Top-Level Data-Memory Hierarchy Performance: Java vs C/C++

Milo Martin Manoj Plakal Venkatesh Iyengar
{milo,plakal,venkates}@cs.wisc.edu

December 18, 1996
CS752, UW-Madison
Prof. Mark D. Hill

1 Introduction and Motivation

Memory hierarchies are designed and tuned by computer architects with a general anticipated workload in mind. Most of the current literature in memory hierarchy design and innovation conducts design evaluation using standard benchmarks like the SPEC suite [Cas95] which consist of C and FORTRAN programs. We believe that with the rise of Java [GJJ96] on the horizon, a substantial number of future machines will be running Java workloads. Due to a unique set of language features, Java workloads have the potential to exhibit significantly different pattern of memory accesses. The machines running these workloads could either use so-called "Java chips" that execute Java bytecodes directly or conventional RISC/CISC processors perhaps slightly modified to support efficient execution of bytecodes. Hence, it is beneficial to analyze the memory hierarchy performance of machines running Java workloads and in particular, to propose memory hierarchy configurations that help in efficient execution of these workloads.

Java bytecodes [Gos95] are often used to represent a Java program. These bytecodes are the instruction set of a virtual machine – the Java Virtual Machine (JVM) [Mic95], a stack-based architecture designed for Java. Java bytecodes also conceptually separate the address space for locals and heap data references, and forces all non-scalar values to be placed in the heap and not on local call stack. This leads us to propose that Java workloads have more memory traffic and different memory access patterns and thus different cache behavior when compared to equivalent C/C++ workloads. We compare the memory hierarchy performance of a machine running a Java workload with that of a machine running an equivalent C/C++ workload. This work focuses on the only the top-level of the data-memory hierarchy. Trace-driven cache simulations were used to obtain data on the behavior of the top of the memory hierarchy for various cache configurations, including configurations specifically designed to exploit Java's specific memory model. Our results suggest that due to the differences in memory models of C/C++ and Java, there is a performance increase gained by specifically configuring a memory system for a Java workload.

2 Related Work

Because Java is a new language which is quickly becoming heavily commercialize, few scholarly publications on Java have been published. Some relevant references include

[RLV+96] and [Ert95]. These papers examined the performance of interpreters: [RLV+96] investigated the performance of various interpreters including Java, Perl and Tcl/Tk while [Ert95] discusses methods of optimizing interpreters for stack-based virtual machines. To the best of the our knowledge, there are no papers published that specifically focus on memory behavior of Java bytecode rather than Java interpreters.

Since Java is a young language, Java benchmarks have not yet matured. However some previous work has been performed, mostly by companies and individuals on the Internet. The "benchmarks" available on the Internet at present are either microbenchmarks (tight loops repeating a single kind of operation such as addition, line drawing, etc.) or graphics-based applets. Applets constitute the vast majority of currently available Java code but we do believe that non-applet Java applications will soon become widespread. We also believe that Java applications will be resource intensive programs, unlike many of the simple applets for which Java is currently known for almost exclusively.

3 A Brief Overview of Java

Java is a new programming language developed at Sun Microsystems targeting the Internet as the future means of code distribution. It claims to be a "clean" subset of C++ augmented with useful features that aim at increasing portability and security of code. It does away with many characteristic features of C++ including pointers, structures, unions, operator overloading, multiple inheritance and explicit deallocation of memory. It introduces a garbage-collected heap, run-time bounds-checking, exception-handling and support for multi-threading. It also has support for invoking code written in native code rather than Java bytecode. Below, we elaborate on the execution and memory models of Java and how they differ from those of C++.

3.1 Java's Execution Model

Java programs can be executed in a variety of ways. (See slide 9[1] for a graphical presentation.) Java source can either be compiled directly to native code, as traditional languages are compiled. The norm is for Java to be compiled to bytecodes which form the instruction set of the Java Virtual Machine. These bytecodes, a la Pascal's P-Code, ensure portability across platforms. The virtual machine is a stack-based architecture to allow compact code and easier emulation on architectures with few general-purpose registers.

