SEJA: SimpleScalar Extensions for Java Applications

Seok Joong Hwang and Seon Wook Kim
School of Electrical Engineering, Korea University
1,5-ka, Anam-dong, Sungbuk-ku, Seoul 136-701, Korea
E-mail : {nzthing, seon}@korea.ac.kr

Abstract: In spite of remarkable acceptance and popular use in many areas due to its portability across platforms, inherently Java programming language has one serious drawback in terms of performance. The performance problem results from high interpretation overhead. In order to overcome this performance limitation, many alternative software and hardware solutions have been proposed. One of the solutions is to extend a processor architecture for accelerating Java applications. In this paper, we present the SimpleScalar-based extension with Kaffe. Even though several versions of processor extensions are available, none of their simulators are available for research community. Our work is able to help researchers study the Java processors and Java virtual machines in detail.

1. Introduction

One of primary reasons why Java programming language have been popularly used in many areas is due to its platform independence. It means that we can run Java executable files on any platform. It is done by once compiling Java applications into Java's platform independent executable files which contains sequence of bytecodes. The bytecode is the name of Java own instruction set architecture. Basically this portability is supported by Java Virtual Machine (JVM) which provides an abstraction layer for Java applications by interpreting their bytecodes in a software manner. Obviously this software interpretation of the bytecodes results in serious performance loss when executing Java applications. This software processing spends lots of machine cycles than executing actual operations of given bytecodes.

In order to overcome this performance limitation, many alternative software and hardware solutions have been proposed. Among software approaches, the most popularly used technique is Just-In-Time (JIT) compilation [1], [2]. The JIT compilation is to compile Java bytecode into machine dependent native instructions just prior to actual execution. While this method improves performance significantly, it causes large memory footprints for containing the compiled instructions. That memory consumption may be inappropriate for an embedded system with limited memory devices. In hardware approaches, one of the promising solutions is to extend processor architecture for accelerating Java applications like Jazelle [3]. This approach takes performance gains through directly executing simple and moderate bytecodes without the software interpretation. Because JVM handles the rest of complex bytecodes, it is sufficient to extend only minor portion of the existing processor architecture.

This work is supported by Nano IP/SoC Promotion Group of Seoul R&BD Program in 2008.

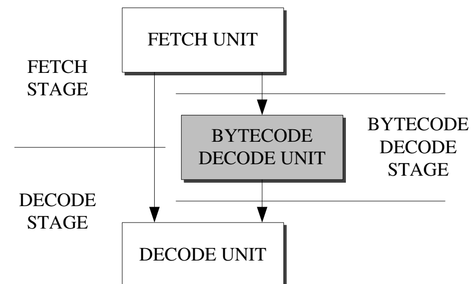

Figure 1. SimpleScalar pipeline extension for the hardware bytecode decoding.

This paper focuses on the processor extension for getting performance improvements on embedded systems. Even though several versions of processor extensions are available, none of their simulators are available for research community. Here we present the SimpleScalar-based extension with Kaffe. The SimpleScalar is the most popular processor simulation tool for program performance analysis, detailed micro-architectural modeling, and hardware-software co-verification [4]. And Kaffe is one of free JVMs which provide full features of JVM [5]. The extended part of SimpleScalar for Java was written in System-C, and it exchanges data in each cycle with the existing SimpleScalar infrastructure. Our work can provide many benefits for researchers and developers on the processor extension: 1) In virtue of the SimpleScalar and Kaffe, it is able to help researchers study the Java processors and Java virtual machines in detail. 2) Being written in System-C, it can be simulated in functional level and also easily used as an IP for implementing other processor extensions.

The rest of this paper is organized as follows. Section 2 presents an overall architecture. Section 3 presents our processor extension in detail. In Section 4, we evaluate performance. Finally, we make the conclusion in Section 5.

