BLP: Applying ILP Techniques to Bytecode Execution

Kevin Scott and Kevin Skadron
Dept. of Computer Science
University of Virginia
Charlottesville, VA 22904
{jks6b,skadron}@cs.virginia.edu

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 explores the concept of bytecode-level parallelism *(BLP): data- and control- independent bytecodes that can be executed concurrently, just as conventional machine instructions are executed concurrently to exploit instruction-level parallelism (ILP). Quantifying the degree of BLP available is an important first step in finding ways to exploit it. This paper presents measurements for the SPECjvm98 suite that show levels of BLP are high even within basic blocks—with an average BLP degree of 5.6—and that levels of BLP are very large—19.8—when speculating past control flow. This suggests that techniques designed to exploit BLP can yield substantial speedups in the performance of a JVM and other interpreters.*

1. Introduction

Bytecode representations are portable formats that allow programs–whether small "applets" or large desktop applications–to run on many 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 almost all 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.

One problem is that most bytecode execution environments, like implementations of the Java Virtual Machine (JVM), do not exploit parallelism in bytecode programs. 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.

We have conducted a preliminary study of whether Java bytecode programs exhibit fine-grained, *bytecode-level parallelism* (BLP), analogous to ILP. If a program exhibits BLP, then applying ILP techniques to bytecode execution—in other words, exploiting BLP rather than ILP—would yield substantial benefits.

In the SPECjvm98 programs [23], we find that the average degree of BLP can be as high as 10.9 independent bytecodes and averages 5.6 (see Section 2). These measurements are conservative, only measuring BLP within basic blocks. If the VM is allowed to speculate past control flow to discover additional independent bytecodes, levels of BLP rise to an average level of 19.8 independent bytecodes.

If these independent bytecodes can indeed be executed in parallel, a JVM system that exploits BLP could potentially achieve a speedup factor of 5.6 or—with software branch prediction in the JVM—even 19.8. The key, of course, would be to minimize overhead associated with BLP execution.

Recent research has focused on improving the performance of bytecode interpreters by using JITs [1, 3, 9, 13] or dynamic compilation environments [22]. Unfortunately, despite improvements like *threading* the interpreter's fetch engine [7] (unrelated to program or processor multithreading), current interpreters lacking native code generation capabilities still serialize bytecode execution. This means, for example, that memory-reference delays entirely stall the interpreter and that bytecodes are never executed in parallel. JITs also suffer some drawbacks. They can 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. Finally, JITs are sufficiently large and complex that they may simply be omitted from handheld devices. These difficulties can perhaps be overcome with sufficient effort, but approaches that improve the performance of the byte-

In the Proceedings of the Second Annual Workshop on Hardware Support for Objects and Microarchitectures for Java, Sept. 17, 2000, Austin, TX—held in conjunction with the International Conference on Computer Design 2000.

code interpreter may be a better use of effort, and such approaches may also lead to synergies with advances in JIT technology.

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. Yet speedups are beneficial even when debugging, because some bugs can take a substantial amount of time to manifest. 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. This is another reason to improve the performance of the bytecode interpreter.

Romer *et al.* [18] argue that interpreter performance is chiefly a characteristic of the interpreter itself and not of the interpreted application, but they do not explore concurrent execution of multiple application instructions. While the interpreter's structure is unquestionably important even for a BLP-capable organization, we expect that different programs would attain different benefits depending on the amount of BLP they contain. We first introduced the notion of BLP in [19]. The only other work of which we are aware that considers the idea of BLP or concurrent execution of multiple bytecodes is that of Radhakrishnan, Talla, and John [16], where the authors propose to improve the performance of the SUN picoJava-II processor [12] with in-order, dual-issue bytecode execution, a fill unit, and stack disambiguation. That work does not, however, consider out-of-order bytecode execution, which would naturally exploit a greater degree of BLP.

This paper characterizes the limits of BLP in the SPECjvm98 workload that would be available for extracting speedups; the next section presents these BLP measurements. Our goal here is not to evaluate new JVM architectures, but Section 3 describes one possible way that a JVM might exploit BLP. Section 4 then concludes the paper.

2. BLP Characterization

Measurements have been obtained with a tool—*BAT*—which we have built on top of the x86-Linux port of the Kaffe [10] JVM. Kaffe is an open-source JVM with a modular JIT compiler that can be deactivated as needed. 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. BAT modifies this bytecode interpreter to perform renaming: stack-relative operands are mapped onto virtual registers, and memory addresses are fully resolved. BAT can be configured to measure BLP within or across the basic blocks executed by the JVM. This approach differs from *Jaba*, which Radhakrishnan, Rubio, and John developed and used to characterize the instruction-level characteristics of Java applications [15]. Jaba performs offline analysis of a bytecode trace that has been generated by instrumenting the Sun JDK's JVM. The Jaba system is particularly attractive in that it can characterize not only the bytecode operations from Java programs, but the native processor instructions executed by the JVM.

