BLP: Applying ILP Techniques to Bytecode Execution

Univ. of Virginia Dept. of Computer Science Technical Report
CS-2000-05
February 2000

Kevin Scott, Kevin Skadron

Dept. of Computer Science
University of Virginia
Charlottesville, VA 22911

Abstract. The popularity of Java has resulted in a flurry of engineering and research activity to improve performance of Java Virtual Machine (JVM) implementations. This paper introduces the concept of *bytecode-level parallelism* (BLP)—data- and control- independent bytecodes that can be executed concurrently—as a vehicle for achieving substantial performance improvements in implementations of JVMs, and describes a JVM architecture—*JVM-BLP*—that uses threads to exploit BLP. Measurements for several large Java programs show levels of BLP can be as high as 14.564 independent instructions, with an average of 6.768.

1 Introduction

Bytecode representations are portable formats that allow programs–whether small "applets" or large desktop applications–to run on different platforms. No source code modification or recompilation is necessary. Many web-based applications are already distributed this way, and bytecode representations could soon become a standard medium for distributing most applications. Unfortunately, current bytecode execution environments often exhibit poor performance, even on small applets. This poor performance is perhaps the greatest technical impediment to widespread use of bytecode representations.

The problem is that most bytecode execution environments, like implementations of the Java Virtual Machine (JVM), do not exploit parallelism in bytecode programs. Current approaches instead merely focus on speeding up the interpreted, but still serial, execution of bytecodes, or on improving the efficiency of just-in-time compilers (JITs). Because current interpreters still serialize bytecode execution, events like memory-reference delays entirely stall the interpreter, and bytecodes that are independent are unable to execute in parallel. JITs also suffer drawbacks. They usually only perform limited optimizations because time for more sophisticated analysis is not available. Furthermore, JIT systems often optimize only selected sections of code, leaving many segments to continue executing in the interpreter. JITs also stall on memory reference delays.

Regardless of how often the JIT is engaged, programmers may wish to disable it while developing or debugging code, just as C programmers typically turn off compiler optimization under these circumstances. Unfortunately, this removes all optimization, and performance reverts to that of a basic, interpreted JVM. Furthermore, some users may never enable the JIT under any circumstances, just as some C programmers never enable optimization. In such cases, the presence of the JIT offers no benefit.

A New Approach to Java Execution. An interpreter is really an instruction-execution engine implemented in software. Computer architects have developed a wealth of techniques to exploit parallelism among machine-level instructions—*instruction-level parallelism* or ILP—in hardware instruction-execution engines, namely CPUs. This paper describes a JVM architecture—*JVM-BLP*—that applies ILP techniques to bytecode execution to exploit *bytecode-level parallelism*, or *BLP*.

Our results show that bytecode programs indeed exhibit BLP, and that for the SPECjvm98 programs, the average degree of BLP can be as high as 11.883 bytecodes and averages 5.794. The potential for speedup is therefore staggering: if these independent bytecodes can indeed be executed in parallel, a JVM system that exploits BLP offers speedups as large as the degree of BLP. This is an upper bound: the goal of this research is to extract as much of this speedup as possible.

The JVM architecture we propose here can be implemented in a hardware-independent fashion, but best performance will come on platforms that can exploit thread-level parallelism and particularly on platforms that support *simultaneous multithreading* (SMT) [16]. This is because SMT can execute multiple threads simultaneously, each and every processor clock cycle.

Related Work. A great deal of work has explored ways to optimize the performance of bytecode interpreters [3, 4, 11, 12, 14] and to augment them with JITs [1, 5, 10, 18]. However, we have been unable to find any work describing the idea of bytecode-level parallelism or an architecture to execute multiple bytecodes concurrently. Instead, in order to motivate JVM-BLP, the next section describes the current state of the art in Java Virtual Machines, just-in-time compilers, and dynamic recompilation systems.