Java bytecodes can be executed in a variety of ways. The usual mode of execution (currently) is interpretation. The bytecodes can also be compiled to native code by an optimizing compiler (For example Toba [DoCS] converts the bytecodes to C/C++ and then uses `gcc` to compile and optimize the C/C++ code) or they can be (naively) compiled on the fly at run-time by a so-called "Just-In-Time" (JIT) compiler. A more interesting mode of execution is via a Java processor that directly executes Java byte-

[1]Instead of including a large number of figures, throughout this paper we simply refer to the slides from the presentation.

codes as its native instruction set. Sun Microsystems has announced that it has designed one such chip, PicoJava [Mic96], and is planning two other versions as well, MicroJava and UltraJava.

Our study is most applicable to hardware that is direct executing bytecodes, such as a Java processor. It however is a reasonable conjecture that our findings will be at least partially applicable to JIT-compiled Java. If JIT-compiled Java should exhibit similar patterns of memory accesses since the compiler does not perform optimizations and generates code naively (for speed). Our findings might also have relevance to Java code compiled to optimized native code. Investigating and analysis of these other methods of executing Java are out of the scope of this paper.[2]

3.2 Java's Memory Model

Java's memory model is quite different from the conventional memory model used by C/C++. In order to contrast the two memory models, we will briefly describe the C/C++ memory model first before describing the model used in Java.

In C/C++ programs, conceptually all data is stored in single shared address space memory. In load/store architectures, operands are brought into registers for computation and then written back to memory (this holds to some extent for CISC architectures too). Optimizing compilers perform register allocation which places frequently used variables in registers. Parameters and local variables of functions are stored in a procedure call-stack, which exists in the same address space as the heap. In essence a C/C++ programmer uses a memory-memory architecture memory model. For example, in C/C++ the programmer can manipulate the address of a local scalar, preventing the compiler from performing the optimization of placing that value in a register.

The Java memory model is higher level then the model used by C/C++. Pointers and references to non-scalars are not allowed, and this language property is also enforced in the stack based bytecodes. Operands are pushed onto an operand stack prior to computation. Local scalar variables and local references to non-scalar items are stored in a statically allocated space (such as the procedure call stack). All non-scalar objects, including non-scalar locals, are allocated on the heap. This heap exists in a different address space from the locals and is garbage-collected. This conceptual separation is enforce by the Java bytecode since the bytecodes use different instructions to access locals and heap objects. Hence, our memory model has three distinct address spaces: operand stack, locals and the heap.

Slide 14 shows the break-up of the data references for our benchmarks. From the data it is evident that operand stack references are responsible for the majority of the total memory references. Furthermore, many of these accesses are to the few locations at the top of the stack. This locality leads us to suspect that having a separate cache or operand stack register file for stack operands might be useful. If no special support

[2] In the last week of our project, we managed to obtain an optimizing compiler that can convert Java source or bytecodes to native code (via C) [DoCS]. The performance of the generated code easily beat the Java interpreter by a large margin and came respectably close to that of equivalent C++ programs. Due to problems with tracing dynamically linked executables with QPT2, no cache simulations were performed.

is provided for the operand stack, the JVM becomes a memory-memory architecture, which is not desirable.

The local value references also contribute to a significant number of references, especially when considering that overall, one would expect there to less local scalar data than the heap data, and this local reference stream should have a large degree of locality. Due to the different instructions in the JVM used to access the different types locals, heap and stack operands, these three can be cached separately, to exploit the significant locality present in the locals and operand stack reference streams.

4 Methodology

Having described the memory model that Java uses, we now propose techniques to take advantage of the memory model. As stated previously, the large number and high locality of operand stack references justifies investigating a separate cache for stack operands. Furthermore, we might even have separate caches for the local variables and the heap, or we might combine them. The operand stack, locals and heap memories can be managed by three separate caches. We have gathered and analyzed empirical data on these approaches. The most basic option is to use a unified cache for all operand stack, locals and heap references. This is the same as having a memory-memory architecture. This is not a practical option due to the high frequency of stack accesses, of which multiple need to be serviced within the same clock cycle. We want to avoid making the stack a performance bottleneck.

