A Decoupled Translate Execute (DTE) Architecture to Improve Performance of Java Execution

Ramesh Radhakrishnan and Lizy Kurian John
Laboratory for Computer Architecture
Department of Electrical and Computer Engg.
The University of Texas at Austin, Austin, Texas 78712
{radhakri,ljohn}@ece.utexas.edu

Abstract

Java is increasing in popularity in the software industry, and is being used to implement server and enterprise scale projects. Such applications require high performance, and need to run efficiently. Techniques like JIT compilers and PicoJava chips offer some speedup, but in this paper we look at alternate techniques in hardware which could be incorporated in future microprocessors. We propose a Decoupled Translate Execute (DTE) Architecture, which takes advantage of run-time characteristics of Java. The DTE architecture can be implemented using a mix of hardware and software or alternatively purely in hardware.

1 Introduction

Java technology has been adopted by the enterprise to offer software solutions in various fields, due to its advantages of portability, security and modularity. A factor which affects the execution speed of Java is that it is still predominantly a software emulated language. A program written in Java is translated (by software emulation) to an architecture-neutral instruction set called bytecodes. These bytecodes are platform independent and can be executed on any system which supports the Java runtime environment. Typically the bytecodes are interpreted or converted to native instruction set of the machine by a Just-in-Time (JIT) compiler.

Translation consumes a major part of the cycles during Java execution using a JIT compiler [1]. In many of the SpecJVM98 programs, approximately half the time is spent in compile time operations and only half of the time is spent in execute related operations [1]. The data cache miss rates during execution with a JIT compiler were seen to be dominated by compulsory write misses, which were arising from installation of the translated code. Another phenomenon is that the translated code is deposited into the data cache and later fetched from the instruction cache, results in avoidable data transfer and double-caching. In this paper, we present the Decoupled Translate Execute (DTE) Architecture which forms a solution to both of the above problems.

2 The DTE Architecture

Translation and *Execution* are two distinct operations occurring during Java execution. While interrelated, there is significant potential overlap that can be exploited between these two operations using a Decoupled Translate Execute Architecture (DTEA) along the lines of the traditional DAE architectures [2, 3, 4]. Figure 1 illustrates the structure of the proposed decoupled architecture, which consists of the execute and translate processors and necessary buffers for interprocessor communication. The bytecode executable will enter the Execute Processor (EP) and the execute processor will check whether this method is available in the Translated Code Cache (TCC) in the translated form. If it is not found in the TCC, it will deposit the methods to be translated into the E-to-T queue. The Translate Processor (TP) will translate the code and put the translated methods into the TCC. The translate operation progresses concurrently with execution, once a few routines have been translated. The execution proceeds as long as the next required method has been translated and deposited into the TCC. The granularity for translation can be bytecode or *method.* Although queue might simplify control, a cache-like structure can help method reuse. Hence we are using a cache-like structure for TP to EP communication and a simple queue structure for EP to TP communication.

Unlike DAEs, splitting code into translate and execute threads is easier and straightforward due to the inherent threaded nature of Java code. The improvement obtained using such a decoupling strategy will

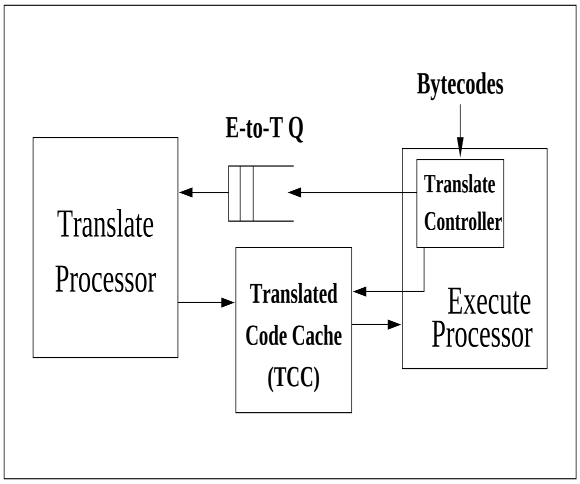

Figure 1: The Decoupled Translate Execute Architecture (DTEA)
E-to-T Q is a FIFO buffer between the Execute processor (EP) and the Translate processor (TP), used by the EP to communicate methods/bytecodes to be translated to the TP. TCC is a cache used by TP to communicate translated methods/bytecodes back to the EP. It stores the translated methods/bytecodes and facilitates method reuse.

depend on the overlap that can be attained between translate and execute operations. In traditional DAE terminology, this would be called *slip*. The distinctly separate threads required during Java execution provides potential for significant overlap and parallelism.