The next section describes some relevant aspects of JVM implementations. Section 3 describes a new JVM architecture that exploits BLP. Section 4 describes the experimental framework that we used to measure BLP and the benchmark programs that we selected. Section 5 presents our results and Section 6 presents our conclusions and suggestions for future work.

2 Java Virtual Machines

The Java Virtual Machine (JVM) is a stack-oriented abstract machine that executes Java bytecode programs [15]. Using bytecodes to encode instructions and an intermediate operand stack during execution allows for small object (class) files. The JVM and class file format together provide a platform-independent execution environment for mobile code.

In addition to target platform independence, the JVM is source-language independent. Even though the JVM was designed to run compiled Java programs and includes some features that facilitate the implementation of Java-like[1] languages, nothing prevents it from running bytecodes generated by compiling other languages. The JVM's well-defined semantics and interfaces, platform independence and ubiquity have led many language implementors to target the JVM [3, 4, 11, 12, 14].

2.1 JVM Interpreted Performance

From the start, Java has suffered from poor performance compared to traditional languages like C. Not all the performance issues are related to the JVM itself. For instance, a recent study has shown that excessive synchronization in the Java core libraries is one major component of poor performance [9]. Nevertheless, the JVM typically remains the biggest performance bottleneck. Early JVMs were implemented as pure bytecode interpreters and used old mark-and-sweep garbage collection technology.

A pure bytecode interpreter can be implemented entirely in a high level language. Bytecodes are stored in an array and indexed by a program counter. Each bytecode has an arm in a switch statement. Each arm contains the high level language code that implements the bytecode's semantics. A typical bytecode interpreter's main loop might look something like this.

```
for(;;) {
    switch(code[pc]) {
        case iadd:
            s1 = pop(); s2 = pop(); push(s1+s2);
            break;
        case iconst_0:
            push(0); break;
        case iconst_1:
            push(1); break;

        ...
    }
    pc++;
}
```

Implementing the JVM as a pure bytecode interpreter has two advantages. First, the implementation is quite straightforward. This technique has been employed for decades in the implementation of programming languages and is well understood. Second, since the whole interpreter can be written in a high level language, the JVM becomes as portable as the implementation language.

The drawback of pure bytecode interpreters is their poor performance relative to native machine code. This is particularly pronounced for bytecode languages that have many primitive operations like the JVM. With primitive operations, the overhead of looking up the next bytecode and dispatching to the arm that implements it is large with respect to the operation's actual computation.

There are two solutions to this problem. The first and most popular is to implement the JVM as a threaded bytecode interpreter [8]. Threaded bytecode interpreters eliminate the for loop and switch statement of the pure bytecode interpreter. The `code` array becomes an array of pointers to labels marking the code that implements individual bytecodes. At the end of each bytecode an indirect jump is made to the next bytecode's label. Here's how the threaded interpreter might look.

[1] By Java-like, we mean strongly typed, object oriented languages with exceptions and automatic storage management.

```
iadd:
    s1 = pop(); s2 = pop(); push(s1+s2);
    goto *pc++;

iconst_0:
    push(0);
    goto *pc++;

iconst_1:
    push(1);
    goto *pc++;

 ...
```

Threaded bytecode interpreters do reduce the amount of overhead per computation. Although ANSI C does not allow the programmer to take the address of a label or to goto a location specified in a program variable, gcc has both of these features, and portability can still be preserved across any platform for which gcc is available.

Another technique that can reduce the amount of overhead per computation is the superoperator [19]. Superoperators merge together primitive operators that occur together frequently in real programs, implementing them as a single block of code. Superoperators can be used with or without threaded bytecode. The obvious disadvantage of superoperators is the increase in the number of operators that must be implemented in the virtual machine. Also, unless superoperator inference is done dynamically, the virtual machine instruction set would have to be changed. We are unaware of any JVM employing this technique.