4.1 Benchmarks

Gathering benchmarks for this project was a challenging task. Java is a young language and benchmarking for Java has not matured into a JavaSPEC suite yet. There are few large Java applications available for public use. The vast majority of Java code available today is in the form of applets which are executed from within a WWW browser and which are also highly graphics-intensive and make extensive use of native code libraries. Furthermore, we also want to compare Java with C/C++, so we required that many of our benchmarks to be written in both languages. Due to the lack of high level translation tools, we ported code from C++ to Java and Java to C++ to obtain these benchmarks. Even if translation tools were available, our results would be more of a measure of the quality of the conversion, and not differences in the two languages.

The details of our benchmarks are given in the table below. The Java version of `linpack` was obtained on the WWW and was hand-converted to C++. `jb` is a tool that converts Bison-generated parsers from C to Java. `Jack` is a LL(k)-parser generator developed at Sun Microsystems. With the possible exception of the eight queens program, we consider the following benchmarks to be sufficiently computationally intensive, and somewhat real-world, not to qualify as toy benchmarks. Even with our efforts, the benchmarks used in this study could provide us with results that are non-representative of real Java workloads. However, these benchmarks are sufficient for a survey of this scale.

Benchmark	Description	Java or C++
8q	Solves the 8-queens chess problem	Both
matmult	Multiplies two 50x50 floating-point matrices	Both
cellauto	100-step simulation of a 200-cell 1-D cellular automaton	Both
quicksort	Quicksort 5000 random integers	Both
wordfreq	Count word frequencies in a 7575-character file	Both
linpack	Standard floating-point benchmark	Both
jb	Bison-generated Java parser (converted to Java by a tool)	Both
jack	LL(k)-parser generated from a Java grammar	Java only
javac	Sun's Java compiler	Java only
kawa	Scheme interpreter in Java	Java only

4.2 Obtaining traces and cache simulations

The traces for the C/C++ programs were obtained by using QPT2 [Lar93], a program which instruments an executable to produce a memory reference trace. The memory reference streams from the trace were as used as out input for our cache simulator, DineroIII (QPT2 and DineroIII are part of the Wisconsin Architectural Research Tool Set [WARTS], refer to `http://www.cs.wisc.edu/~larus/warts.html`). To obtain Java traces, we modified Sun Microsystem's Java interpreter (from Sun's Java Development Kit 1.0) to generate a trace of all code/stack/local/heap memory references. This master trace was then processed to yield separate stack, local, heap and various combined cache configurations in Dinero format.

When simulating various cache configurations with Dinero, the main parameters we investigated were cache size, associativity, and block size. All other cache parameters were kept at reasonable default values. Furthermore, associativity and block-size were varied only in initial tests in order to find a set of reasonable values that gave good performance. We then fixed those parameters for subsequent simulations.

The C/C++ executables traced used the SPARC instruction set and were compiled with `gcc`. Our patching of the Java interpreter ignored garbage collection, threads, exception handling and native code. Our benchmarks don't use exceptions or threads, and experiments show that garbage collection was not a significant factor due to the fact it was only invoked minimally by a single one of our benchmarks.

5 Results and Discussion

We first simulated separate caches for the stack, locals and the heap and investigated the effects of varying cache size, associativity and block size. We also looked at the performance obtained when we put these caches together in various configurations. These results are described in the first subsection below. The next subsection compares the performance of these configurations with that of equivalent C/C++ programs on conventional unified data caches.

In a number of our graphs we have depicted an average over all the benchmarks we used. This average is actually the arithmetic mean of the individual miss ratios. We felt this was the most reasonable way of arriving at a combined measure.

5.1 Cache configurations for Java

First we investigated the individual cache performances— the stack, the local and the heap cache separately. This also allowed us find reasonable values for the parameters for each cache to use later in the combined cache configurations.

5.1.1 Stack cache

The stack cache handles all the operand references. Hence it is accessed by almost every instruction, and should be expected to have a large number of references. We investigated cache sizes of 16 bytes to 512 bytes with block sizes of 1 to 4 words, on three associativities, 1, 2 and 4. Our results for the stack cache simulations are shown in slides 17, 18, and 19.

