Java Microarchitectures

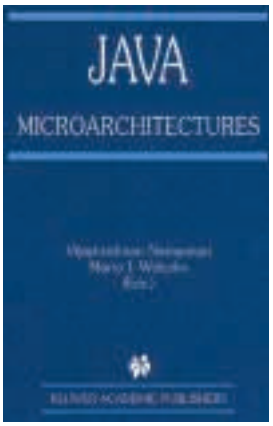
by: Purnendu Sinha, Nikhil Varma & Vasudevan Janarthanan, Concordia University, Montreal, QC.

Book Edited by: Vijaykrishnan Narayanan, Mario I.Wolczko
Published by: Kluwer Academic Publishers
ISBN: 1-4020-7034-9
2002

The need for an architectural neutral language that could be used to produce code that would run on a variety of CPUs under differing environments led to the emergence of the Java programming language. With the rise of the World Wide Web, Java has propelled to the forefront of computer language design, because the web, too, demands portable programs. The environmental change that prompted Java was the need for platform-independent programs destined for distribution on the Internet. The Java platform with its target platform neutrality, simplified object model, strong notions of security and portability, as well as multithreading support, provides many advantages for a new generation of networked, embedded and real-time systems. All those features would not have been possible without appropriate hardware support. This book delves in-depth into the various hardware requirements (with suitable case-studies and examples) for realizing the advantages of Java.

Java's portability is attained by compiling Java programs to Java bytecodes (JBCs) and interpreting them on the platform-independent Java Virtual Machine (JVM). Bytecode is a highly optimized set of instructions designed to be executed by the Java run-time system namely the JVM. The first chapter presents a platform-independent dynamic analysis of the JVM, including data related to bytecode instruction usage, method frequencies and stack frame profiles. In order to test the technique, the SPEC JVM98 benchmark suite has been used, since this suite does not allow the supply of source code to all the applications. This type of analysis helps one to clarify the potential impact of the data gained from static analysis, provide information on the scope and coverage of the test suite used and act as a basis for machine-dependent studies. Based on the results tabulated in this chapter, one can gain insights into the optimization work needed for improving the performance of Java to bytecode compilers and for the design of the JVM.

As Java is being used on a variety of platforms, there is a growing importance to study and optimize the memory behavior of programs because of the various disparities between the processor and memory speeds. Chapter 2 helps one understand the memory behavior of important Java workloads used in benchmarking JVMs like SPEC JVM98 and JIT compilers. With the help of the benchmark, characteristics like heap accesses, data misses, object-field accesses including hot spots and memory system interactions have been inspected. The analysis presented in this chapter provides important insights into understanding the key sources of performance loss in Java programs. This chapter also presents a set of recommendations to computer architects and implementers of JVM components with structured information about the Java workloads useful in formulating their designs and also techniques to run Java programs more efficiently.



In embedded application, the architectural support is a key factor for improving performance. The third chapter describes a hardware architecture that provides an efficient implementation of the JVM for embedded and real-time systems. As the proposed architecture provides direct support for the entire JVM instruction set and thread model, it obviates the need for a Java interpreter or JIT compiler as well as traditional RTOS. This chapter looks into the aspects such as memory management, concurrency, interrupts and concludes that the aJile embedded Java microprocessor provides an efficient platform for developing embedded applications in Java.

Java programs execute indirectly through a translation layer built into the Java Virtual Machine (JVM). The translation process essentially converts the bytecodes into corresponding machine-specific binary instructions. The JVM has a stack architecture where operands are executed one by one using the push-pop technique. Chapter 4 presents a processor architecture for the hardware execution of the bytecodes and resolves the issue of stack dependency by the use of a hardware bytecode folding algorithm. The architecture provides a dual processing capability of execution of bytecodes and native binaries. This chapter discusses and analyzes the bytecode processing in various stages of the instruction pipeline. It also presents a comparison between the hardware translation approach and the other hardware approaches supporting Java in hardware.

