FPGA Implementation of a Statically Reconfigurable Java Environment for Embedded Systems

Shinsuke Nino Takayuki Mori YoungHun Ko Yuichiro Shibata Kiyoshi Oguri

Department of Computer and Information Sciences, Nagasaki University
1–14 Bunkyo-machi Nagasaki 852–8521, Japan
E-mail: {nino,mori,younghun,shibata,oguri}@pca.cis.nagasaki-u.ac.jp

Abstract

A demand for low power and high performance Java environments is now growing in the embedded systems field. One approach is dedicated Java processors which directly execute Java bytecode. We have proposed a novel reconfigurable Java environment which consists of a general purpose core processor with configurable bytecode processing units, a bytecode compiler, and a software Java Virtual Machine (JVM). This paper discusses design of a memory system for the reconfigurable Java architecture focusing on a hardware custom stack. Empirical evaluation using prototype systems reveals that adding a 2-word hardware stack shows the best results, achieving the performance enhancement of up to 15.9% compared to software execution.

1 Introduction

Java is widely used for embedded systems such as mobile phones and PDAs. Although one of the benefits of using Java is its portability, this platform-independent model is also disadvantageous in terms of execution speed. While the increase in the microprocessor clock frequency has been alleviating this disadvantage for desktop computers, a demand for high performance and low power Java environments is still growing in the field of embedded systems. Reconfigurable computing is one of the most promising approach to this problem[1, 2].

Our proposed Java environment[3] system consists of an FPGA-based statically reconfigurable core processor with bytecode processing hardware for custom instructions, a bytecode compiler, and a software JVM[4]. A given class file is analyzed and modified to utilize custom instructions by the bytecode compiler. At the same time, hardware modules for the required custom instructions are generated and the core processor is tailored to the given class file. The modified class file is interpreted by the software JVM which runs on the customized processor configured on the FPGA. Static reconfigurability of FPGAs and custom instructions that are directly executed by the dedicated hardware enable a high-speed Java processing environment at a reasonable hardware cost, compared to a normal software JVM environment.

Obviously, design of memory structure is a key to this approach in terms of architecture. Since Java bytecodes inherently have frequent access to stack structure, a provision of a hard-wired custom stack in the FPGA would be effective. However, full hardware implementation of the Java stack is costly and causes a large overhead of data consistency process with the memory. In this paper, performance impacts and hardware costs of a custom stack are empirically evaluated with several prototype systems, so that efficient stack structure for a reconfigurable Java environment is revealed.

The rest of the paper is organized as follows. In Section 2, an overview of the proposed Java environment is described. In Section 3 and Section 4, the prototype system is presented in detail in terms of hardware and software. After evaluation results are shown and discussed in Section 5, the paper is concluded in Section 6.

2 Overview of the environment

The heart of hardware of the proposed Java environment is a customizable MIPS-based RISC processor configured on an FPGA, to which desired custom instructions can easily be added. Hardware direct execution of frequently used bytecodes is carried out by the added custom instruction hardware modules. In addition, convectional optimization techniques such as bytecode folding [5] can also be used to make a new single custom instruction which executes a series of processes for multiple bytecodes to boost the performance. Since JVMs are stack machines, the frequency of stack access is inherently high. A provision of a hard-wired custom stack in the FPGA is an effective approach to mitigate the stack access overhead.

Figure 1 shows an overview of the proposed Java environment. An execution flow on this system is as follows. At first, a class file to be executed is analyzed by the bytecode compiler. Then, the byte compiler replaces some of the bytecodes in the class file with custom instruction codes. The compiler also generates Verilog-HDL files for the instruction modules corresponding to

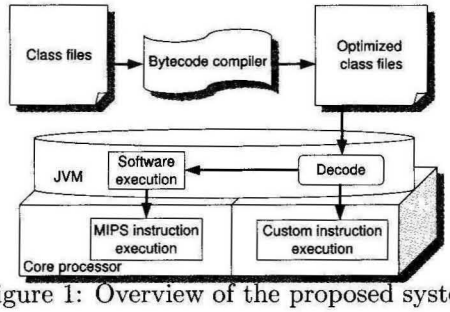

Figure 1: Overview of the proposed system

the inserted custom instructions. The bytecode decoding functions of the JVM are also modified so as to support the custom instructions. The instruction codes for the custom instructions are assigned from the reserved instruction space of the MIPS architecture and the JVM issues these instructions via inline assembly. The other original bytecodes are processed with a sequence of normal MIPS instructions.