BAT measures BLP by accumulating renamed and resolved bytecodes in a "BLP buffer" and then scanning the buffer for independent bytecodes. After each pass, the average BLP is updated, "executed" bytecodes are removed from the buffer, and dependences are updated in preparation for the next pass. Our experiments used a BLP buffer of 1024 entries.

Measurements have been obtained for all the SPECjvm98 [23] benchmarks *mtrt*.[1] We were not able to run the programs to completion (except for *check* and *linpack*), because our current algorithm for performing stack renaming never frees any memory allocated for that purpose. As a result, BAT eventually exhausts available memory and terminates. Our results appear in Table 1, and report average BLP for the portion of execution that BAT is able to capture before terminating. We report two types of BLP measurements—BLP within a basic block, and BLP across basic block boundaries with perfect branch prediction.

Benchmark	Bytecodes Simulated	BLP (basic block)	BLP (perfect BP)
check	0.6 M	4.1	11.5
compress	447 M	8.1	11.5
jess	279 M	3.8	27.8
raytrace	222 M	3.5	42.8
db	447 M	5.5	18.3
javac	279 M	4.7	15.6
mpegaudio	253 M	10.9	12.2
jack	209 M	3.9	18.4
Arithmetic mean		5.6	19.8

Table 1. BLP for all the SPECjvm98 benchmarks. BLP is reported as an average across the number of bytecodes executed.

From these numbers, we conclude that there is a significant amount of BLP present even in non-numeric, control-

[1] Due to an unusually small (512MB) data segment size on our simulation host *mtrt* exhausts available memory before meaningful data can be collected. This is unfortunate, since *mtrt* is dual-threaded.

bound codes such as compilers (*javac*) and parser generators (*jack*). Even higher levels of BLP are found in loop-bound codes like *compress* and *mpegaudio*.

The high levels of BLP within basic blocks are especially encouraging, because they do not include BLP that can be exposed by speculating across basic blocks (*e.g.*, via software branch prediction in the JVM). This means that a JVM architecture that exploits BLP should realize substantial speedups even if it entirely omits the branch-prediction unit and only exploits BLP within basic blocks. But hardware dynamic branch prediction in microprocessors has dramatic effects on the degree of ILP exposed in today's superscalar microprocessors [20, 24], and we expect a software branch prediction system in the JVM to have similarly powerful effects on exposing BLP and boosting the speedups realized by a system that exploits BLP. The BLP available when speculating past control flow can be as high as 43 independent instructions (*raytrace*, with an average of 19.8). Even if *raytrace* is removed from consideration because of its unusually large BLP, the average is still an impressive 16.5.

3. JVM-BLP Architecture

The preceding results suggest that Java programs exhibit substantial BLP. 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 briefly describes one way in which a multi-threaded JVM might exploit BLP. The proposed software architecture—JVM-BLP—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) [11]. This is because SMT can execute multiple threads simultaneously, each and every processor clock cycle (see next section for more detail).

3.1. JVM-BLP Architecture

Figure 1 shows a diagram of a possible JVM-BLP architecture. The principal ideas are similar to those pervading computer architecture research. The software required to implement JVM-BLP therefore bears a great deal of similarity to microarchitectural simulators like SimpleScalar [2]—indeed, some software, like the software branch predictor mentioned below, could be taken directly from such simulation toolkits.

The diagram shows a **control unit** that reads bytecodes from the currently-executing method. 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.

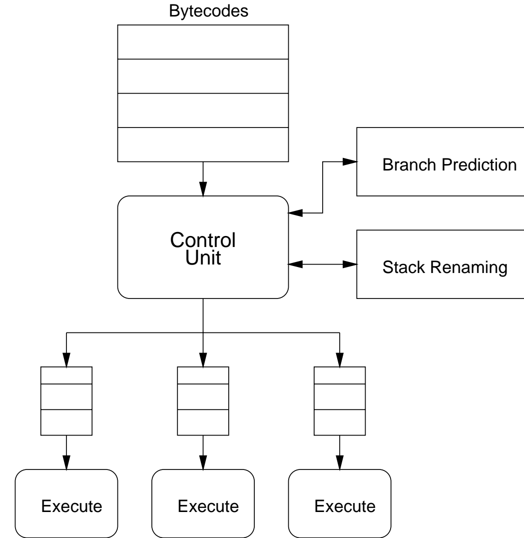

Figure 1. JVM-BLP Organization

The **stack renaming unit** keeps track of the next available virtual destination register. No virtual destination register is written twice. This 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. 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.

