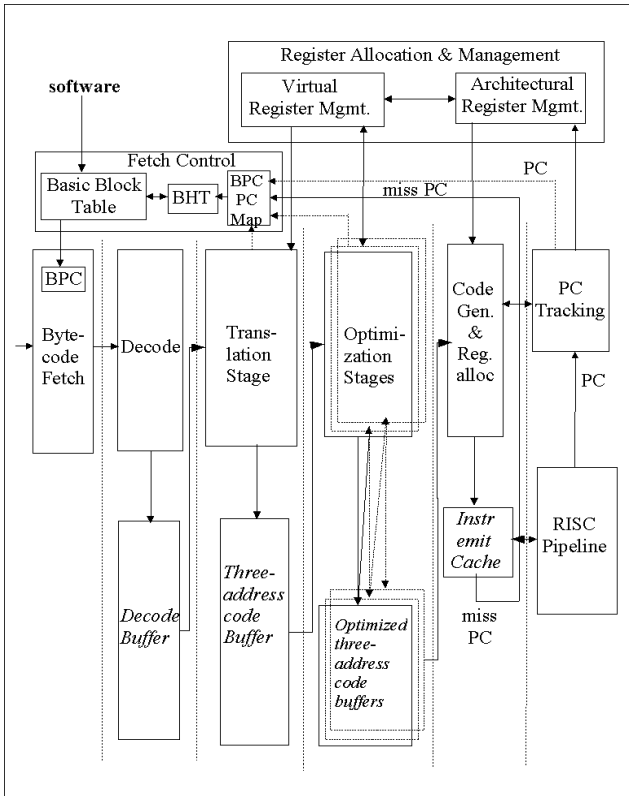

Fig. 2. Architecture of hardware compilation pipeline.

After initial bytecode-fetch and decode, the decoded bytecodes are translated into an intermediate representation form that facilitates various optimisations. The translation stage allocates virtual registers when generating the intermediate code. These virtual registers are latter mapped into architectural registers in the code generation and register allocation stage. The optimisation stage is a series of shorter stages that perform various machine-independent and machine-dependent optimisations on the intermediate representation before generating the native code. The optimisation stages follow one after the other in an order that will benefit the following optimisation stages. For example, the common sub-expression elimination (CSE) optimisation, introduces more temporary variables while eliminating the sub-expressions. Performing the copy-propagation optimisation following CSE removes the unnecessary move instructions generated during CSE optimisation. The generated native codes are then passed to the backend native processor for execution. Runtime informations, such as execution PC and branch prediction information, are collected from the backend native pipeline and used for the efficient compilation of the bytecodes into their native-code equivalent.

3.2 Micro-architecture of pipeline stages

3.2.1 Fetch stage

In order to perform compilation, a basic block is fetched first. The software collects the basic-block information, such as start and end bytecode PC and the information of the following basic block, from a method and stores it in the Basic Block Table (BBT) of the fetch control logic, as shown in Fig. 3.

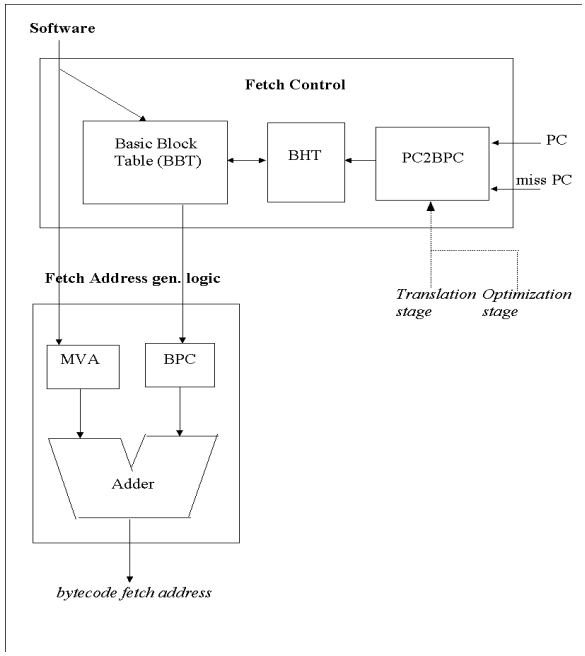

Fig. 3. Fetch stage.