Perhaps the most serious disadvantage of bytecode interpreters, pure or threaded, is that they do not make effective use of the underlying machine's hardware registers. The lack of global register allocation can be the largest single factor contributing to poor JVM performance [1, 10].

2.2 JVM JIT Performance

The research community and Java vendors have attacked the shortcomings of bytecode interpreters through the development of JIT compilers. In a JIT compiler, compilation occurs when a method is first invoked; its bytecodes are translated to machine code and then cached for later use. The JIT compiler is in complete control of how to use the processor's resources to effectively implement the semantics of the method's bytecodes. It is important to note that in many JVMs, the JIT only operates on portions of code that it deems worth optimizing, and remaining segments of code continue to be interpreted. The JIT may also be turned off by the user for easier debugging.

For some Java methods, JIT compilation is a huge win over bytecode interpretation. Yet in some instances it yields no benefits, and in other instances it actually hurts program performance. The bulk of current JIT compiler research focuses on how to decide when to dynamically compile a method and how much optimization to apply when compiling bytecodes to native machine code [1, 5, 10, 18].

JIT compilers, by necessity, have machine-dependent components. Minimally, a JIT compiler must have a machine dependent code generator that maps bytecodes onto native machine code. Although code generation can be factored and driven by formal descriptions of the bytecode mapping, there is still a non-trivial amount of software that must be rewritten for every platform to which the JVM is ported. In the case of aggressive optimizers, there may be additional non-portable code that performs machine specific optimizations.

An alternative compiler-based approach is to implement a dynamic compiler in the JVM, as in Sun's HotSpot system [21] or TransMeta's Code Morphing system [13]. Like a JIT, HotSpot performs compilation on the fly. The important difference is that HotSpot is a thread running in the background of the JVM, identifying code segments that are candidates for optimization, and optimizing these "hot spots". Although HotSpot may be a superior approach to on-the-fly compilation and optimization, it shares the same drawbacks as JITs, namely large volumes of machine-dependent code.

3 Proposed JVM Architecture

A faster interpreted-execution environment is desirable, because current bytecode interpreters are simple and portable, but execute bytecodes serially and are therefore slow. JIT compilers (JITs) are faster, but complex and not portable. Furthermore, JITs must sometimes leave sections of code unoptimized, and these code segments continue to be interpreted. In addition to these considerations, speeding up interpreted execution is attractive because interpreters have other uses, like verifying operational semantics and code safety, for which JIT compilers are not an option but for which the interpreter's performance remains important.

As mentioned in the introduction, an interpreter is really an instruction-execution engine implemented in software. Computer architects have already developed a wealth of techniques to exploit parallelism among machine-level instructions (instruction-level parallelism or ILP). This section describes an architecture for a JVM interpreter that that breaks the serial-execution bottleneck by exploiting *bytecode-level parallelism*, or *BLP*.

Figure 1 shows a diagram of the proposed JVM architecture. We call this system the *JVM-BLP*: a JVM that exploits BLP. Our first implementation will be a JVM, but this same architecture can easily be generalized to most interpreted languages.

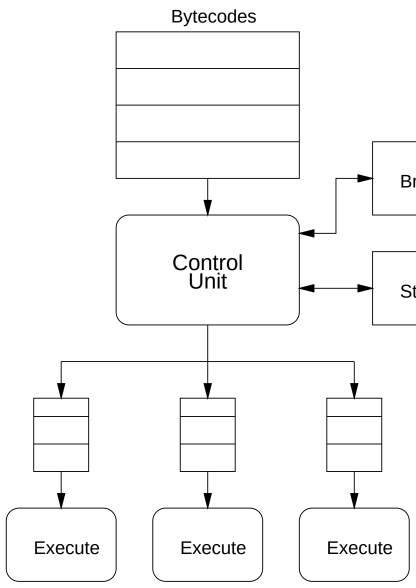

Fig. 1. JVM-BLP Organization

The principal ideas are remarkably similar to those pervading computer architecture research. The diagram shows a **control unit** that reads bytecodes from the method currently executing.

