Instruction Folding for an Asynchronous Java Co-Processor

Tero Säntti and Juha Plosila
{teansa | juplos}@utu.fi
Communication Systems Laboratory
Department of Information Technology
University of Turku 20014 Turku Finland

***Abstract*— This paper presents a novel method for instruction folding in Java execution. The approach is to use an asynchronous co-processor for execution of Java bytecode. The folding is done in the same pipeline stage with instruction decoding. The co-processor is designed using asynchronous techniques to provide low power usage with reasonable performance. The co-processor can be used in a single CPU and single co-processor environment or in a network of multiple CPUs and co-processors. The co-processor does not need to know what kind of environment it is placed in, as all communication goes through an interface unit designed especially for that environment. This modularity of the design makes the co-processor more reusable and allows system level scalability. This work is a part of a project focusing on design of an advanced Java co-processor for Java intensive SoC applications.**

I. Introduction

Java is very popular and portable, as it is a write-once run-any-where language. This enables coders to develop portable software for any platform. Java code is first compiled into bytecode, which is then run on a Java Virtual Machine (hereafter JVM). The JVM acts as an interpreter from bytecode to native microcode, or more recently uses just in time compilation (JIT) to affect the same result a bit faster at the cost of memory. This software only approach is quite inefficient in terms of power consumption and execution time. These problems rise from the fact that executing one Java instruction requires several native instructions. Another source for inefficiency is the cache usage. As the JVM is the only part of software running natively, it occupies the instruction cache, whereas the Java bytecode is treated as data for the JVM, hence being located in the data cache. Also the actual data processed by the Java code is assigned to the data cache. This clearly causes more memory accesses missing the cache. When the execution of the bytecode is performed on a hardware co-processor this is avoided and the overall amount of memory accesses is reduced.

This work is a part of the REALJava [3] project, which aims to design a Java co-processor that is easily integrated to various systems. We have chosen to use asynchronous techniques in this project because then we can achieve good performance with reasonable power consumption and very easy integration with existing systems, since no clock limitations need to be considered. Asynchronous self-timed circuit technology [6], where timing is based on local handshakes between circuit blocks instead of a global clock signal, provides a promising platform for obtaining a highly modular low-power and low-noise Java accelerator implementation.

Overview of the paper We proceed as follows. In Section 2 we shortly describe the structure of any JVM, and show how the proposed co-processor fits into the specifications. Section 3 describes the pipeline structure. In Section 4 the instruction folding is explained, and Section 5 shows the folding unit in more detail. Finally in Section 6 we draw some conclusions and describe the future efforts related to the REALJava co-processor.

II. Generic Java Virtual Machine Structure

In the Java Virtual Machine Specification [5], Second Edition the structure and behavior of all JVM's is specified at a quite abstract level. This specification can be met using several techniques. The usual solutions are software only, including some performance enhancing features, such as JIT (Just In Time Compilation). We have chosen to use a HW/SW combination [3] in order to maximize the hardware support and minimize the power consumption.

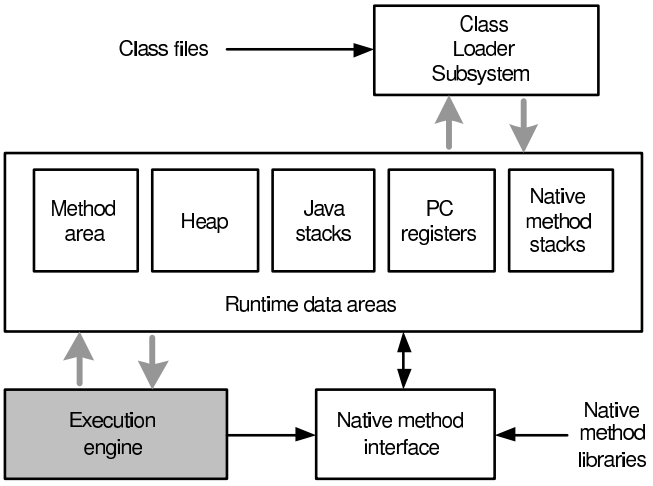

Fig. 1. Internal architecture of the JVM

The HW portion (highlighted in Figure 1) handles most of the actual Java bytecode execution, whereas the SW portion takes care of memory management, class loading and native

method calling. This partitioning gives the possibility to use the co-processor with any type of host CPU(s) and operating systems, as all of the platform dependent properties are implemented in software and (most of) the common bytecode execution is done in hardware.

Because Java supports multithreading at language level, it makes sense to integrate several co-processors as a SoC. This gives an ideal solution for complex systems running several Java threads and possibly some native code at the same time. This approach brings forth true multithreading and thus improves performance. Also large systems possibly contain several software subsystems, such as internet protocols, user interface controllers and so on, these can easily be coded in Java, and since they all are executed in parallel the user experience is enhanced. The multithreading also improves the predictability of the real time performance, as the threads do not get any wait states and the caches and stacks for each thread are kept intact inside their respective JPUs. Also the addition of timers gives the programmer new methods for controlling timing of execution with real time events.