2. Overall Architecture

Based on the SimpleScalar, we developed SEJA which represents for SimpleScalar Extensions for Java Applications. SEJA introduces an additional pipeline stage, called a bytecode decode stage, to translate bytecodes into SimpleScalar's native instructions. As shown in Figure 1, the bytecode decode stage is inserted between fetch and decode stages. With containing a bytecode decoder, this stage is dynamically activated only when the processor executes Java bytecodes. The SimpleScalar is written in C language and it simulates processor's behaviors cycle by cycle. Therefore, it can be naturally integrated with clock based SystemC models [6]. We implemented the hardware decode unit in SystemC and aug-

```
class SIMPLESCALAR:sc_module{
        sc_in<PC_TYPE> PC_Inc;
        sc_in<bool> AvoidBypass;
        sc_in<INST_TYPE> InstIn;
        sc_in<bool> clk;

        sc_out<bool> Awake;
        sc_out<INST_TYPE> InstOut;

        static void simplescalar_init();
        void simplescalar_loop();
        bool isTerminated;

        SC_CTOR(SIMPLESCALAR){
                isTerminated = false;
                SC_METHOD(simplescalar_loop);
                sensitive_pos << clk;
        }

}


int sc_main(int ac, char *av[]){
        sc_signal<…> …
        sc_signal<bool>   clk;

        SIMPLESCALAR::simplescalar_init();
        SIMPLESCALAR ss("Simple Scalar");
        ss(……);

        sc_initialize();
        while(!ss.isTerminated){
                clk.write(1);
                sc_cycle( …);
                clk.write(0);
                sc_cycle( …);
        }

        return 0;
}
```

Figure 2. Integration of the hardware decode unit and the SimpleScalar in SystemC.

mented the SimpleScalar code for modeling fetch and decode stages to exchange information in every clock cycles. Figure 2 shows how the hardware decode unit is integrated with the SimpleScalar in SystemC. The main simulation loop of the SimpleScalar is wrapped by a SystemC module. By using this wrapper module, the SimpleScalar simulation loop can communicate and synchronize with the hardware decode unit based in every clock cycle.

We also modified Java Virtual Machine (JVM) to take advantage of the extended processor's decoding capability. The JVM is based on the Kaffe which is one of free JVMs. The original Kaffe interprets Java methods in a core function, runVirtualMcahine(). But SEJA replaced the function with runHWdecoder() as depicted in Figure 3. It activates the hardware decode unit by jumping to a start address of bytecodes like a function call. The hardware decode unit decodes simple bytecodes into native instructions directly, or calls a microcode for complex ones. The microcode has a special interface with the hardware decode unit in order to reduce call overheads. It can invoke another Java method or call JVM's software function for complex jobs such as memory allocation, class/constant/field resolutions, exception handling, and so on.

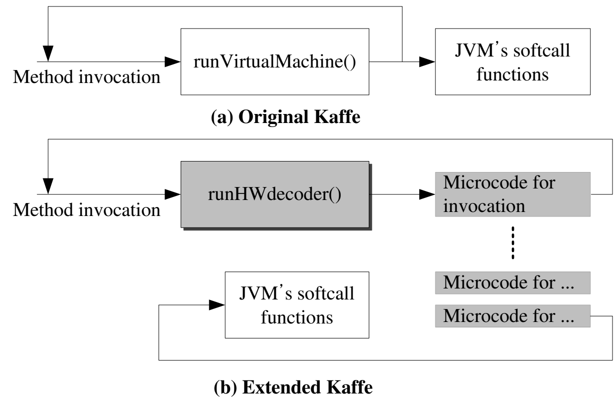

Figure 3. Kaffe extension to support the hardware decoder.

3. Hardware Decode Unit

In this section, we present the hardware decode unit in detail. The hardware decode unit consists of several subcomponents; bytecode fetch/decode management unit (BFDMU), stack management unit (SMU), dynamic code ROM, operand translation unit (OTU). Figure 4 depicts how the subcomponents are organized and they interact with the fetch and decode units of the extended SimpleScalar.