It is important to note that JVM-BLP should realize substantial speedups even if it entirely omits the branch-prediction unit and only exploits BLP within basic blocks, because the SPECjvm98 [23] programs exhibit an average

level of BLP of 5.6 just within basic blocks. But speculative execution is likely to yield substantial benefits, especially if organized to ensure the ability to perform speculative method execution to accommodate the frequent method calls in typical Java programs. Together with software branch prediction, this makes the full BLP of 19.8 available for possible speedups.

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 perform address calculations itself in order to detect and properly handle dependences carried through memory. The control unit might also perform bytecode folding [4], in which multiple stack-based instructions are coalesced into a single operation. Finally, note that the control unit must also account for garbage collection; a compacting garbage collector may move objects in memory.

The control unit and execution units are presumably implemented as threads. The control unit should be organized to minimize the JVM state that must be accessed by other threads and guarded with synchronization operations.

3.2. Hardware Support

In order for this JVM architecture to be effective, the implementation hardware must have support for thread-level parallelism. An ideal candidate is a simultaneous-multithreading (SMT) architecture [11]. SMT allows multiple threads to be active at once within the same processor core, reducing the need for context switching among threads and reducing the cost of inter-thread synchronization. The fine-grained nature of SMT's multi-threading allows one thread to use cycles during which other threads are stalled. Furthermore, in every processor cycle SMT issues instructions from multiple threads. This means that not only is each clock cycle used to accomplish work, each issue slot within a clock cycle is used to accomplish work: if any thread has fewer instructions ready to issue than the processor has issue capacity, the remaining issue slots are filled in with instructions from other runnable threads. This means that multiple JVM-BLP execution units can make progress simultaneously.

No SMT processors are commercially available at the time, so experiments on this proposed architecture must be carried out using a simulator. At least two simulators are available: Tullsen's SMTSIM [21] (Tullsen was part of the research group that first proposed SMT), and an SMT version [17] of SimpleScalar's *sim-outorder* simulator [2].

What makes SMT a particularly attractive target for JVM-BLP is the fact that Compaq has announced that the Alpha 21464, to be introduced in 2003, will be an SMT processor [5]. We expect that the JVM-BLP system would also perform well on other multi-threaded systems that have the ability to concurrently execute multiple threads. For example, JVM-BLP can also take advantage of *chip multiprocessing* [14] that Sun's MAJC processor architecture [8] and IBM's Power4 Processor [6] support. Chip multiprocessing runs threads on separate processor cores that share the same die and usually share the same second-level cache. The threads are therefore not as closely coupled as in an SMT processor, and the multi-threading and synchronization overhead rises compared to SMT. Whether SMT, CMP, or some other hardware organization, some form of close coupling is necessary; otherwise, memory coherence will become too expensive.

3.3. Discussion

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 outpaced 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. Because the JVM-BLP structures just described bear so much similarity to the structures used in conventional ILP processors, it would be beneficial to find some way to directly use these structures. Examples include the hardware branch predictor and the native processor registers. These are all areas which will require significant study.

Note that another way to exploit BLP is to use a conventional, superscalar, out-of-order processor, because the out-of-order execution permits speculative execution of bytecodes. A conventional interpreter, however, will execute bytecodes serially, without regard for BLP. New compilation techniques that permit even conventional processors to exploit BLP are another promising avenue of research.

Our goal here has not been to evaluate new JVM architectures. Instead, we wish to disseminate our measurements showing that JVM programs exhibit substantial degrees of BLP, because we feel these results motivate a variety of research. The JVM-BLP architecture just described outlines how a JVM might use multi-threading and an SMT or CMP to exploit BLP, but other techniques for exploiting BLP are certainly possible.

4. Conclusions and Future Work

In summary, this paper shows that Java bytecode programs exhibit substantial degrees of BLP. The average degree of BLP just within basic blocks for the SPECjvm98

benchmarks is 5.6 independent bytecodes, and can be as high as 10.9. These are actually pessimistic measures, as they do not examine BLP beyond control-flow instructions. When executing past basic-block boundaries, BLP levels rise to an average of 19.8, with a highest observed value of 42.8.

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. The extremely high observed levels of BLP suggest large potential benefits can be realized by executing bytecodes concurrently. We briefly outlined one JVM architecture that uses multi-threading to adapt these microprocessor-based techniques to the bytecode-execution engine of a JVM. Speeding up a JVM interpreter allows users to realize substantially better Java performance, without the need for a JIT, for code that the JIT cannot optimize, and under circumstances in which the JIT must be disabled. Note that the proposed JVM architecture can easily be extended to other interpreted environments.