If the last bytecode in the basic block happens to be a branch bytecode, then there are two possible following

basic blocks. Java ISA does not allow static branch prediction information to be embedded in the branch bytecode instruction, which can cause the miss penalty to be high unless if nothing is done about it. Furthermore, our front-end pipeline increases the backend pipeline depth by additional stages for processing the Java bytecodes. In order to reduce the prediction miss penalty, we construct the branch history information in BHT by tracking the native PC from the backend pipeline, using a native PC-to-bytecode PC mapping table (PC2BPC). Using this information, only the potential basic block following the current is fetched for hardware compilation. When there is no branch history information available, both paths of the branch instruction are fetched for compilation. The method is loaded in the virtual memory by the class loader and the native virtual address required to access the method's code is stored in the MVA register (Method Virtual Address) of fetch-address generation logic. The bytecode address within a method pointed by BPC (Bytecode PC) register, starts from zero, pointed by BPC (Bytecode PC) register, of the fetch address generation logic. The MVA register contents added with the BPC register contents forms the bytecode fetch address, as shown in the fetch address generation logic.

3.2.2 Decode stage

The size of a JVM instruction varies from one byte to several bytes; this complicates the decoding logic and complex bytecode also need software support in decoding. The decoding stage components are shown in Fig. 4.

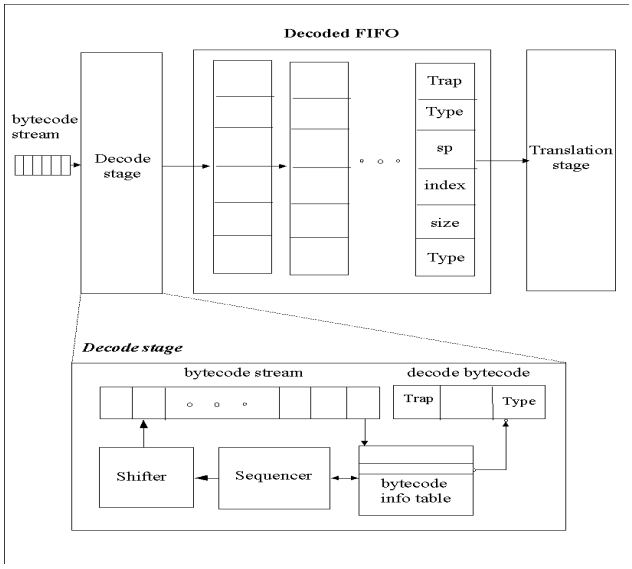

Fig. 4. Decode stage.

The input bytecode stream is stored in a shift register for decoding. The first bytecode in the stream is used to index into the bytecode information table, to access the attributes associated with that bytecode. The attributes includes the type of operation, the size of the bytecode, how the stack will be modified by this bytecode, whether trap into software is needed for this bytecode, the index into local variable and constant pool available in the bytecode stream, object reference, etc. Using the size information, the input bytecode stream is shifted to access the next bytecode. The decoded bytecodes are passed for translation through a decode FIFO buffer.

3.2.3 Translation stage

The Translation consist of four major components:- the Mimic Stack Manager (MSM), the Mimic Stack, virtual register allocator, and the three-address code generator, these are shown in the Fig. 5.