Slide 17 plots miss ratio against cache size for each of the benchmarks assuming a 16 byte block size. From this we can see that the miss ratio drops swiftly with increase in cache size. A 128 byte cache size is enough to keep the miss rate to within about 1%. This suggests that a small cache would be enough to cater for most of the stack misses in spite of the large number of references.

In slide 18 we see the effect of various associativities on the miss rate for cache configurations with 16 byte block sizes. On the whole, it is pretty obvious that direct mapping isn't much worse than the others, and will do for the stack cache. These values are averaged over the set of benchmarks.

Slide 19 shows the effect of different block sizes on the miss ratio for a cache with direct mapping. These values are averages for the set of benchmarks. In general, the 32 byte blocks do not perform well, possibly due to capacity misses in the small cache. Of the rest, the best choice would be the 16 byte block size.

5.1.2 Locals cache

Locals refer to all the local variables which are not allocated on the heap. For the local cache we investigated cache sizes of 64 bytes to 2K, block sizes of 1 to 4 words, and 1, 2 and 4 way associativity. The results for the locals cache simulations are shown in slide 20 and 21.

Simulations in slide 20 indicate that the miss ratio for the local cache also falls steeply and a cache of about 1K can reduce the cache misses to within 2%. This simulation assumes 16 byte blocks and direct mapping.

The graph in slide 21 shows the effect of block size on the miss ratio for a local cache with direct mapping. Similar to the stack cache, the 16 byte cache tends to outperform the other sizes. The results here are averaged over all the benchmarks.

5.1.3 Heap cache

The heap cache gets all the allocated references and hence would require to be reasonably larger than the other two caches. We carried out simulations for cache sizes of 1K to

32K, with associativities of 1, 2 and 4 way. The results for the heap cache simulations are shown in slides 22 and 23.

From the graph in slide 22, we can see that the miss ratio for the heap does not drop as much as the stack or the local cache, with increase in the cache size. This is understandable due to the lack of locality in the heap. However, the miss rate tends to remain somewhat constant after a size of about 8K. This graph shows the results for all the benchmarks.

In slide 23, observing the performance on different associativities, we see a reasonable improvement from direct mapping to 2–way associativity. However, there isn't much of a difference between 2–way and 4–way mapping especially at larger cache sizes. This suggests that a 2–way mapping might be adequate.

5.1.4 Putting it all together

Then, we decided to combine the separate caches we considered above and went on to compare three different configurations:

- A separate stack cache with locals and heap combined in a single cache
- the stack, the locals, and the heap in three different caches, with a small local cache.
- the stack, the locals, and the heap in three different caches, with a large local cache.

We fixed our stack cache size to 128 bytes with 16-byte blocks and direct mapping, our local cache for 16 byte blocks with direct mapping and our heap cache for 2-way set-associativity with 16-byte blocks. For one simulation we used 512 byte local cache size and for the other we used 1K local cache size. Results show that in general, the three cache configuration performs better than the two cache configuration though they do have similar trends for larger cache sizes. However the larger local cache always performs slightly better then the smaller cache size though not by much. Increasing the stack cache size to 512 bytes shows a marginal improvement. The results are shown in slides 24 and 25.

5.2 Comparing C/C++ with Java

Now that we have investigated the cache configurations that are suitable for Java, we now compare the performance of some of these suitable cache configurations with that of C/C++ programs running on a system with a unified data cache.

Slide 26 shows the comparative miss ratios of C/C++ vs. Java for our benchmarks. It includes miss ratios for the C/C++ programs and Java programs with three separate caches as well as the two cache configuration. Our results show that C/C++ and Java have similar miss ratios. However, we have not yet presented the effect of all the factors.

Slide 27 shows the heap references alone, the locals and the heap, and the total data references as well as the C/C++ references. The graph values has been normalized with respect to the C/C++ references. The results show that Java has **significantly many**

more data references than C/C++. Comparing miss ratios is meaningful only when we are comparing programs with approximately equal references. Hence, if we want to make any meaningful comparisons between C/C++ and Java, we will need to take the total number of misses into account and not just miss ratio. The total number of misses gives us a better picture of the time taken on misses and execution time is actually the ultimate measure.