In addition to substantially improving the performance of interpreters, exposing BLP opens new opportunities for research in compilers and computer architecture. In the area of compilers, for example, just as a conventional program can be optimized to expose greater degrees of ILP, a system that exploits BLP creates the need for new bytecode optimizations that exploit greater degrees of BLP. Furthermore, because JITs typically compile only selected segments of a bytecode program, exploiting BLP offers the opportunity to look for synergies between BLP execution and JIT compilation techniques. Finally, in the area of computer architecture, BLP creates the opportunity to apply ILP techniques to native execution of bytecodes, as well as new opportunities to explore microarchitectural support for JVMs.

References

[1] A.-R. Adl-Tabatabai, M. Cierniak, G.-Y. Lueh, V. M. Parikh, and J. M. Stichnoth. Fast and effective code generation in a just-in-time Java compiler. *ACM SIGPLAN Notices*, 33(5):280–290, May 1998.

[2] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. *Computer Architecture News*, 25(3):13–25, June 1997.

[3] M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño dynamic optimizing compiler for Java. In *Proceedings of the ACM 1999 Java Grande Conference*, pages 129–141, New York, 1999.

[4] L.-C. Chang, L.-R. Ton, M.-F. Kao, and C.-P. Chung. Stack operations folding in Java processors. *IEEE Proceedings on Computers and Digital Techniques*, pages 333–40, Sept. 1998.

[5] K. Diefendorff. Compaq chooses SMT for Alpha. *Microprocessor Report*, pages 1, 6–11, Dec. 6 1999.

[6] K. Diefendorff. Power4 focuses on memory bandwidth. *Microprocessor Report*, pages 11–17, Oct. 6 1999.

[7] M. A. Ertl. Stack caching for interpreters. *ACM SIGPLAN Notices*, 30(6):315–327, June 1995.

[8] L. Gwennap. MAJC gives VLIW a new twist. *Microprocessor Report*, Sep. 13 1999.

[9] C.-H. A. Hsieh, J. C. Gyllenhaal, and W. W. Hwu. Java bytecode to native code translation: the Caffeine prototype and preliminary results. In *Proceedings of the 29th annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 1996.

[10] The Kaffe virtual machine. see http://www.kaffe.org.

[11] J. L. Lo, S. J. Eggers, J. S. Emer, H. M. Levy, R. L. Stamm, and D. M. Tullsen. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems*, 15(3):322–354, Aug. 1997.

[12] S. Microsystems. *picoJava-II Programmer's Reference Manual*. Mar. 1999.

[13] T. Newhall and B. P. Miller. Performance measurement of dynamically compiled Java executions. In *Proceedings of the ACM 1999 Java Grande Conference*, pages 42–50, New York, 1999.

[14] K. Olukotun, L. Hammond, and M. Willey. Improving the performance of speculatively parallel applications on the Hydra CMP. In *Proceedings of the 13th International Conference on Supercomputing*, June 1999.

[15] R. Radhakrishnan, J. Rubio, and L. K. John. Characterization of Java applications at bytecode and Ultra-SPARC machine code levels. In *1999 IEEE International Conference on Computer Design*, Oct. 1999.

[16] R. Radhakrishnan, D. Talla, and L. K. John. Allowing for ILP in an embedded Java processor. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 294–305, June 2000.

[17] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 25–36, June 2000.

[18] T. H. Romer et al. The structure and performance of interpreters. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 150–59, Oct. 1996.

[19] K. Scott and K. Skadron. BLP: Applying ILP techniques to bytecode execution. Tech. Report CS-2000-05, University of Virginia Department of Computer Science, Feb. 2000.

[20] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark. Branch prediction, instruction-window size, and cache size: Performance tradeoffs and simulation techniques. *IEEE Transactions on Computers*, 48(11):1260–81, Nov. 1999.

[21] SMTSIM. see http://www-cse.ucsd.edu/users/tullsen/smtsim.html.

[22] Sun Microsystems, Inc. The Java Hotspot performance engine architecture. Whitepaper; see http://java.sun.com/products/hotspot/whitepaper.html, Apr. 1999.

[23] The Standard Performance Evaluation Corporation. SPEC JVM98 benchmarks. see http://www.specbench.org/osg/jvm98, Dec. 1999.

[24] D. W. Wall. Limits of Instruction-Level Parallelism. Technical Report 93.6, Compaq Western Research Laboratory, Nov., 1993.