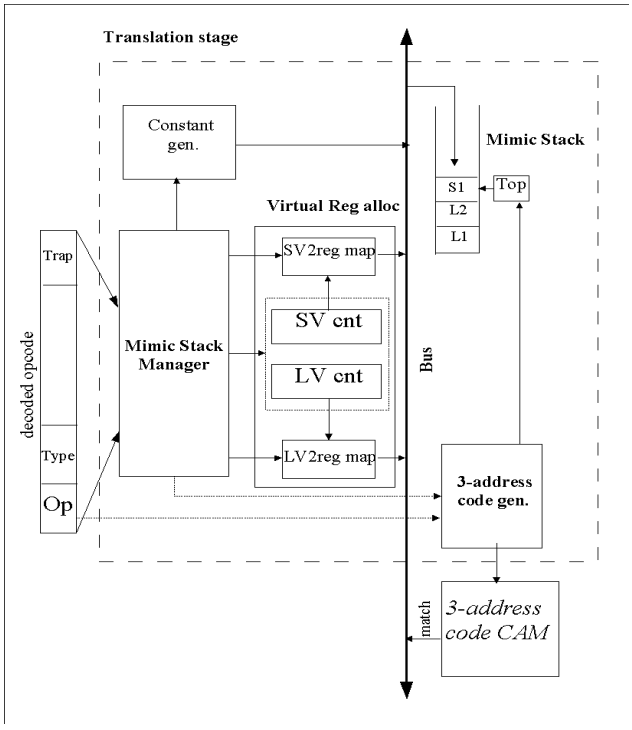

Fig. 5. Bytecode to three-address code translation stage.

The translation stage uses the mimic stack to transform the bytecodes to three-address intermediate representation. The main difference between the mimic stack and the Java processor stack is that in the mimic stack register numbers (virtual or symbolic registers), rather than the contents of the registers are used. The MSM coordinates the translation process by communicating with the mimic stack, virtual register allocator, and the three-address code generator. The MSM uses the decoded Java bytecodes and its attributes

to mimic the operation on the mimic stack. Two types of virtual registers are allocated based on temporary stack variable or local variables are allocated using the counters SVcnt and LVcnt, respectively. Whenever the same variable is referenced again in the bytecodes, the mapping table returns the previously allocated virtual register number. Once the mimic stack manipulation is over, the three-address code generator pops the mimic stack contents and the three-address code is generated.

Following the translation stage, the three-address code is optimised. The first optimisation performed is common sub-expression elimination since the postfix representation of bytecodes does not eliminate common sub-expressions. This optimisation can be performed in the translation stage itself by using a Content Addressable Memory (CAM) to store the three-address codes. By doing a CAM search before adding an entry on the CAM could detect the common sub-expressions. The result register of a matching CAM entry is pushed again into the mimic stack for to allow reuse of the result of the sub-expression, and the code for the redundant sub-expression is deleted. Fig. 6 shows a sample Java program with common sub-expressions and Fig. 7 shows the equivalent bytecode, the mimic stack manipulation by the translation stage, and the native code generated by the common sub-expression elimination optimisation.

```
class dag {
    public static void main(String[] args) {
      int a=5,b=6,c=7,d=8,e=9;
      a = (a+b) + (((a+b)-c)*(d+e)) * ((((a+b)-c)*(d+e))-e);
           ------      ------------------      --------------------
    }
  }
```

Fig. 6. Java code with sub-expressions

```
Bytecode          Native code       Mimic Stack
---------- int a=5,b=6,c=7,d=8,e=9; ----------
0 iconst_5                          5
1 istore_1        mov 5, r1
2 bipush 6
4 istore_2        mov 6, r2         6
5 bipush 7
7 istore_3        mov 7, r3         7
8 bipush 8
10 istore 4       mov 8, r4         8
12 bipush 9
14 istore 5       mov 9, r5         9
---------- (a+b) ------------------
16 iload_1                          r1
17 iload_2                          r1, r2
18 iadd           add r1,r2,t1      t1
----------  (((a+b)-c)*(d+e)) ------------
19 iload_1                          t1, r1
20 iload_2                          t1, r1, r2
21 iadd                       t1, r1, r2, + ←CAM Match
                                    t1, t1(pushed)
22 iload_3                          t1, t1, r3
23 isub           sub t1,r3,t2      t1, t1, r3
                                    t1, t2
24 iload 4                          t1, t2, r4,
26 iload 5                          t1, t2, r4, r5
28 iadd           add r4,r5,t3      t1, t2, t3

29 imul           mul t2,t3,t4      t1, t4
----------  (((a+b)-c)*(d+e))-e -----------
30 iload_1                          t1, t4, r1
31 iload_2                          t1, t4, r1, r2
32 iadd                    t1, t4, r1, r2, + ←CAM Match
                                    t1, t4, t1(pushed)
33 iload_3                          t1, t4, t1, r3
34 isub                    t1, t4, t1, r3, - ←CAM Match
                                    t1, t4, t2(pushed)
35 iload 4                          t1, t4, t2, r4
37 iload 5                          t1, t4, t2, r4, r5
39 iadd                 t1, t4, t2, r4, r5, + ←CAM Match
                                    t1, t4, t2, t3(pushed)
40 imul                   t1, t4, t2, t3, * ←CAM Match
                                    t1, t4, t4(pushed)
41 iload 5                          t1, t4, t4, r5
43 isub           sub t4,r5,t5      t1, t4, t5
-----  (((a+b)-c)*(d+e)) * ((((a+b)-c)*(d+e))-e) -----------
44 imul           mul t4, t5, t6    t1, t6
--- (a+b) + (((a+b)-c)*(d+e)) * ((((a+b)-c)*(d+e))-e) ------
 45 iadd          add t1, t6, t7    t7
-a = (a+b)+ (((a+b)-c)*(d+e)) * ((((a+b)-c)*(d+e))-e)-----
46 istore_1       mov t7, r1        EMPTY STACK
447 return
```