3 Hardware design

We have implemented an original core processor for executing a JVM on an FPGA. The interface of custom instruction modules is designed not to disturb a normal instruction flow in the main pipeline. In addition, a custom stack and a stack controller are provided to store the data that are frequently accessed by Java programs.

3.1 Core processor and caches

The architecture of the core processor is based on the 32-bit five-stage pipelined MIPS R3000, which has 32 general purpose registers as well as some special registers. Figure 2 shows a conceptual diagram of the core processor. A special decoder for custom instructions is reconfigurable and attached in the instruction decode (ID) stage of the main pipeline.

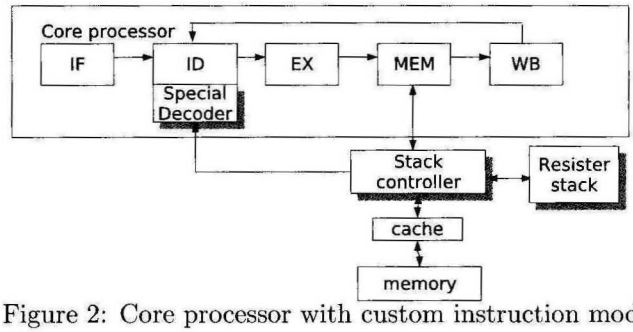

Figure 2: Core processor with custom instruction modules

In the FPGA, a couple of single level direct mapped write-back caches for instruction and data, each of which has a capacity of 4KB, are provided.

3.2 Custom stack

A custom stack is a technique to speedup stack access by caching some stack entries in hardware registers. Also, the custom stack provides a wide data bandwidth for custom instructions by allowing to access multiple stack entries at the same time. Figure 3 shows the structure of custom stack.

Access to the custom stack is managed by a stack controller. To keep consistency to the original stack on the main memory, the stack controller initiates memory access when the value of the stack pointer is updated. This transaction can be overlapped with process of normal instructions of the core processor, but following stack access is stalled until the transaction is completed.

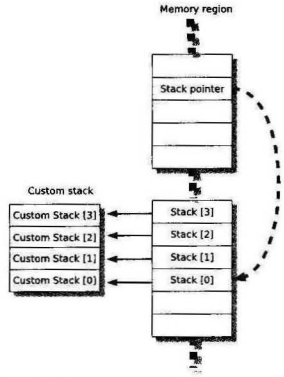

Figure 3: Structure of the custom stack

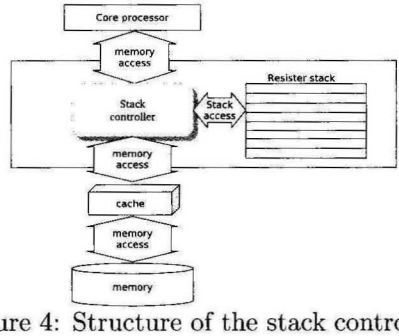

Figure 4: Structure of the stack controller

Figure 4 shows the structure of the stack controller. The stack controller supports the following three operations.

1. Data access: If data requested by the processor is in the custom stack, the stack controller gives the corresponding data to the processor. Otherwise the controller accesses the cache.
2. Stack set: This operation fills the custom stack with the upper data of the original stack in the memory.
3. Stack data manipulation: Some Java bytecodes require to manipulate stack data. An example is a swap operation of the top two entries of the stack.

3.3 Custom instructions

When a software JVM processes a single bytecode, a bunch of native MIPS instructions are executed. For example, even a simple bytecode such as addition on stack data requires the following processes.

1. Stack data on the main memory are popped out and stored in the registers for arithmetic.

2. The arithmetic operation is performed on the registered data.
3. The result value is pushed back into the stack structure on the main memory.

To reduce the time required for the stack data operations, custom instructions are introduced in the processor. This combines the above mentioned processes into a single custom instruction. Figure 5 shows the effect of using a custom instruction in stead of a sequence of native MIPS instructions. Currently, our processor supports custom instructions for arithmetic on stack data, that is, IADD, ISUB, IOR, IXOR, ISHR, ISHL, and IUSHR.