Translation of bytecodes is not a straightforward work, i.e. it is not an one-to-one mapping process. The number of translated native instructions is different for each bytecode. And it is possible to have a conditional branch between translated native instructions of even a single bytecode. Therefore we should control a sequence of translated native instructions according to the current machine state. BFDMU is in charge of this job. And it also manages a sequence of bytecodes to be decoded since the existing fetch stage has no idea on Java program counter. Dynamic code ROM provides control information to BFDMU as well as it contains the corresponding native instructions except for operand fields. OTU completes the instructions by assembling operands. We support hardware stack caching by using a set of processor registers in order to reduce overheads from frequent memory accesses - the Java bytecode always uses a stack space due to the absence of registers. SMU manages this stack caching in a transparent manner; it supports automatic spilling and reloading of the cache registers.

3.1 Bytecode Fetch/Decode Management Unit

This BFDMU controls the rest of the hardware decode unit and communicates with the exiting fetch stage of the SimpleScalar. Because a bytecode consists of variable number of bytes, we maintain a fetch queue in order to continuously supply complete bytecodes into the dynamic code ROM. As shown in Figure 5, the fetch queue provides one byte opcode and up to two bytes immediate field in a cycle according to Java program counter. After the dynamic code ROM identifies the actual length of an input bytecode, BFDMU pro-

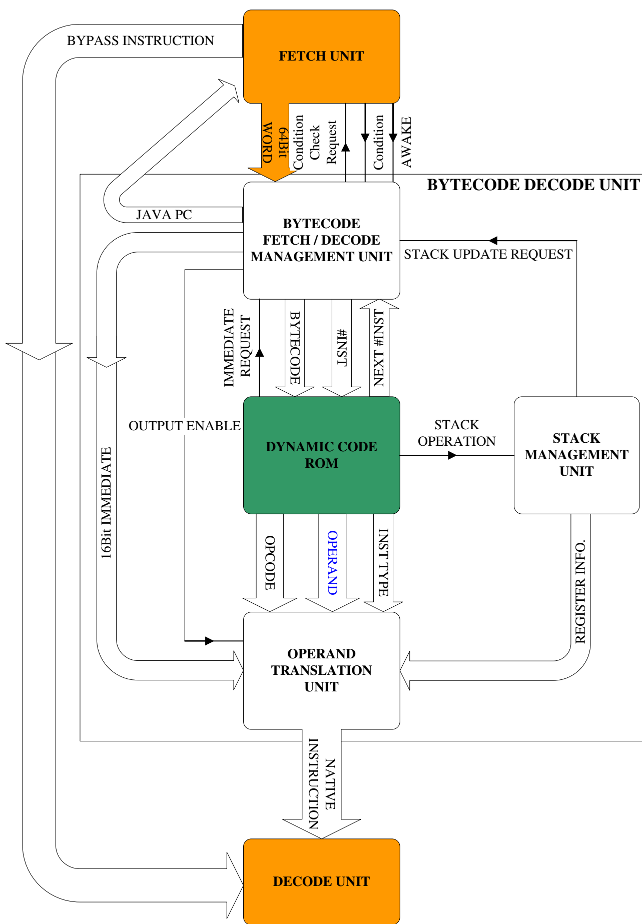

Figure 4. Subcomponents of the bytecode decode unit and their organization.

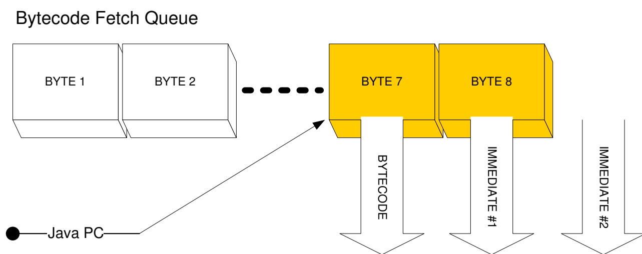

Figure 5. Bytecode selection from fetch queue.

vides more bytes if the previous bytecode was supplied incompletely.