Fig. 7. Common Sub-expression Elimination during three-address code generation.

When the three-address code generated for a basic block exceeds the size of the CAM, the CSE optimisation is limited to the CAM size. The optimisation is not performed for common sub-expressions across a span that exceeds the CAM size, and partially optimised three address codes are generated in such cases. A study revealed that 90% of the basic blocks are less than 32 bytecodes, and so a 32 or 64 entry CAM is sufficient at translation stage.

The three-address code CAM is shown in the Fig. 8. Its operation is as follows the instruction and the source operands are checked for a match. Commutative operations, such as multiplication and addition, allows the operands to be interchangeable. In order to handle the commutative operations, the search is performed once or

twice - first with the given operand order and then again if the first search does not match in the CAM and the operator has commutative property with the order of operands reversed. To perform the CSE optimisation a search against all preceding three-address codes is required; this is accelerated by hardware through parallel search and consumes only one cycle. Additional logic to detect if the result register is modified in-between is not shown in the figure.

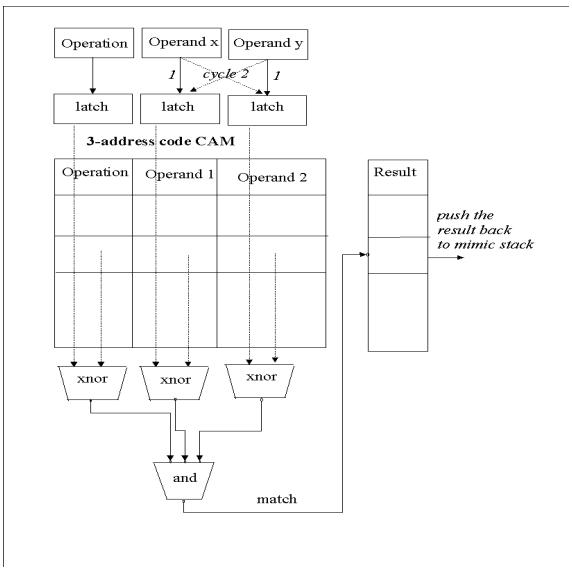

Fig. 8. Three-address code CAM.

The *iinc* bytecode is a special instruction; it is the only instruction that directly operates on the memory, i.e. local variables, without the need to push the local variable on to the stack. The handling of such a non-stack operation among of all stack operations is slightly tricky when we use the mimic stack. Because the mimic stack uses the register reference and not the contents, if a bytecode sequence that pushes a local variable content into the stack is followed by *iinc* instruction which directly operates on the local variable, there will be a data inconsistency between the stack and the local variable in memory; this is perfectly legal, as the bytecode operation following the *iinc* instruction consumes the earlier value from the stack. But when using the register reference for mimic stack, a search should be performed on the mimic stack to see if the local variable is in use in the mimic stack; if so a copy of the local variable should be made and the value then incremented.