When processing a new bytecode, the control unit maps stack locations to **virtual registers** using the stack renaming unit. For example a JVM $iadd$ operation might get translated to $vr_2 \leftarrow iadd(vr_1, vr_0)$. This means that $iadd$ now gets its operands from virtual registers vr_1 and vr_0 and produces a result in vr_2. This step is crucial since there is very little parallelism if bytecodes are forced to communicate operands to one another strictly through the operand stack.

The **stack renaming unit** keeps track of the next available virtual destination register. No virtual destination register is written twice. This effectively eliminates register WAW and WAR hazards. The stack renaming unit also maintains a stack of virtual registers that are the sources of bytecodes not yet processed. In the $iadd$ example, the **renaming stack** would be popped twice to get the virtual source registers, vr_1 and vr_0. The destination register, vr_2, would be pushed onto the renaming stack.

If the bytecode being read by the control unit may change the flow of control of the program, then the **branch prediction unit** is consulted. If the branch predictor indicates a taken branch, the control unit would begin executing along the new flow of control. We expect that the branch prediction unit would be similar to those employed in modern CPUs. Fortunately, this branch predictor is more constrained than those implemented in hardware for CPUs. This means that the bytecode branch predictor can potentially be much more aggressive, and could even be a background thread [6] that continuously tracks the bytecode program's control flow. As in ILP processors, branch mispredictions require squashing mis-speculated work. The control unit tracks the program order of bytecodes, and commits or squashes their results as appropriate once the outcome of preceding branches has been verified. In-order commit also maintains precise exceptions.

The control unit next schedules independent bytecodes for execution by one of the **execution units**. Figure 1 shows three execution units, but there may be more or less. Bytecodes are placed into the execution queues so that load is distributed evenly among execution units. The control unit will probably have to perform all address calculations itself so that it can detect and properly handle dependences carried through memory.

The control unit and execution units are implemented as threads. In order for this JVM architecture to be effective, the implementation hardware must have support for thread-level parallelism. A likely candidate is a simultaneous-multithreading (SMT) architecture. Unlike common superscalar processors which are restricted to issuing the instructions of a single thread, an SMT processor may choose from the instructions of several threads to fill issue slots. When instructions of these threads are independent of one another, i.e., they exhibit TLP, research has shown SMT to be an effective CPU organization [16].

No SMT processors are commercially available at the time, so experiments on this proposed architecture will have to be carried out using a simulator. Fortunately, Compaq has announced that the Alpha 21464, to be introduced in 2003, will be an SMT system [7]. We expect that the JVM-BLP system would also perform well on other multithreaded systems that have the ability to switch among threads with very low overhead.

This SMT-BLP approach has numerous advantages. The proposed JVM architecture can be implemented purely in a high level language. Even if this approach is no less complex than JVM + JIT, it affords speedups when users disable the JIT and for sections of code that the JIT does not or cannot compile and optimize. It is also code that is platform independent. Porting to a new platform involves only recompilation and possibly a port to a new thread interface. No changes to the JVM's internals are necessary, and from a portability standpoint this is an important advantage.

More importantly, the proposed architecture exploits BLP, especially when implemented on an SMT processor. This offers the potential for substantial speedup over conventional, serialized interpreted execution and even over execution using a JIT. The upper bound on speedup is the lesser of the BLP and the thread capacity of the host processor. Observed levels of BLP are sufficiently high, and in the short term, the the upper bound on speedup will most likely be the thread capacity of the host processor and not BLP. This still offers the potential for profound speedups, speedups that will grow as processor thread capacities expand. The goal of this research is to extract as much of this speedup as possible.