The translate processor may use software translation or hardware translation. Software translation consists of exactly the same operations that happens in current JITs during the translate phase. Hardware translation would involve template matching and generation of translated code as modern x86 processors generate ROPs(RISC ops) or UOPS(RISC like micro-operations). A multithreaded processor utilizing different threads for translation and execution may be considered as a software implementation of the DTEA.

3 The Hard DTE Architecture

In this section we describe a subset of the DTE Architecture, the "Hard DTE" Architecture. In the Hard DTE architecture the translation of bytecodes to native code is done through a hardware translator, as opposed to using a software emulator. This provides advantages of speed in translation, but at the expense of increasing design complexity and chip area.

Software emulation of bytecodes (interpretation, JIT compilation) is slow. Our studies on Java workloads [5] show that interpreted execution requires on the average 35 SPARC instructions to emulate each bytecode. The average SPARC instructions required to emulate a bytecode was approximately 20 when using a JIT compiler [6]. Software emulation is easy to implement for new platforms but cannot offer a solution for fast execution of Java bytecodes. Using a hardware solution, which implies executing the bytecodes directly in hardware would eliminate the requirement of a software layer to emulate the bytecodes.

There are several issues to be considered while employing hardware to perform translation. The current hardware implementations of the Java Virtual Machine (JVM) such as PicoJava are stack based. However, the Instruction Level Parallelism (ILP) which can be exploited using a stack architecture is limited. Figure 2 shows the percentage reduction in execution cycles and improvement in instructions per Cycle (IPC) for Java applications executed using the interpreter and the JIT compiler. The programs executed using the JIT compiler are seen to show slightly higher improvements in both execution time and IPC when executed on 4-way and 16-way processors. The individual data for each benchmark can be found in [1], and we include Figure 2 in this document to serve as a motivation for hardware translation into a non-stack ISA. Our simulator models realistic branch prediction, however, assumes a perfect Branch Target Buffer (BTB) and therefore the performance of the interpreted programs will be still worse on a real machine in which the BTB is modeled (since the interpreter has more indirect branches than the JIT mode of execution).

Based on results of preliminary performance estimates, we favor a hardware implementation of the DTEA concept. Both EP and TP can be built on the same chip and the architecture can be very similar to modern dynamically scheduled microprocessors except for a few building blocks. This architecture will include a hardware unit to convert the bytecodes into simple RISC like instructions. These converted instructions will be register based, thereby enabling use of standard techniques to obtain high performance. The Translated Code Cache (TCC) will store the converted bytecodes. The TCC will be very similar to a Trace cache [7, 8], in the sense that it captures the dynamic translated code during program execution. A high-level block diagram of a system with the proposed hardware units is shown in Figure 3. Instead of fetching bytecodes from the standard instruction cache, whenever possible, the converted bytecodes from the decoded bytecode instruction cache will be directly fetched and executed by the processor core. All blocks in Figure 1, except the highlighted blocks are part of any state of the art micro-

