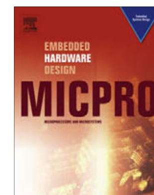
Microprocessors and Microsystems 33 (2009) 415–429

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro



Extending an embedded RISC microprocessor for efficient translation based Java execution ☆

Isidoros Sideris *, Kiamal Pekmestzi, George Economakos

Microprocessors and Digital Systems Laboratory, School of Electrical and Computer Engineering, National Technical University of Athens, 9 Heroon Polytechneiou, Athens 15780, Greece

ARTICLE INFO

Article history:
Available online 10 July 2009

Keywords:
Java processor
Codesigned virtual machine
Embedded Java

ABSTRACT

Java has gained great popularity in embedded appliances such as set-top boxes, smart phones and other hand held devices. In this paper we propose a translation based hw/sw codesigned Java virtual machine architecture, which extends a typical embedded RISC processor. The architectural extensions we propose include special instructions that accelerate translated blocks dispatch and security checks for arrays and objects. The extensions are done in a way that operating systems support is maintained, something that makes their adoption more attractive. Benchmarking using Embedded Caffeine Mark (ECM) benchmarks, showed significant speedups, especially when high performance RISC processors are employed.

© 2009 Elsevier B.V. All rights reserved.

1. Introduction

Java has been popular in the embedded systems market. Smart phones, set-top boxes and other embedded appliances are Java enabled. Java brings compatibility and security, but not at no cost. The extra software layer, which intermediates between the Java program binary and the underlying CPU, degrades performance significantly.

In PC and servers domain, aggressive dynamic translators which profile program execution and perform optimizations based on information collected in runtime, are in abundance. In embedded systems domain, just-in-time (JIT) compilers [15] can be employed, but not with too sophisticated optimizations, since their increased memory footprint and the greater translation cost may be prohibitive. Java processors [28,23,32] that execute Java bytecodes directly in hardware, are also a common solution. They provide similar performance, with reduced memory footprint, but sometimes suffer from operating systems support.

Additionally, the Java exception mechanism, which is a very useful and powerful language feature, impedes performance significantly, since the generated code contains redundant checks for violations. Considering the fact that the checks seldom result in exceptions, it would be useful to be eliminated. JIT compilers apply sophisticated optimizations which extract checks outside loops, or generate speculative code, which assume that no exception will happen and roll back in case it does [31].

Array violation checks occur frequently in DSP and multimedia kernels, since they are abundant in array accesses and it is a really limiting factor. Their elimination would be of vital importance.

In this paper, we present a translation based virtual machine architecture running on an embedded RISC processor, we propose some ISA extensions which accelerate the virtual machine execution and evaluate their impact on performance.

The translation optimizations that are applied are simple and therefore fast. On the other hand, the use of optimized target instructions (added as an ISA extension) results in quite efficient code. Thus, the codesigned virtual machine performs fast, since the generated code is kept efficient without applying sophisticated optimizations, which increase the translation cost greatly.

The proposed ISA extension consists of two subsets. The first subset contains instructions that diminish the overhead of security checks, since they incorporate them in hardware. In particular, there are special instructions for array handling that maintain the array sizes in a small cache and check for out of bounds violations in the execution stage. Furthermore, object accesses are accelerated by hardware checks for null pointer exceptions. In case an exception occurs, which is not the usual case, a trap is caused which is handled by a special handler in software. By eliminating the redundant checks important performance gains can be obtained.

The second subset contains instructions that support the dispatch between successive translated blocks, which usually impedes performance in translation based virtual machines. In such virtual machines, the Java program counter is mapped to the real addresses of the translated blocks using a software structure. A long search in such a structure at every dispatch is very detrimental to performance. To address this problem, we have included some special branch instructions which use a small fully associative cache to

☆ This work was partially funded by the Greek Ministry of Development, Project PENED03 ED-908.

* Corresponding author. Tel.: +30 2107723653.
E-mail address: isidoros@microlab.ntua.gr (I. Sideris).

0141-9331/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.micpro.2009.06.003

map Java PCs to real addresses [19,21]. We have explored the performance speedup for various sizes. For very large sizes the technique degenerates to translation chaining [30], a technique that augments every translated block with direct branches to the two possible successive translated blocks.

The extension of a typical RISC processor, instead of designing a dedicated Java processor, has the great benefit of operating systems support maintenance, which is of paramount importance. What is more, it does not incur great area increase.

In order to evaluate the performance impact of the proposed ISA extensions, we have used the SimpleScalar toolset [10] and conducted experiments using various processor configurations ranging from a typical scalar RISC processor to a 4-way superscalar out of order core. In a scalar RISC processor the extension results in a speedup of 5.13x. The absolute performance reaches 4.51 ECM/MHz in that configuration, while in a 2-way out of order core and a 4-way out of order core performance reaches 7.84 ECM/MHz and 9.17 ECM/MHz respectively, something that shows that the translation method generates code which can be parallelized efficiently.