Several requirements must be met in order for the JVM-BLP to realize these speedups. The control unit must proceed fast enough so that it is not out paced by the execution units, and the bytecodes being processed must exhibit substantial BLP. Furthermore, the overhead of stack renaming, of identifying independent bytecodes, of assigning bytecodes to threads, and of maintaining the precise exception model must be kept low enough to avoid overwhelming the benefits of parallelized execution. These are areas which will require significant study, and which constitute a substantial portion of our proposed work.

Exposing BLP opens new opportunities for research in compilers and computer architecture. Among the questions raised are:

1. How can the overhead of finding independent bytecodes and assigning them to threads be minimized?
2. How can a JVM-BLP system look past control-flow in order to expose as much BLP as possible, while minimizing the overhead of this branch prediction?
3. How can the JVM maximize BLP past memory references? This involves both looking past memory disambiguation and maximizing the overall JVM-BLP system's memory-hierarchy performance.
4. How does the ability to exploit BLP affect the role of a JIT in a JVM system? Efficient interpreted execution may allow the JIT to focus on doing a better job of optimizing a smaller number of more critical sections of code.
5. What transformations can a Java compiler perform to expose more BLP? For example, if control flow restrictions hamper BLP in other benchmarks, it may be necessary to do loop unrolling or software pipelining on bytecodes to expose parallelism beyond normal control flow boundaries.
6. How can hardware more directly support the ability to exploit BLP?

These questions are beyond the scope of this paper. Our goal here is to describe a general architecture for exploiting BLP, and quantify the intrinsic levels of BLP that Java programs exhibit. The next step—future work—will be to implement an SMT-based JVM that successfully exploits BLP to improve the performance of bytecode programs.

4 Experimental Framework

The previous section described a multi-threaded virtual-machine architecture that exploits BLP to speed up execution of bytecode programs. In order to determine the potential speedup of such an architecture, this paper measures the BLP exhibited by the SPECjvm98 programs [22] and other popular benchmark programs.

4.1 Bytecode-Program Analysis

Our measurements have been obtained by modifying the x86-Linux port of Kaffe. Kaffe is an open source JVM with a modular JIT compiler. Kaffe consists of about 110,000 lines of C code, and has been ported to a dozen or so platforms.

With the JIT disabled, Kaffe implements the JVM using a pure bytecode interpreter. We modified this bytecode interpreter to collect traces of bytecodes as executed by the JVM. We translate these bytecodes to an intermediate form similar to RTLs [2] that we call JRTLs. In JRTLs, stack-relative operands are mapped onto virtual registers, and memory addresses are fully resolved. The mapping from stack locations to virtual registers is accomplished in much the same fashion as described in Section 3. This register-renaming scheme is typical of code generation for stack-based intermediate languages.

The user is able to choose a method or set of methods for which trace statistics should be gathered. When one of these methods is encountered, trace data is captured until the method completes. Trace data is captured for all methods called by one of the specified methods, including recursive calls to itself.

As bytecodes are captured, the JRTLs to which they are mapped are placed into a 1 K-entry buffer. When the buffer is filled, or the capture conditions are no longer satisfied, the JRTLs stored in the buffer are analyzed for BLP.

Our system collects statistics on a per thread basis. Each thread of execution has its own BLP-statistics context that includes cumulative BLP, number of bytecodes issued, stack renaming information, and a JRTL buffer. When a thread terminates, the information stored in this context is written to a disk file. Note that only one of the benchmarks that we studied in SPECjvm98, **raytrace**, was explicitly multithreaded.

Calculating BLP is done by simulating instruction issue. Starting from the oldest instruction in the buffer, each instruction is considered for issue if it is not RAW dependent on prior instructions. Once the end of the buffer is reached, all instructions marked ready have been determined to be independent of one another. These instructions are issued and removed from the buffer. This process is repeated until the buffer is empty. BLP is calculated as the average number of bytecodes issued per pass through the buffer (per "cycle"). Our metric for BLP is *BPC*: bytecodes issued per cycle.