There are several sources to make it complex to control a sequence of translated native instructions. SMU supports automatic spilling and reloading of stack cache registers if all cache registers are exhausted or a cache miss occurs. It implies that a mixture of translated native instructions can be different even for the same bytecode. And we should handle unsupported bytecodes and internal branches in a bytecode. Therefore, BFDMU needs to have proper states to accommodate these situations. As shown in Figure 6, we defined five state for the decoder: `Suspend`, `Terminate`, `Normal`, `Condition Check`, and `Stack Update`. `Suspend` state is a software interpretation mode because the hardware

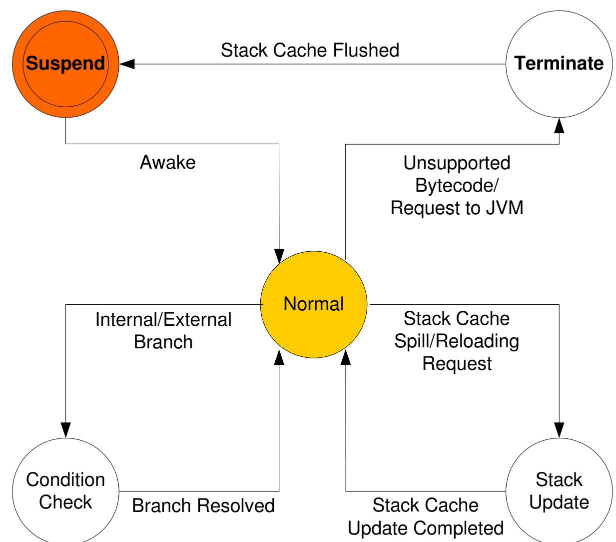

Figure 6. The state diagram of the BFDMU.

decoder is disabled in this time. `Terminate` state is preparing a phase period before entering into `Suspend` state; It flushes all cache entries into a real stack memory. Whenever the decoder meets an unsupported bytecode or demands JVM's help, BFDMU enters into this state. `Normal` state is a normal hardware bytecode decoding state. `Stack Update` state is to handle situations that all stack cache registers are exhausted or a cache miss occurs. `Condition Check` state is to check whether an internal branch should be taken or not.

3.2 Stack Management Unit

Because Java bytecode execution is primarily based on a stack machine, it is important to have an optimized way to access a stack memory. In order to reduce frequent memory accesses to/from the stack, we provide a popular stack caching method that uses a set of processor registers as a circular buffer [7]. SMU is in charge of managing this circular buffer. As shown in Figure 7, the circular buffer is maintained with two masks to indicate stack top and bottom registers. For a stack push operation, SMU takes a left circular shift on the stack top mask and then assigns a new stack slot to a register that is indicated by the mask. But if all registers are exhausted for the caching, SMU requests BFDMU to spill a stack bottom register into the real stack memory. After the spilling completes, a left circular shift is taken on the stack bottom mask. For a stack pop operation, a right circular shift is taken on the stack top mask. When all cache registers are freed for pop operations, SMU requests BFDMU to reload a register content from the real stack memory while taking a right circular shift on the stack bottom mask.

3.3 Dynamic Code ROM

A bytecode in a viewpoint of the hardware decoder has several properties: variable length, variable number of translated native instructions, dynamically determined sequence of translated native instruction due to internal branches, and also

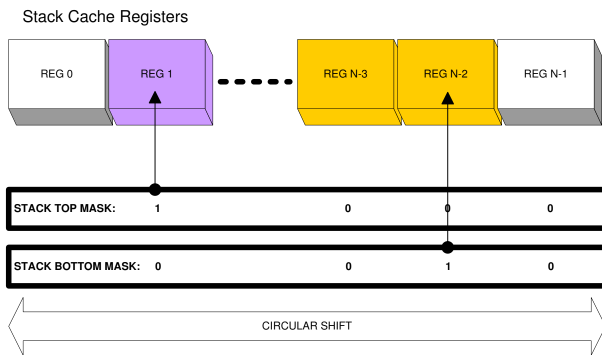

Figure 7. Strategy of register assignment.

dynamically determined operand registers due to the stack caching.