1.1. Outline

The rest of the paper is organized as follows. In Section 2 the virtual machine framework used in this study is presented. In Section 3 we present the proposed ISA extensions for array and object accesses, while in Section 4 we introduce the special dispatch instructions. Section 5 discusses some complemental hardware support. Section 6 presents experimental results. Section 7 lists some related work. Finally, Section 8 concludes the paper.

2. Virtual machine framework architecture

The execution of Java programs in the proposed system is based on just-in-time translation of bytecode blocks. The host hardware platform is an embedded RISC processor and the translation transforms bytecode blocks into RISC instructions for efficient execution.

2.1. Overview of dynamic translation

Interpreting bytecodes is slow. The JVM interpreter must fetch, decode and execute each bytecode. Consider for example the bytecode sequence of Fig. 1. The first instruction loads the local variable *y* onto the stack, the second loads the constant 2, while the third the local variable *z*. The `imul` instruction extracts the two topmost values from the stack, and pushes their product to the stack. Similarly, the `iadd` instruction adds the two values and pushes the result to the stack. The `istore` instruction stores the end result in the variable *x*. The overhead of interpreting such a sequence is obvious. A typical interpreter implementation consists of a loop which fetches each bytecode and depending on its opcode value dispatches to the implementation of the bytecode. Such an implementation is poor, since for each bytecode we execute lots of host machine instructions. What is more, the branch in the interpreter loop cannot be predicted efficiently by branch predictors.

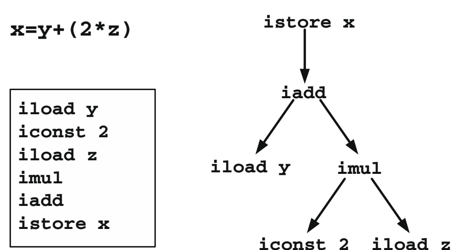


Fig. 1. Bytecodes example.

To address this problem, just-in-time (JIT) translators have been proposed [15], which dynamically translate bytecode sequences into native code just before execution. Compared to traditional static compilers, JIT compilers must perform translation at a much faster rate. They cannot afford to spend much time for performing optimizations, since the translation overhead should be optimized in short time by faster code execution.

Generally, dynamic translators preserve an intermediate representation of source blocks. However, the stack based nature of the JVM instruction set constitutes a convenient representation, since there are no register identifiers, but only an operand stack. Returning to the example of Fig. 1, we can see that the data flow graph extraction is straightforward. This simplicity has enabled even hardware decoding of bytecode sequences for speeding up the translation into native instructions [23].

2.2. Identifying dynamic blocks

One design decision that must be taken is that of selecting the delimiters of the blocks, that is finding where to start and where to end a source block of bytecodes. One convenient solution is to start blocks at branch targets and end them in branch or call instructions. Consider for example the control flow graph of Fig. 2a. Each node is a basic block, that is a block with one entry point, one exit point and no branch target contained in it. The blocks we use may consist of many entry points. For example the sequence DEG is one such block. The sequences FG and CG are two other examples. Since some such blocks may have more than one entry points, and we cannot branch in the middle of a translated block, we duplicate their basic blocks as shown in Fig. 2b. The resulting blocks are superblocks with only one exit. Practically, we perform translation for each block which starts from a not yet encountered branch target and ends in the first branch instruction.

2.3. VM dispatch

The dispatch mechanism of the proposed virtual machine is shown in Fig. 3. The emulation manager maintains a map table which maps Java bytecode blocks addresses to addresses where the corresponding translated blocks are kept. Whenever a branch or call instruction is executed, the emulation manager checks if there is a mapping in the table, and if so, it dispatches execution to the corresponding translated block. Otherwise, it calls the translator in order for the bytecode block to be translated. After the translation, the map table is updated and the newly translated block is executed. The blocks always end in a control flow instruction. The emulation manager intervenes so as to dispatch execution between successive blocks.

2.4. Instruction folding

Besides the benefits of the stack based ISA in handling bytecodes, it does cause serious performance degradation, since every bytecode instruction depends on the other due to the stack. Instruction folding algorithms have been proposed, which convert bytecode sequences into RISC instructions, thus removing stack accesses. Consider for example the bytecode sequence of Fig. 4a, which consists of two `iload` instructions which load two local variables onto the stack, the `iadd` instruction which performs an operation and an `istore` instruction, which stores the end result in one variable. By keeping the local variables in registers, we can substitute these four bytecodes into one RISC instruction which can be executed in just one cycle. This idea has been

Download English Version:

https://daneshyari.com/en/article/463162

Download Persian Version:

https://daneshyari.com/article/463162

Daneshyari.com