Available online at www.sciencedirect.com

ScienceDirect

Journal of Systems Architecture 54 (2008) 707–728

JOURNAL OF SYSTEMS ARCHITECTURE

www.elsevier.com/locate/sysarc

A predecoding technique for ILP exploitation in Java processors ☆

Isidoros Sideris *, Kiamal Pekmestzi, George Economakos

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

Received 27 March 2007; received in revised form 25 September 2007; accepted 28 January 2008
Available online 20 February 2008

Abstract

Java processors have been introduced to offer hardware acceleration for Java applications. They execute Java bytecodes directly in hardware. However, the stack nature of the Java virtual machine instruction set imposes a limitation on the achievable execution performance. In order to exploit instruction level parallelism and allow out of order execution, we must remove the stack completely. This can be achieved by recursive stack folding algorithms, such as OPEX, which dynamically transform groups of Java bytecodes to RISC like instructions. However, the decoding throughputs that are obtained are limited. In this paper, we explore microarchitectural techniques to improve the decoding throughput of Java processors. Our techniques are based on the use of a predecoded cache to store the folding results, so that it could be reused. The ultimate goal is to exploit every possible instruction level parallelism in Java programs by having a superscalar out of order core in the backend being fed at a sustainable rate. With the use of a predecoded cache of 2×2048 entries and a 4-way superscalar core we have from 4.8 to 18.3 times better performance than an architecture employing pattern based folding.
© 2008 Elsevier B.V. All rights reserved.

Keywords: Java processor; Stack folding; Predecoded cache; ILP

1. Introduction

Java with its simple object model and its platform neutrality has become popular in a wide range of applications, ranging from small embedded systems to high performance servers.

However, the existence of a virtual machine between the software application and the host software platform degrades dramatically the speed of execution.

The more naive method to execute Java bytecodes is to interpret them. However, there are several techniques [9,10,15] that are much more effective, and raise performance substantially.

In environments with general purpose processors, JIT [6] and adaptive compilation techniques dominate. These techniques profile method execution and dynamically compile the most often ones. The overhead of the compilation is amortized in short time.

Although this technique is dominant in PC and server environments, it cannot be applied to embedded systems, because it demands considerable amount of memory and processing speed. Java processors [14,18] were adopted to accelerate Java applications on these devices. These processors execute the Java bytecodes directly in hardware. However, they do not exploit instruction level parallelism, because they are mainly stack machines. The serialization incurred from the stack, poses significant limitations to the instruction parallelism.

PicoJava [14,4] tries to eliminate stack accesses by folding groups of instructions to a RISC like operation. More complex folding methods exist, as OPEX [8,7,12], which performs nested folding, compared to picoJava which only performs folding based on patterns.

OPEX removes the stack completely by performing nested folding in the decode stage of the processor. The

☆ This work is part of the PENED 03ED-908 project, co-funded by National and Community Funds (25% from the Greek Ministry of Development-GSRT and 75% from E.U.-European Social Fund).

* Corresponding author. Tel.: +30 2107721800.
E-mail address: isidoros@microlab.ntua.gr (I. Sideris).

1383-7621/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.sysarc.2008.01.008

decoding unit is coupled with the execution unit. OPEX processor folds instructions in the execution path. Its main drawback is that the folding rate that it achieves is not sufficient to feed a superscalar backend core [19], which may make use of all the available instruction level parallelism.

Another approach [5] tries to extract traces of instructions that are independent and can be executed in parallel. The detection of the traces is performed by finding the clean stack points, namely the points where the stack consists of zero elements. Obviously the instructions that follow such a point are not dependent on the previous, as they do not use any value from the stack. The main drawback of this method is that the marking of the traces is performed in software, during the verification stage.