Using hardware support to assist the execution of bytecodes eliminates the requirement of a software layer to emulate the bytecodes. A hardware accelerator or coprocessor that works in conjunction with a standard microprocessor can improve the execution of Java programs. The fifth chapter introduces micro-architectural techniques to improve the performance of Java applications executing on embedded Java processors and general-purpose processors. The mechanism of using a fill unit to store decoded bytecodes into a decoded bytecode cache improves considerably the fetch and decode bandwidth of Java processors. A Hardware Interpreter (Hard-Int) architecture is also proposed which bridges the performance gap between the execution of Java applications and natively compiled code by dynamically translating the bytecodes to native language instructions.

Continuing along our discussion on the issue of design of architectures for efficient execution of Java Virtual Machine (JVM) bytecode, Chapter 6 describes the Delft-Java architecture and the mechanisms required to dynamically translate JVM instructions into Delft-Java instructions. This chapter also provides micro-architectural support for dynamic translation, dynamic linking, multiple thread units, multiple instruction issue, dependency collapsing and features applicable to modern super-scalar processors. Through examples, this chapter illustrates the effectiveness of Delft-Java architecture in accelerating the Java program execution.

Since Java Virtual Machines (JVMs) rely on dynamic compilation for performance, they suffer from large memory footprints and high startup costs, which are serious problems for embedded devices. The seventh chapter presents a Quasi-static compilation framework that enables efficient use of volatile storage while executing compiled Java code on an embedded device. In this approach, the pre-compiled binary images are reused and adapted to a new execution context by using an indirection table to hold relocated values, leaving the executable code unmodified and able to be placed in ROM. This approach enables the Java code to be stored in a shared location and be used by different applications.

With hardware support for multithreading, virtual machines (VM) can enhance performance, by exploiting thread-level concurrency, by executing tasks such as garbage collection (GC) concurrently with program execution. Chapter 8 discusses a Relational Profiling Architecture (RPA)

Continued on page 16

Book Review: Continued from page 12

that can be augmented to the existing hardware supports to enable VMs perform concurrent garbage collection. The proposed architecture helps the GC system to have a short pause time and a low average run-time overhead. This chapter also presents tabulated results of simulated times for various phases of GC for various available benchmarks.

It has been noted that the Java language provides unique opportunities to exploit parallelism by permitting architectures to execute single threaded applications as multi-threaded applications. The ninth chapter presents a technique called Space-Time Dimensional Computing (STC) for the execution of speculative threads extracted at the method and loop level from non-threaded Java programs. This chapter also provides hardware support to efficiently implement STC without introducing delays in critical paths for obtaining high frequency designs. Furthermore, this chapter describes an architecture named MAJC, which has been designed to support the STC technique.

In order to produce a single-chip multiprocessor and to provide support for high performance Java based systems of the future, the Java Machine and Integrated Circuit Architecture (JAMAICA) was designed. Chapter 10 discusses the design of Instruction Set Architecture (ISA) of JAMAICA that has on-chip multiprocessor structure targeted for multithreaded Java implementations. A selection of programs from the SPEC JVM98 benchmarks has been used to analyze the various ways in which bytecode can be executed and the resulting overheads that occur. This chapter also presents some optimization techniques to decrease the method call overheads and compares the effects of proposed optimizations on static instruction count for selected SPEC JVM98 kernels.

The JAMAICA system is a combination of a multithreaded single-chip multiprocessor and a dynamic thread distribution mechanism to provide hardware support for fine-grained Java threads. The last chapter provides an overview of the threading mechanism and investigates the granularity of parallelism that can be exploited in this way. This chapter also confirms through experiments with two real Java applications that the technique could be used in place of more traditional load balancing methods. The JAMAICA system considered in this chapter is a Container Managed Persistence (CMP) processor where each processor core is multithreaded keeping the processors always occupied.

Java provides unique opportunities to exploit parallelism by permitting architectures to execute single-threaded applications as multi-threaded applications.

In summary, this book provides a detailed analysis of hardware support for Java. In particular, it introduces the state-of-the-art in the area of design and development of Java micro-architectures. The book presents extensive simulation results covering different proposed architectures that could benefit practicing engineers and academic researchers alike in the design, implementation and evaluation of newer architectures. As Java-based technology is evolving, this book could be a valuable tool in understanding the impact of Java's features on micro-architectural resources.