To cover these properties, dynamic code ROM provides the following additional information rather than just containing pure native instructions.

- Immediate field length: Length of immediate fields.
- Branch type: Internal/External/No branch. Internal branch is for a conditional jump between translated instructions corresponding to a single bytecodes, and External branch is a type of inter-bytecode branches.
- Next instruction sequence: A sequence number of an instruction which will be picked up at the next cycle if there is no taken branch.
- Conditional instruction sequence: A sequence number of instruction which will be pick up if a branch is taken. Dynamic code ROM is address by a bytecode and these sequence numbers.
- Instruction format: Register/Immediate/Jump format. OTU completes operands according to this format and the following information on stack operands.
- Request to push a stack entry: whether it requires a stack push operation or not
- Request to pop a stack entry: whether it requires a stack pop operation or not
- Request of escaping: Request to escape from the hardware decode mode if it meets an unsupported bytecode or demands JVM's help.

Since stack entries are cached in the circular register buffer, dynamic code ROM does not contain complete forms of native instructions that have fixed operand register numbers. Instead, it contains pseudo register numbered instructions. The pseudo number indicates its location in the stack space and will be translated into a real register number by OTU.

3.4 Operand Translation Unit

As we have mentioned just above, this unit translates a pseudo register numbered instruction into a complete form of a native instruction by getting register assignment information from STU. And it assemblies immediate fields which were obtained by BFDMU into the native instruction.

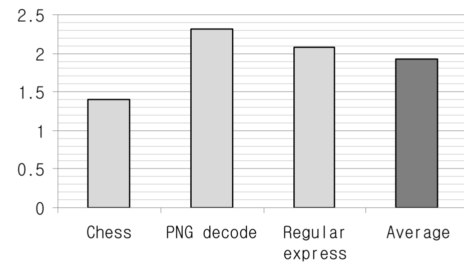

Figure 8. Speedup with respect to the original Kaffe.

4. Performance

In this section, we evaluate the performance of SEJA and compare it against that of the unmodified Java runtime environment in running three EEMBC's GrinderBench benchmarks [8]. Because SEJA is based on the SimpleScalar, we can easily analyze the performance by varying processor configurations. However, here we evaluate the performance by modeling the processor with an in-order single issue pipeline and 16KB L1 instruction and data caches and a unified 64KB L2 cache. In this configuration, SEJA achieved speedup by 193% on average and up to 231% as shown in Figure 8.

5. Conclusion

In this paper, we presented SEJA, SimpleScalar Extensions for Java Applications to accelerate the Java execution time. SEJA use the processor extension method to avoid the software interpretation overhead by executing bytecodes directly in the hardware. Even though several versions of processor extensions are available, none of their simulators are available for research community. Since SEJA is based on the SimpleScalar which is the most popular processor simulation tool with Kaffe JVM, our work is able to help researchers study behavior of the Java virtual machines and Java processors in detail. From the performance result, we showed that SEJA could accelerate the Java execution time about 1.93 times on average and up to 2.31 times than the software interpretation method.

References

[1] OpenJIT. http://www.openjit.org/.
[2] shuJIT. http://www.shudo.net/jit/#docs.
[3] ARM. Jazelle. http://www.arm.com/products/esd/jazelle_home.html.
[4] Doug Burger and Todd M. Austin. The simplescalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, 1997.
[5] Sadaf Mumtaz and Naveed Ahmad. Architecture of Kaffe. http://www.kaffe.org.
[6] Open SystemC Initiative. SystemC. http://www.systemc.org.
[7] Martin Schoeberl. Design and implementation of an efficient stack machine. In *Proceedings of the 12th IEEE Reconfigurable Architecture Workshop (RAW2005)*, Denver, Colorado, USA, April 2005. IEEE.
[8] GrinderBench. http://www.grinderbench.com.