In our work, we propose a hardware folding technique that dynamically transforms Java bytecodes groups into RISC instructions, storing them in a cache so as to enable reuse. The core folding algorithm is based on OPEX, modified appropriately to make reuse feasible. The method is like a simple hardware JIT compilation method, but in a fine grained form. It compiles blocks of code between successive clean stack points, and stores the RISC products in a cache for later use. The cache can provide the execution unit at a very high rate, not letting it to starve.

The technique has the benefit over normal JIT compilation, that we do not experience the latency of compiling a whole method. The fine grain translation is done in few cycles, so the user does not experience high start up time, something common in embedded JIT enabled virtual machines. Additionally, the memory requirements are fixed (only the predecoded cache is used) as opposed to JIT solutions which usually result in memory bloat. Furthermore, the translation is seamless to the user. It is a dynamic optimization technique in hardware.

The idea of caching the folding results has appeared in [17] as an extension to picoJava II processor. In combination with a stack disambiguation technique that differentiates operand stack accesses from local variable accesses, it has made feasible some level of parallelism. However, the folding is pattern based, and the architecture does not permit out of order execution.

Our architecture aims at full exploitation of the instruction level parallelism in Java programs. We do so by removing completely the stack by folding even nested folding groups. Due to the limited folding throughput, we store the folding results in a predecoded cache, so as to provide a superscalar backend core at an efficient rate [20]. Our folding algorithm is based on OPEX, modified appropriately to enable reuse of the folding results.

As a variation we place the folding unit out of the critical path of the processor, so as not to experience great initial latencies. Having the folding in the fill side of the processor, more sophisticated techniques could be applied. A straightforward modification is to combine many predecoded cache lines into a trace cache entry, at the expense of greater complexity.

We have used specjvm98 [3] benchmarks for our evaluation and we have investigated our technique for different cache configurations. Assuming a 4-way superscalar out of order core as the execution unit and microcode interpretation of the complex object oriented Java bytecodes, we have evaluated the performance of our architecture in terms of Java bytecodes per cycle and RISC instructions per cycle. With a predecoded cache of 2×2048 entries we have seen performance of 4.42 Java bytecodes per cycle with a corresponding RISC throughput of 3.19 RISC instructions per cycle. The pure decoding throughput of this architecture reaches 6.39 Java bytecodes per cycle, which compared to the corresponding rate of an architecture with pattern based folding (1.15), constitutes a 453% speedup.

The architecture with the folding unit in the fill side of the processor performs slightly worse, but it has the real benefit of relaxing the hardware design of the folding unit. The incorporation of a trace cache is expected to enhance the performance considerably.

1.1. Outline

In the following section we describe the core folding algorithm of our technique and an overview of our architecture. Section 3 presents our caching mechanism and the incorporation of the cache in the processor fetch stage, while Section 4 introduces our simulation environment and presents experimental results. The last section summarizes our work.

2. Folding

Our architecture is designed to reuse the folding results by keeping them in a predecoded cache. The folding takes place in the first occurrence of a Java bytecodes sequence. The next requests of that sequence are highly likely to hit in the cache. The folding algorithm we use is based on OPEX [8], modified appropriately to enable arbitrary break of folding results, so that they can be stored in a cache. It is a recursive stack folding algorithm, that is it folds nested folding groups, thus it removes the stack completely. Our folding algorithm differentiates from OPEX in that it does not follow the execution path as OPEX, but it handles Java bytecode sequences delimited appropriately.

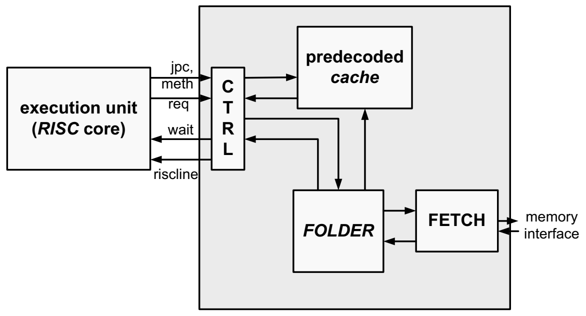

Fig. 1. Architecture overview.