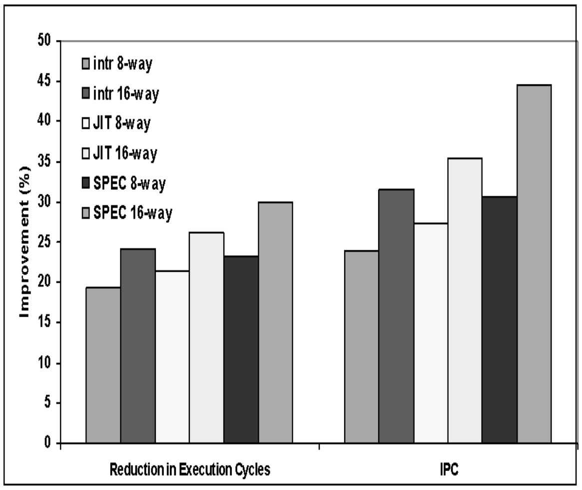

Figure 2: A comparison of scalability during interpreted and JIT compiled execution.
Percentage reduction in execution cycles and improvement in IPC relative to a 4-way configuration is shown. The interpreted code is increasingly stack-oriented, while the JIT code incorporates register optimizations. (This figure is provided as motivation for non-stack ISA in the core of Java processors.)

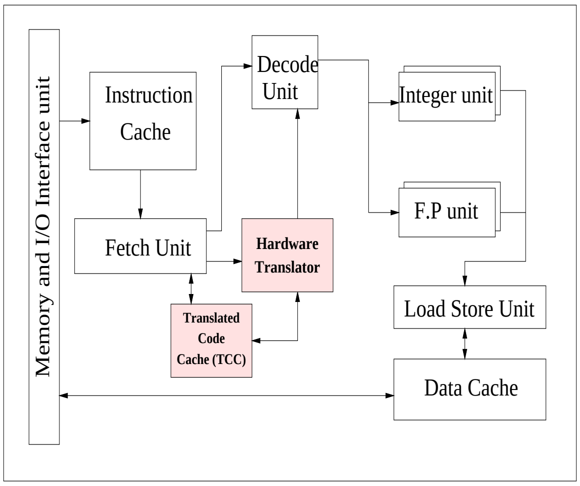

Figure 3: The Hard-DTEA implementation - microarchitecture of a decoupled processor that performs translations in hardware

processor. Although not explicitly indicated, there are queues between various blocks in Figure 3, especially between the fetch unit and the Hardware Translator. Storing the translated bytecodes/methods in the TCC also provides opportunity for a variety of optimizations as performed by the fill-unit in conjunction with a trace cache [9].

3.1 Fill Unit approach to decoding bytecodes

The block diagram of the front-end for the Hard DTE is shown in Figure 4. The hardware translator/decoder will convert bytecodes into smaller fixed length instructions, which will be packed into the Translated Code Cache (TCC) by the fill unit. A line in the TCC can be finalized or completed when one comes across a control flow instruction. The branch unit will provide the next instruction address, and it is fed back to the instruction cache and the TCC. If the lookup in the TCC results in a hit, then the decoded instructions are fed to the dynamically scheduled micro-engine directly from the TCC.

The fill unit and the TCC help in creating larger atomic units of work, which can be fed to the dynamically scheduled micro-engine. Increasing the fetch bandwidth, using the hardware translator and the TCC will allow us to execute the bytecodes in a superscalar

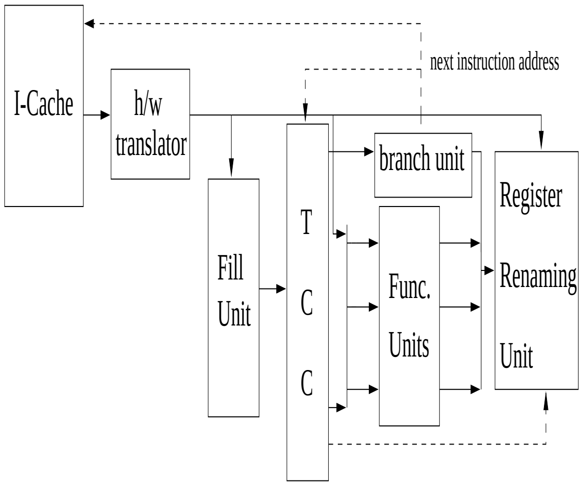

Figure 4: Block diagram of the Hardware DTEA with an instruction fill unit

fashion, unlike the PicoJava which did not execute instructions out of order as the instruction fetch bandwidth was very limited. The PicoJava fetches only 1-2 bytecode instructions every cycle. The stack creates an implicit dependency amongst the bytecodes and the variable length of the bytecode instructions make it harder to decode multiple instructions simultaneously.