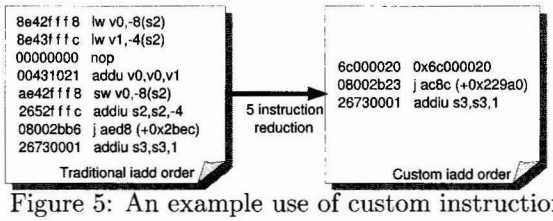

Figure 5: An example use of custom instructions

4 Software environments

4.1 Waba

Waba [6] is a lightweight Java virtual machine mainly developed for portable devices. Since the grammar of Waba is completely a subset of that of Java, Waba programs can be developed using standard programming tools for Java. In order to optimize the execution performance for portable devices, functionalities that are not frequently used for these devices are omitted in Waba.

Operation cords supported by Waba are a set of Java standard operation cords except for those related to exceptions and threads. The Waba virtual machine does not support 64-bit data types (long and double), the quick operation and the reserved operation. The Waba virtual machine were easily ported to our processor together with the Red Hat newlib standard library.

Our modification to the original Waba virtual machine allows to execute custom instructions which are directly processed by the processor. Using GCC inline assembly as shown in Figure 6, machine codes for custom instructions are inserted.

```
case OP_iadd:
        asm(".word 0x6c000020");
        pc++;
        break;
```

Figure 6: Use of a custom instruction in source code

4.2 A bytecode compiler

In order to enable bytecode execution with custom instructions, given class files must be modified. Therefore, we have developed a bytecode compiler using the Byte Code Engineering Library (BCEL) [7] by the Jakarta project which provides an easy-to-use API to bytecodes in class files.

The bytecode compiler analyzes a given class file and replaces targets codes with new custom bytecodes using the undefined bytecode space of Java. Use of some conventional optimization techniques such as instruction folding [5] are partially supported at present.

The compiler also generates Verilog-HDL files for additional functional units which are required to execute the custom instructions that are used in the given class files. To keep the hardware cost of the processor core low, only enough circuits to execute the given class files are generated. In other words, structure of the processor core can be statically tailored to each class file.

5 Evaluation

5.1 Hardware costs

We have implemented four prototype architectures parameterizing the number of entries of the custom stack. The *base* architecture only supports normal software execution. On the other hand, the *hard2*, *hard4*, and *hard8* architectures provide 2, 4, and 8-word custom stacks, respectively, enabling bytecode processing by custom instructions. Here, the word size is 4 bytes for all the architectures. Eight simple arithmetic instructions are implemented as custom instructions as described in Section 3.3.

Table 1 shows the implementation results of these architectures. The target device is Xilinx Virtex-II Pro XC2VP7, and ise-8.2i is used for synthesis and place-and-route processing.

By adding custom instruction units and the custom stack hardware, 18 to 28% of FPGA slices are increased. In addition, the operational frequency is degraded by 14 to 15%. The stack controller between the processor and the memory gave a critical path. Therefore, the frequency was independent on the number of entries of the custom stack.

Table 1: Implementation results

name	slices	FFs	4input LUTs	frequency
base	3086	1920	5829	52.95MHz
hard2	3646	2034	6910	45.07MHz
hard4	3724	2105	7067	45.07MHz
hard8	3963	2250	7525	45.47MHz

5.2 Breakdown of bytecodes executed

Here, effects of the proposed architecture are evaluated using four benchmark program; FFT, DCT, MD5 and SHA-1. Figure 7 shows frequency of bytecodes executed for each benchmark program. The bytecode group that treats the stack, that is, the load, arithmetic, store, and constant push account for a large percentage of the total bytecodes executed. Since load instructions also require local variables as well as the stack data, they are not targets of custom instructions at present. Although Figure7 suggests a merit of hardware execution of these instructions, this would cause a large increase in hardware and would require another detailed trade-off analysis.