3.3.4 Optimization stages

The optimisation stage is a series of specialized optimisation sub-stages in sequence. The optimisations can be classified as machine-dependent or machine-independent, as shown in the Fig. 9. It should be noted that CSE optimisation is performed during the translation stage, when the intermediate three-address codes are generated. The CSE optimisation introduces many redundant move operations which need to be eliminated before any other optimisation, in order to conserve space in the three-address CAM buffer, as well as other intermediate holding buffers in the following stages. The copy propagation optimisation eliminates redundant copy operations and is therefore placed as the first optimisation stage.

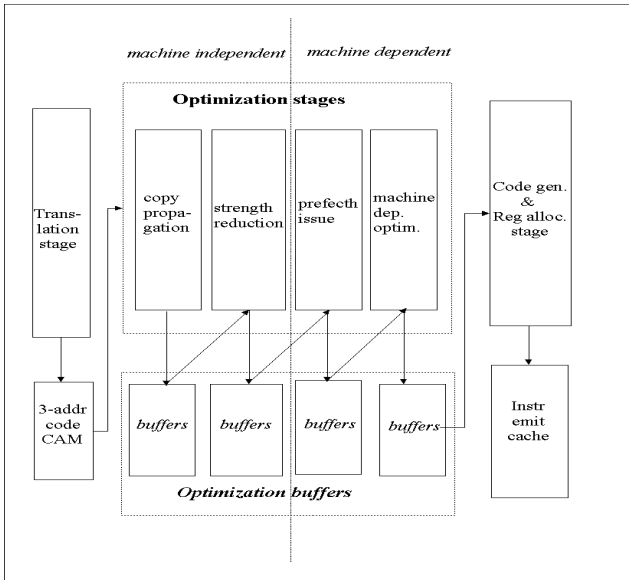

Fig. 9. Optimisation stages.

The other optimisations include strength reduction, which replaces an operation with its equivalent simpler operation e.g. a multiplication by 2 can be replaced by either addition operation to itself or by the left shift operation in case of integer operands. There are several possible machine-dependent optimisations. Software prefetch instructions can be embedded in the native code at appropriate places can reduce the load stall created in the backend pipeline [19]. The other optimisation includes delay slot scheduling, with the delay slot instruction filled from the branch target code. The optimised codes are placed in the buffers that are finally accessed by the code generation stage to produce the native code.

3.3.5 Code generation and Register allocation stage

In the code generation stage, the optimised three-address codes are converted into native codes by allocating architectural registers. The architectural registers replace the virtual registers allocated in translation stage. The temporary stack registers are given priority over local

variable registers for architectural register assignment. The difference between the translation stage register allocation and the architectural register allocation is that spill/fill codes are generated if there are no architectural registers left for allocation This is one reason for combining the register allocation with code generation. The calling convention of the native processor is taken care while generating code for method calls. The register allocation is done in two phases:- during the translation stage and during the code generation stage. The architectural register allocation, the virtual-to-architectural register mapping, and the code generation logic are shown in the figure 10.