The system architecture can be chosen to be a network of any kind or bus based, as suitable for other components in the system. The structure of the underlying network or bus is rather irrelevant, as long as the lower level provides two properties: 1) the datagrams must arrive in their destination in the same order that they were sent, and 2) the datagrams arriving from two different sources to a same destination must be identifiable. The first property can be be achieved with a lower level network protocol, like ATM adaptation layer (AAL) for internet, or by the physical structure of bus. The example we use here is a pipelined bus structure which guarantees the order of the datagrams by structure. The second property seems quite natural, and should be present in all solutions. For demonstration purposes we have chosen to use the pipelined bus [4], since it provides a good platform for multiple processing units accessing the bus simultaneously. The structure is shown in Figure 2. A simple yet efficient protocol for this case is given in [7].

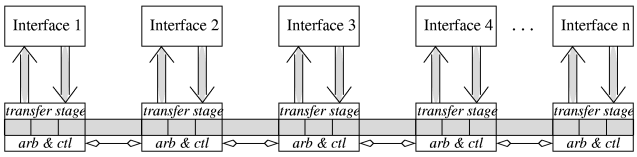

Fig. 2. Detailed view of the pipelined bus with the interface units.

III. Pipeline Structure

The pipeline structure of the co-processor differs from the structure normally used for processors. This is due to the fact, that normally the instruction set of a processor is engineered with hardware implementation in mind, but this is not the case for Java. The Java bytecode is designed to be executed in software, resulting in several significant differences. Additionally the bytecode instructions are based on a stack, instead of the normal processor approach of using several registers. This calls for optimizations not seen in conventional processor design. The pipeline architecture is covered in more detail in [8].

The Java Virtual Machine Specification states that the JVM has no internal registers, instead the temporary and working data is stored in a stack. Normally the software coder can improve performance by reordering the register accesses to keep the pipeline flowing, but in Java this is not possible, since all instructions that manipulate data are based on the stack. This situation is comparable with a normal processor architecture with only one register available to the programmer. This would keep the pipeline stalled for a large portion of the time, because of data dependency issues. To keep our pipeline in effective use, we have modified the normal pipelining strategy to better suit the stack based operation.

In the Figure 3 a simplified view of the pipeline structure is shown. The PC and OP_TOP labels stand for program counter and stack top, respectively. The boxes below those labels show how they are moved along the pipeline, to keep the values correct with the instructions related to them. The pipeline control unit sends a halt command to all pipeline stages upon receiving an external halt command or a halt request from the fold and decode unit. The fold and decode unit is required to have halt access to facilitate pipeline halting when a software handled instruction is encountered. After the whole pipeline is idle, the pipeline control sends an IRQ to the host processor.

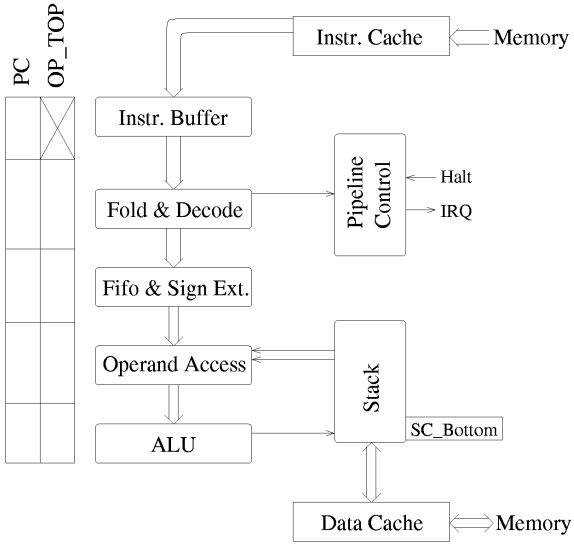

Fig. 3. A Simplified view of the pipeline.

The ALU contains the write back stage. The write back stage is included to the ALU because the bytecode instructions are based on the stack. One might wonder what this has to do with selecting the pipeline stages, but the answer is rather simple. In Java bytecode the instructions take the operands from the stack and write the result back to the stack. This causes the “normal” pipeline structure to generate huge amounts of wait-states to move the data to and from the stack.

LV	A local variable load, a load from a global register or a push constant
OP	An operation that uses the top two entries of the stack and produces a one word result which is stored on the top of the stack.
OP1	An operation that uses the topmost element of the stack and breaks the group
OP2	An operation that uses the top two entries of the stack and breaks the group.
MEM	A local variable store or a global register store
NF	Non-foldable instructions
TRAP	An instruction which is trapped by the hardware and is executed in software instead.