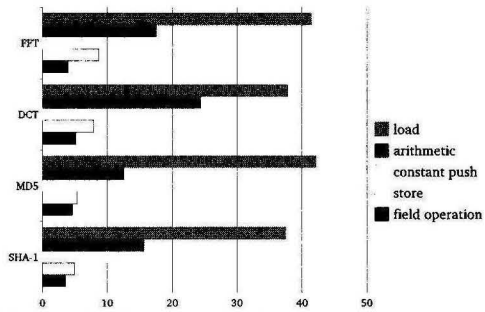

Figure 7: Frequency of bytecodes executed

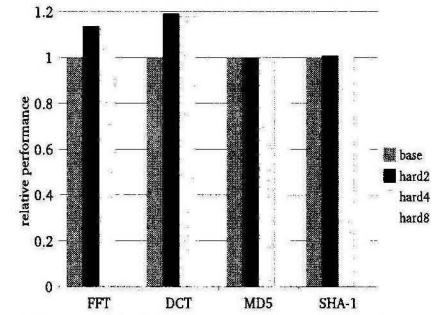

Figure 9: Performance comparison

Figure 8 illustrates the frequency of the aforementioned eight custom bytecodes executed for each benchmark program. On average, 14.4% of the bytecodes were executed using custom instructions on the core processor.

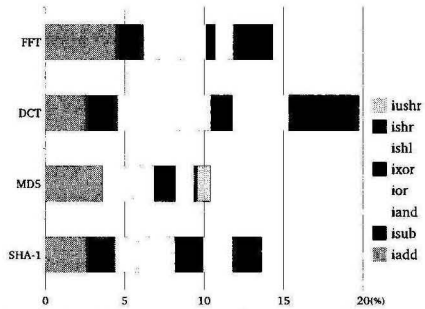

Figure 8: Frequency of custom bytecodes

5.3 Performance evaluation

Execution performance of the benchmark programs on the prototype system is measured and summarized in Figure 9, where the data are normalized to the base architecture. Access latencies for the custom stack and the cache are 1 and 2 clock cycles, respectively, while access to the main memory requires 15 clock cycles.

The best performance gain of 15.9% was achieved by DCT for the hard2 architecture. Since the DCT program is dominated by ALU arithmetic operations which can be processed with custom instructions and the custom stack, performance improvement for this application is relatively high. FFT also improves the performance by 10%. However, the effect of the custom stack for MD5 and SHA-1 is not very clear.

MD5 and SHA1 have a small number of bytecode instructions, and this would be a factor in exposing an overhead of the bytecode processing. However, increasing in supported custom instructions including instruction folding can cancel this overhead to some extent, by making the best use of a wide bandwidth to the custom stack.

In terms of the number of entries of the custom stack, the 2-word stack (hard2) showed the best performance in all the evaluated architectures. This is attributed to the following two reasons. First, access of the virtual machine is generally concentrated to the upper part of the stack. Therefore, the number of entries required to speedup the stack access is essentially small. Second, our stack controller accesses the data cache whenever the value of the stack pointer is changed in order to keep the consistency. Since this overhead grows proportionally to the stack size, the 8-word stack (hard8) degrades the performance compared to the small size custom stacks.

6 Conclusion

In this paper, we presented a cost-effective Java framework using static reconfigurability of FPGAs focusing design of a hardware stack. Empirical evaluation based on the prototype systems revealed that adding a 2-word hardware stack showed the best results, achieving the performance enhancement of 15.9% in spite of some optimization techniques such as instruction folding had not been implemented yet. Our future work includes supporting more custom instructions, supporting instruction folding, introducing of additional registers for storing local variables, and evaluation using larger benchmark applications.

References

[1] M. Schoeberl, "Evaluation of a Java processor," *Tagungsband Austrochip 2005*, pp. 127–134, Oct. 2005.

[2] J.Kreuzinger, U.Brinkschulte, M.Pfeffer, S.Uhrig, and T.Ungerer, "Real-time event-handling and scheduling on a multithreaded Java microcontroller," *Microprocessors and Microsystems*, vol. 27, no. 1, pp. 19–31, 2003.

[3] S. Nino, J. Sakamoto, T. Mori, Y. Shibata, and K. Oguri, "A reconfigurable Java environment for embedded systems," *Proc. COOL Chips*, p. 189, Apr. 2006.

[4] T.Lindholm and F.Uellim, *The Java Virtual Machine Specification*, 2nd ed. Prentice Hall PTR, 1999.

[5] L.-R. Ton, L.-C. Chang, M.-F. Kao, H.-M. Tseng, S.-S. Shang, R.-L. Ma, D.-C. Wang, and C.-P. Chung, "Instruction folding in Java processor," *Proc. ICPADS*, pp. 138–143, 1997.

[6] "Wabasoft," http://www.wabasoft.com/.

[7] "Jakarta BCEL," http://jakarta.apache.org/bcel/.