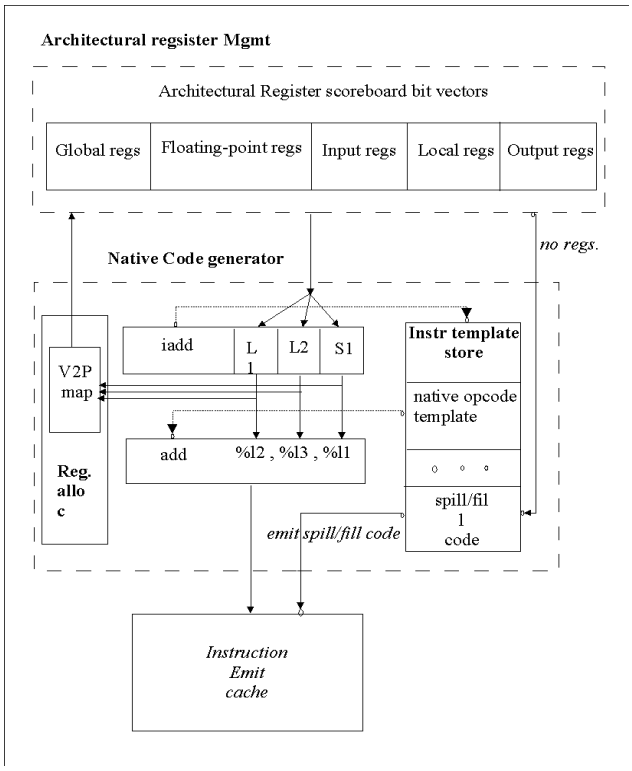

Fig. 10. Code generation and Register allocation stage.

In the translation stage register allocation, the depth of the mimic stack is monitored at the end of each basic block. If the mimic stack in not empty, its contents are explicitly spilled into designated memory locations, by generating equivalent three-address codes. The situation where a variable is pushed and a branch is taken leaves the mimic stack non-empty at the basic block end. For example, x = (x>y)?x:y; generates such non-empty stacks at the end of the basic block. In the code generation stage, the spill/fill of registers occurs when there is not enough architectural registers available for allocation. A virtual-to-architectural register mapping table keeps track of the allocation mapping is referenced before allocating a new architectural register.

Using the three-address opcode to index into the instruction template store, which holds the native instruction template (i.e., opcode part with the unfilled register slots), does the code generation. The unfilled register slots are filled into the template after register allocation to form the native opcode. The generated native opcodes are stored in the Instruction Emit Cache (IEC) for execution by the backend pipeline. The native PC is initialised in a way that it causes the IEC to be searched in order to execute a method. In case of an instruction miss, the backend pipeline stalls, while the missed PC is mapped to BPC and the corresponding the basic block is compiled on-the-fly by the pipeline to fill the IEC.

The information required for a Java method to execute is available in the Java frame data that is on a Java stack. The Java frame data is mapped on a C stack frame, as shown in Fig. 11, and the generated native code accesses all the information related to that method through the Java frame information available on the C stack. The native-code generation for accessing the Java frame information becomes simple by offsetting into the native stack pointer (SP). The constant pool pointer and the method information all are accessed through emitting native load/store instructions with appropriate offsets into the SP or FP pointers. The previous frame is accessible by using the frame pointer (FP).

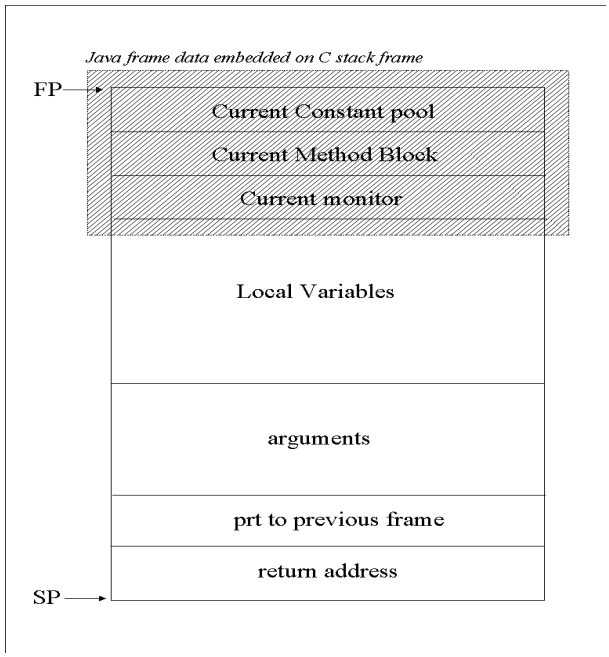

Fig. 11. Java frame data mapped onto native C stack frame

4 Simulation environment

The simulation environment consists of front-end hardware compilation pipeline and the backend native processor pipeline. For the backend pipeline we plan to use the Simplescalar processor simulator and for the front-end we are modifying the Latte JVM JIT compiler to incorporate the hardware compilation pipeline stages and to generate code for, the Simplescalar processor. SpecJVM98 will be used as benchmarks to evaluate the hardware compilation pipeline. Various data are being collected to determine the size of hardware resources required at each pipeline stage.