TABLE I
INSTRUCTION CLASSES

Thus the execution in the ALU will be often halted while the data is moved back and forth.

IV. Principles of Instruction Folding

The instruction folding is performed in order to remove unnecessary cycles in ALU and also to minimize redundant stack accesses. These performance hindrances are caused by bytecode instructions first pushing a value to the stack and immediately popping it out for processing. The folding procedure removes these two instructions, and replace them with one instruction carrying the value and the processing instruction to the ALU in one cycle.

With the instruction classes presented in Table I we can fold instructions in patterns shown in Table II. These patterns all produce VLIW instructions with up to two literal data elements, an opcode and a destination identifier. It can be noticed that the maximum length of folding is four instructions. This however does not mean "only" four bytes in the original bytecode stream. The original stream may have had some literal data included, and these are also placed in the VLIW, as shown in Figure 5. According to our preliminary analysis the reduction in the number of executed instructions was between 26.8% and 33.3%. In stack accesses the reduction varied between 39.3% and 51.2%. This analysis was run on a modified version of SableVM [1]. SableVM is licensed under the GNU Lesser General Public License (LGPL), and it can be obtained from "http://sablevm.org/".

Pattern	Instructions
LV LV OP MEM	4
LV LV OP	3
LV LV OP2	3
LV OP MEM	3
LV OP	2
LV OP1	2
LV OP2	2
LV MEM	2
OP MEM	2

TABLE II
POSSIBLE FOLDINGS

The fact that the whole co-processor is asynchronous helps us in the folding. In asynchronous circuits the blocks can run at independent speeds. This means that the folding unit can perform for instance a maximum of n foldings per second, where as the ALU may be significantly slower, say $n/2$ operations per second. The negative effects of independent speed, such as synchronization delays can be reduced using an intermediate fifo. The timing marginal for folding is increased because with asynchronous techniques all units exhibit average case performance. This means that the ALU may complete some instructions (bit-wise OR, etc.) in very short time, whereas some instructions (32-bit multiplication) take a lot more time. Since folding may produce new VLIW (Very Long Instruction Word) instructions at the rate of 1/1 to 1/4 in comparison to the original bytecode stream, the fifo balances the effects of both folding and the average case performance of the ALU. In our architecture the fifo also performs minor tasks, such as sign extension and address calculation for local variable accesses.

V. Folding Unit in More Detail

The folding unit receives data from the instruction buffer. The instruction cache handles the actual memory accessing, so the instruction buffer needs only to access the cache. The address is generated at the instruction buffer. The buffer is active in communication towards the cache and passive towards the folding and decoding unit. The folding unit is active in both directions, towards the buffer and towards the decoded instruction fifo. The fifo performs sign extension on the data, if required. The fifo is passive in all directions, towards the folding unit and towards the register access unit.

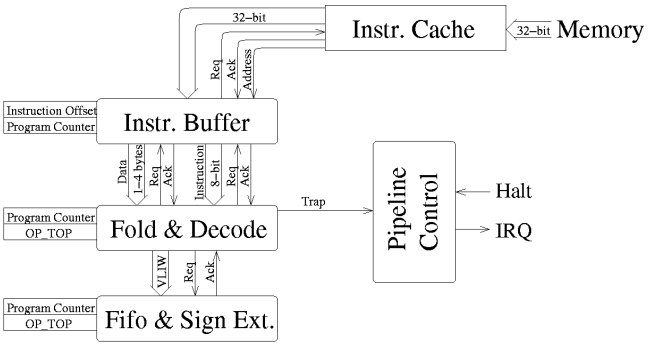

Fig. 4. The instruction folding and decoding pipeline.

The folding and decode unit has two communication channels to the instruction buffer. This is required because instructions may be followed by data, such as literal operand or an address. The amount of data can be found out only by decoding the instruction first. After the decoding is completed,

the correct amount of data bytes is read in parallel. The amount of data is between 0 and 4 bytes. If it is 0 bytes, no request is sent to the data read port. Since we read the data items in parallel to the fetch data –module shown in Figure 5, the instruction buffer can move the next instruction to the output end of the buffer without unnecessary delays.

After the instruction has been decoded and the data related to that instruction is read in, the next instruction is checked to see if it can be folded with the previous one. If it can be, then the procedure is repeated to see if the third instruction can be folded. If at any point the instructions can not be folded together, the previous instructions are sent out, and the procedure starts over with the current instruction as a base for new foldings.