3.2 Decoding bytecodes into micro-ops

A hardware decoder will convert the bytecodes to multiple micro-operations. A decoding structure similar to the Intel P6 decoder can be used, which can decode multiple simple and one complex instruction in one cycle. Another approach is to decode only the commonly used bytecodes and store them in the TCC, and to execute the other bytecodes by trapping to micro-code or software.

Consider the bytecode, `dup2`. This bytecode duplicates the top two elements of the stack and pushes them on the stack. This bytecode can be split into the following five micro-operations:

```
TOSB <--  [SP-4] ; load TOS (top of stack)
TOSA <--  [SP-8] ; load TOS-1
SP <-- SP + 8    ; TOS = TOS + 2
[SP-4] <-- TOSA  ; store new TOS
[SP-8] <-- TOSB  ; store new TOS-1
```

In decoding this bytecode, we used 2 temporary registers TOSA and TOSB. We also modified the stack pointer (SP). The bytecode required 2 stores, 2 loads and 1 computation instruction and used 2 intermediete registers to store temporaries. Similarly, we can calculate the load/store instructions and computation instructions required to implement different bytecodes to come up with a resource template which can be used during decoding.

4 Concluding Remarks

In this paper we propose the DTE architecture and also describe a Hard DTE, which is a subset of the DTE architecture. The DTE architecture takes advantage of the concurrency available at Java run time, during the *translate* and *execute* stages.

The DTEA architecture and in particular, the Hard-DTEA are being evaluated using the SpecJVM98 benchmarks. A set of Java micro-benchmarks are also used for validation purposes in the study. The ability of the DTEA concept to exploit concurrency between translation and execution, hide the cost of translation, and enable Java to achieve the performance of off-line compiled code without losing the advantages associated with JIT compilation is being studied.

References

[1] R. Radhakrishnan, N. Vijaykrishnan, L. John and A. Sivasubramanium, "Architectural Issues in Java Runtime Systems," Tech. Rep. TR-990719, Laboratory of Computer Architecture. Electrical and Computer Engineering Department, University of Texas at Austin, 1999. http://www.ece.utexas.edu/projects/ece/lca/ps/tr990719.pdf.

[2] J. E. Smith, " Decoupled Access/Execute Computer Architecture," in *ACM Transactions Computer Systems*, pp. 289–308, November 1984.

[3] L. John, P. T. Hulina, and L. Coraor, "Memory latency effects in decoupled architectures with a single data memory module," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 236–245, May 1992.

[4] L. John, V. Reddy, P. T. Hulina, and L. Coraor, "Program balance and its impact on high performance RISC architectures," in *Proceedings of the International Symposium on High Performance Computer Architecture*, pp. 370–379, Jan 1995.

[5] R. Radhakrishnan, J. Rubio, and L. John, "Characterization of Java applications at the bytecode level and at UltraSPARC-II Machine Code Level," in *Proceedings of International Conference on Computer Design*, October 1999. To appear.

[6] R. Radhakrishnan, J. Rubio, L. John and N. Vijaykrishnan, "Execution characteristics of just-in-time compilers," Tech. Rep. TR-990713, Laboratory of Computer Architecture. Electrical and Computer Engineering Department, University of Texas at Austin, 1999. http://www.ece.utexas.edu/projects/ece/lca/ps/tr990717.ps.

[7] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching," in *Proceedings of the 29th International Symposium on Microarchitecture*, Dec 1996.

[8] S. J. Patel, M. Evers, and Y. N. Patt, "Improving trace cache effectiveness with branch promotion and trace packing," in *Proceedings of the 25th International Symposium on Computer Architecture*, June 1998.

[9] D. H. Friendly, S. J. Patel, and Y. N. Patt, "Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors," in *Proceedings of the 31st ACM/IEEE International Symposium on Microarchitecture*, Dec 1998.