Slide 28 combines the previous two graphs to show the relative number of misses of Java programs compared to C/C++ programs. This was done by multiplying the average miss ratios in slide 26 with the average number of data references made (from slide 27). We can conclude that Java actually has several more misses than does C/C++. Since what really counts in the end is the execution time, this shows that it is not enough to simply have split caches for stack, locals and heap. Though we may reduce the miss ratio using such methods, the relatively huge number of references made by Java translates into many more misses even for such low miss ratios.

6 Future Work and Conclusion

Other interesting extensions include tracing of garbage collection, which has been shown to have a significant effect on caches [WLM92], examining threading, which leads to less locality. In addition, insight might be gained through investigating the memory system performance of recently announced Java compilers like Toba [DoCS] which promise to provide an efficient platform for executing Java programs.

Since Java has a different memory model than C/C++, this grants an opportunity to exploit this difference to see if there is a particular memory hierarchy which suited it. We have conducted trace-driven cache simulations at the byte-code level for a number of Java benchmarks and the results allow us to conclude:

- Having a separate small stack cache is advisable since it handles a vast percentage of the references, and has a low miss rate.

- Having split caches for the three different types of data references is also beneficial. Since the local and stack caches are relatively small in size, the extra cost is more than justified by the improvement obtained in the miss rate. Furthermore split caches provide us with extra read/write ports and the possibility of overlapping hits in the three caches.

- In particular, it seems reasonable to use a direct-mapped stack cache with 16-byte blocks and size of 128, 256 or 512 bytes; a direct-mapped 1 Kbyte locals cache with 16-byte blocks and a 2-way set-associative heap cache of whatever size is desired (4K/8K/16K). These parameters would probably ensure low miss ratios for workloads similar to the benchmarks we used.

These cache configurations are beneficial for Java programs but are not sufficient to speed up Java when compared to C/C++, where the larger number of data references made by Java translates into many more misses even for low miss ratios. Hence, more aggressive memory hierarchies (which might include non-blocking caches, victim caches and stream buffers) need to be proposed for Java processors if we want C/C++-comparable performance from Java.

References

[Cas95] B. Case. SPEC95 Retires SPEC92. *Microprocessor Report*, 9(11):11–14, August 1995.

[DoCS] University of Arizona at Tucson Department of Computer Science. The sumatra project. `http://www.cs.arizona.edu/sumatra/toba`.

[Ert95] M. Anton Ertl. Stack Caching of Interpreters. In *Proceedings of the ACM SIGPLAN 8th Conference on Programming Language Design and Implementation*, pages 315–327, La Jolla, CA, June 1995.

[GJJ96] James Gosling, William Joy, and Guy L. Steele Jr. *The Java Programming Language*. The Java Series. Addison-Wesley, 1996.

[Gos95] James Gosling. Java Intermediate Bytecodes. In *Proceedings of the ACM SIGPLAN Workshop on Intermediate Representations (IR '95)*, pages 111–118, San Francisco, CA, January 1995.

[Lar93] James R. Larus. Efficient Program Tracing. *IEEE Computer*, 26(5):52–61, May 1993.

[Mic95] Sun Microsystems. *The Java Virtual Machine Specification*. Sun Microsystems, Mountain View, CA, 1995.

[Mic96] Sun Microsystems. *picoJava I Microprocessor Core Architecture*. Sun Microsystems, Mountain View, CA, 1996.

[RLV+96] Theodore H. Romer, Dennis Lee, Geoffrey M. Voelker, Alec Wolamn, Wayne A. Wong, Jean-Loup Baer, Brian N. Bershad, and Henry M. Levy. The Structure and Performance of Interpreters. In *Proceedings of the ACM 7th International Conference on Architectural Support for Programming Languages and Systems (ASPLOS VII)*, pages 150–159, Cambridge, MA, October 1996.

[WLM92] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Caching Considerations for Generational Garbage Collection. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming, San Francisco, California*, pages 34–42, June 1992.