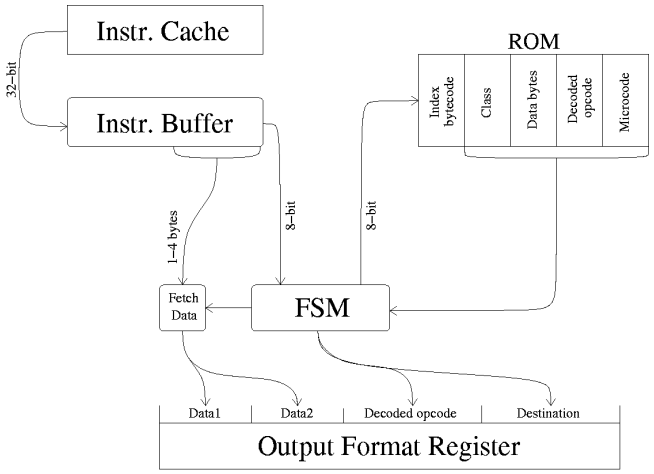

Fig. 5. The internal structure of the folding unit.

Figure 5 shows the internal structure of the folding unit. The FSM stands for Finite State Machine, which controls the operation of the unit. The ROM table approach is chosen, because Java bytecode is not optimized for hardware decoding. The instructions of Java bytecode are just listed in order and given the order number as an opcode. This would lead to a very complicated decoder, if implemented directly using standard logic elements. The ROM approach is also further validated by the fact that we can easily store microcode in the same table, as well as instruction classes and the number of data bytes related to a given instruction. This keeps our FSM simple and fast. All the entries in the ROM table are coded with one-hot scheme and the table is implemented as a precharged MOS NOR ROM matrix. The precharging is done when request is low, so the response time is minimal.

The output format register stores partial foldings, until they are completed. If a folding pattern is not terminated with a valid instruction for that pattern, the partial folding is executed one by one, and folding of the next instruction will be attempted. The register keeps record of which fields in it are valid at any given time. When a pattern is completed, the register pushes its contents to the fifo in the main pipeline, and prepares for a new folding autonomously.

VI. Conclusions and Future Work

A novel pipeline structure with instruction folding for Java execution was presented. The structure takes into account the peculiarities of the Java bytecode streams, and provides reasonable performance with low power usage. The instruction folding removes unnecessary stack accesses, thus eliminating needless power consumption as well as reducing execution time.

The approach chosen here is energy aware, even in comparison to running compiled C code on the CPU. This is achieved by using asynchronous circuit techniques. Asynchronous circuits excel especially in situations where the workload of the processing unit is not constant. Synchronous systems waste a lot of power by clocking internal latches even when no processing is done. This type of energy wasting is not present in asynchronous system. Also the current consumption of asynchronous systems (usually) is more stable, resulting in less noise.

We are currently investigating the potential benefits of integrating hardware timers to the JPU and expanding the JVM with own functions designed to make use of the timers. These functions would make it possible for users to execute pieces of code based on timer information. This would bring the real time performance of Java applications to a new level.

We plan to continue with designing the REALJava co-processor. The co-processor concept and hardware-software co-operation will be verified by building a FPGA demonstrator. At the same time dedicated memory structures supporting advanced garbage collection methods are investigated. After the FPGA phase we will continue manufacturing the co-processor as a separate ASIC. Later a larger SoC / NoC system with several CPUs and JPUs will be designed to implement a real-life application.

References

[1] E. Gagnon. “A Portable Research Framework for the Execution of Java Bytecode“, Ph.D. thesis, School of Computer Science, McGill University, Montreal, 2002.

[2] J. Hennessy and D. Patterson. “Computer Architecture: a Quantitative Approach”, Second Edition, Morgan Kaufmann Publishers, Inc., 1996.

[3] Z. Liang, J. Plosila, and K. Sere. “Asynchronous Java Accelerator for Embedded Java Virtual Machine”, *In Proc. of IEEE CAS Symposium on Emerging Technologies, Frontiers of Mobile and Wireless Communication*, Shanghai, China, June 2004.

[4] P. Liljeberg, J. Plosila, and J. Isoaho. “Self-Timed Communication Platform for Implementing High-Performance Systems-on-Chip”, *the VLSI Integration* Journal 38, Elsevier, 2004.

[5] T. Lindholm and F. Yellin. “The Java Virtual Machine Specification”, Second Edition, Addison-Wesley, 1997.

[6] J. Sparso and S. Furber. ”Principles of Asynchronous Circuit Design - A System Perspective”, Kluwer Academic Publishers, 2001.

[7] T. Säntti and J. Plosila. “Communication Scheme for an Advanced Java Co-Processor”, *In Proc. IEEE Norchip 2004*, Oslo, Norway, November 2004.

[8] T. Säntti and J. Plosila. “Architecture for an Advanced Java Co-Processor”, *In Proc. International Symposium on Signals, Circuits and Systems 2005*, Iasi, Romania, July 2005.