5 Conclusion and Future work

In this paper, we have described the design and the micro-architectural details of a hardware compilation pipeline for a high-performance front-end Java processor. We have studied the possibility that the hardware compilation will consume few cycles compared to few hundred cycles consumed by JIT compilers for compiling the same amount of Java byte codes to native codes. Various local optimisation techniques that can be implemented in hardware with minimum resources have been discussed and their micro-architectures. The translation of bytecodes to three-address code using the mimic stack has been described, and a two-stage hardware register-allocation algorithm is proposed. During the translation stage virtual registers are allocated and in the code generation stage virtual registers are mapped to architectural registers by hardware. The architecture of the pipeline and the various stages show that a complete hardware compilation pipeline is feasible for Java bytecode compilation and seems to require a reasonable amount of hardware resources for high-speed compilation.

6 Literature

[1]. Sun Microsystems, "picoJava II Microarchitecture Guide:, March 1999.

[2]. Mladen Berekovic, et al, "Hardware Realization of a Java Virtual Machine for High Performance Multimedia Application", Journal of VLSI Signal Processing, Kluwer Academic Publisher, 1999.

[3]. Vijayakrishnan Narayanan, "Issues in the Design of a Java Processor Architecture", Ph.D. thesis, Dec 1998.

[4]. Watheq El-Kharashi, et al, "An Operand Extraction-based Stack folding Algorithm for Java processors", Hardware Support for objects and Microarchitecture for Java - 2nd annual workshop, Texas, Sep 2000.

[5]. Kemal Ebcioglu, et al., "A JAVA ILP Machine Based on Fast Compilation", International Workshop on Security and Efficiency Aspects of Java, part of the IEEE MASCOTS Conference. Eilat, Israel, January 9-10, 1997.

[6]. Andreas Krall and et al, "JavaVM Implementation: Compiler versus Hardware", Computer Architecture (ACAC '98), Australian Computer Science Communications 20(4). Perth, Australia.

[7]. Vijaykrishnan, Ranganathan, et al., "Object-Oriented Architectural Support for a Java Processor", ECOOP'98, LNCS 1445, pp.330-355, 1998, Springer-Verlag Berlin Heidelberg 1998.

[8] Chang, Ton, Kao, Chung, "Stack operation folding in Java processors", IEEE Proc. of Computing Digital Tech. Vol. 145, No. 5, September 1998.

[9] Byung-Sun, et al, "LaTTe: A Java VM Just-in-Time Compiler with Fast and efficient register allocation". Intl conference on Parallel architectures and compilation techniques, Oct. 1999.

[10]. T. Suganuma, et. al, "Overview of the IBM Java Just-in-Time compiler". IBM systems Journal, vol. 39, no 1. 2000.

[11]. Ali-Reza Adl-Tabatabai et. al., "Fast, Efficient code generation in a Just-In-Time Java compiler". Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation, pages 280-290. ACM Press, 1998.

[12]. Alfred V. Aho, Ravi Sethi, Ullman, " Compilers, principles, techniques and tools", Addison Wesley publishing, 1986.

[13]. Bill Venners, "Inside Java Virtual Machine", McGraw hill. 1998.

[14]. SPEC JVM98 benchmarks, "http://www.spec.org/osg/jvm98, 1998.

[15]. T. Wilkinson, " Kaffe: A JIT and interpreting virtual machine to run Java code", http://www.transvirtual.com/ 1998.

[16].J2C,
"http://www.webcity.co.jp/info/andoh/java/j2c.html

[17].TurboJcompiler,"
http://www.osf.org/www/java/turbo".

[18].
http://java.sun.com/products/hotspot/docs/whitepaper/Java_HotSpot_WP_Final_4_30_01.html

[19]. Callahan, K. Kennedy, Porterfield, "Software prefetching," proceeding of the fourth international conference on architectural support for programming languages and operating systems, April 1991.