4.2 Benchmarks

Choosing a set of benchmark programs is always a difficult task. While there are a great many GUI java applications and applets, we chose not include any in our study of BLP. Our intuition was that the performance of these programs would be most affected by user reaction time, and not the JVM implementation. Instead, we chose to concentrate on non-GUI Java applications such as compilers, parser generators, renderers and audio format decoding.

Our benchmark set consists of 8 of the 9 SPECjvm98 benchmark applications and the Java port of Linpack. The SPECjvm98 programs that we studied were:

– **check** - A simple JVM validation tool
– **compress** - A java port of the ubiquitous Unix compress utility.
– **jess** - An expert system shell.
– **raytrace** - An off-line raytracer.
– **db** - A Java database manager.
– **javac** - The Java language compiler.
– **mpegaudio** - A Java MP3 decoder.
– **jack** - A parser generator.

Gathering BLP data proved to be quite expensive in terms of CPU time required. As a result we elected to terminate simulations before program completion and report BLP among the first several hundred million bytecodes executed. One of the SPECjvm98 benchmarks, **mtrt**, was omitted altogether due to time constraints.

5 Results

Our measure of BLP assumes that no branches are predicted correctly. This only measures BLP within individual basic blocks. While it provides an interesting view of the Java programs' characteristics, it is a pessimistic indicator of how much BLP can actually be extracted from Java programs. It would be quite easy for a JVM-BLP system to implement accurate branch prediction. Today's microprocessors implement hardware branch predictors that achieve prediction accuracies of 90–97% [17] for the SPECint95 benchmarks [20], and these hardware-based techniques use simple table-driven approaches that can be easily implemented in a software JVM. The JVM-BLP might even augment these techniques with algorithms that take advantage of the flexibility of a software execution environment.

The numbers we report are therefore low compared to the levels of BLP that can be realized with a JVM-BLP system that employs software branch prediction.

Table 1 presents our results. Both *check* and *linpack* were run to completion. From these numbers, we conclude that there is a significant amount of BLP present even in non-numeric control bound codes such as compilers (*javac*) and parser generators (*jack*). Even higher levels of BLP are found in loop bound codes like *compress, mpegaudio,* and *linpack.*

6 Conclusions and Future Work

In this paper we have introduced the notion of *bytecode-level parallelism* (BLP)—independent Java bytecodes that can be executed in parallel. Given sufficient amounts of BLP in Java programs, an appropriately implemented Java

Benchmark	Bytecodes Simulated	BLP
check	621,361	4.171
compress	201,288,705	8.599
jess	279,102,465	3.765
raytrace	222,370,817	3.497
db	464,251,905	5.518
javac	150,559,745	5.015
mpegaudio	114,443,265	11.883
jack	209,347,585	3.901
linpack	9,421,825	14.564

Table 1. Results. Bytecodes simulated denotes the number of bytecodes run before the simulation was terminated. BLP numbers are based entirely on these bytecodes.

Virtual Machine (JVM) may find profit in parallel bytecode execution. We have proposed one such JVM and undertaken a study of BLP to justify further investigation. Our experiments show average BLP levels of 6.768 among the benchmarks chosen, and that BLP levels are as high as 14.564 for one of them.

The observation that Java programs exhibit substantial levels of BLP suggests that a parallelized Java execution environment can realize substantial speedups. Microprocessor designers already employ a wealth of effective techniques to expose parallelism among machine-level instructions and to exploit this to speed up program execution. This proposal describes a JVM architecture—*JVM-BLP*—that takes advantage of the simultaneous multithreading to appear in the Alpha 21464 to employ ILP-style techniques in the JVM.

Speeding up a JVM interpreter allows users to realize substantially better Java performance, without the need for a just-in-time compiler (JIT), for code that the JIT cannot optimize, and under circumstances in which the JIT must be disabled. The interaction between the BLP engine and the JIT is an interesting area for future research, and in fact this new execution paradigm raises a wealth of other interesting avenues for research, some of which we describe in Section 3.

References

1. Ali-Reza Adl-Tabatabai, Michal Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast and effective code generation in a just-in-time Java compiler. *ACM SIGPLAN Notices*, 33(5):280–290, May 1998.
2. Manuel E. Benitez and Jack W. Davidson. The advantages of machine-dependent global optimization. In Jürg Gutknecht, editor, *Programming Languages and System Architectures*, volume 782 of *Lecture Notes in Computer Science*, pages 105–124. Springer Verlag, March 1994.
3. Nick Benton and Andrew Kennedy. Interlanguage working without tears: Blending SML with Java. *ACM SIGPLAN Notices*, 34(9):126–137, September 1999. Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '99).
4. Nick Benton, Andrew Kennedy, and George Russell. Compiling Standard ML to Java bytecodes. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1) of *ACM SIGPLAN Notices*, pages 129–140. ACM, June 1999.
5. Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V.C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeño dynamic optimizing compiler for Java. In *Proceedings of the ACM 1999 conference on Java Grande*, pages 129–141, New York, 1999. ACM Press.
6. R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt. Simultaneous subordinate microthreading (SSMT). In *Proc. of the 26th International Symposium on Computer Architecture*, pages 186–95, May 1999.
7. K. Diefendorff. Compaq chooses smt for alpha. *Microprocessor Report*, pages 1, 6–11, Dec. 6 1999.
8. M. Anton Ertl. Stack caching for interpreters. *ACM SIGPLAN Notices*, 30(6):315–327, June 1995.
9. Allan Heydon and Marc Najork. Performance limitations of the Java core libraries. In *Proceedings of the ACM 1999 conference on Java Grande*, pages 35–41, New York, 1999. ACM Press.
10. C.-H. A. Hsieh, J. C. Gyllenhaal, and W. W. Hwu. Java bytecode to native code translation: the Caffeine prototype and preliminary results. In IEEE, editor, *Proceedings of the 29th annual IEEE/ACM International Symposium on Microarchitecture, December 2–4, 1996, Paris, France*, pages ??–??, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1996. IEEE Computer Society Press.
11. JPython. see http://www.jpython.org.
12. Thomas Kistler and Hannes Marais. WebL — a programming language for the Web. *Computer Networks and ISDN Systems*, 30(1–7):259–270, April 1998.

13. Alexander Klaiber. The technology behind crusoe processors. Whitepaper; see http://www.transmeta.com/crusoe/download/pdf/crusoetechwp.pdf, Jan. 2000.
14. Ioi K. Lam and Brian C. Smith. Jacl: A Tcl implementation in Java. In *Proceedings of the 5th Annual Tcl/Tk Workshop*, pages 31–36, July 14–17 1997.
15. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison Wesley Longman, Inc., second edition, April 1999.
16. Jack L. Lo, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Rebecca L. Stamm, and Dean M. Tullsen. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems*, 15(3):322–354, August 1997.
17. S. McFarling. Combining branch predictors. Tech. Note TN-36, DEC WRL, June 1993.
18. Tia Newhall and Barton P. Miller. Performance measurement of dynamically compiled java executions. In *Proceedings of the ACM 1999 conference on Java Grande*, pages 42–50, New York, 1999. ACM Press.
19. Todd A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In ACM, editor, *Conference record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: San Francisco, California, January 22–25, 1995*, pages 322–332, New York, NY, USA, 1995. ACM Press.
20. Standard Performance Evaluation Corporation. SPEC CPU95 Benchmarks. WWW site: http://www.specbench.org/osg/cpu95, December 1999.
21. Sun Microsystems, Inc. The Java Hotspot performance engine architecture. Whitepaper; see http://java.sun.com/products/hotspot/whitepaper.html, Apr. 1999.
22. The Standard Performance Evaluation Corporation. SPEC JVM98 benchmarks. see http://www.specbench.org/osg/jvm98, December 1999.