RC22296 (W0201-041) January 4, 2002
Computer Science

IBM Research Report

A Study of Memory Behavior of Java Workloads

Yefim Shuf[*], Mauricio J. Serrano[+], Manish Gupta[*], Jaswinder Pal Singh[♦]

[*]IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

[+]Intel Microprocessor Research Labs
mauricio.j.serrano@intel.com

[♦]Princeton University

Research Division
Almaden - Austin - Beijing - Delhi - Haifa - India - T. J. Watson - Tokyo - Zurich

Chapter 1

A STUDY OF MEMORY BEHAVIOR OF JAVA WORKLOADS

Yefim Shuf*
IBM T. J. Watson Research Center
yefim@us.ibm.com

Mauricio J. Serrano†
Intel Microprocessor Research Labs
mauricio.j.serrano@intel.com

Manish Gupta
IBM T. J. Watson Research Center
mgupta@us.ibm.com

Jaswinder Pal Singh
Princeton University
jps@cs.princeton.edu

Abstract This paper studies the memory behavior of important Java workloads used in benchmarking Java Virtual Machines (JVMs), based on instrumentation of both application and library code in a state-of-the-art JVM, and provides structured information about these workloads to help guide systems' design. We begin by characterizing the inherent memory behavior of the benchmarks, such as information on the breakup of heap accesses among different categories and on the hotness of references to fields and methods. We then provide detailed information about misses in the data TLB and caches, including the distribution of misses over different kinds of accesses and over different methods. In

*Also affiliated with Princeton University; yshuf@cs.princeton.edu
†Work done at IBM T. J. Watson Research Center

the process, we make interesting discoveries about TLB behavior and limitations of data prefetching schemes discussed in the literature in dealing with pointer-intensive Java codes. Throughout this paper, we develop a set of recommendations to computer architects and compiler writers on how to optimize computer systems and system software to run Java programs more efficiently. This paper also makes the first attempt to compare the characteristics of SPECjvm98 to those of a server-oriented benchmark, pBOB, and explain why the current set of SPECjvm98 benchmarks may not be adequate for a comprehensive and objective evaluation of JVMs and just-in-time (JIT) compilers.

We discover that the fraction of accesses to array elements is quite significant and demonstrate that the number of "hot spots" in the benchmarks is small. We also show that even a fairly large L2 data cache is not effective for many Java benchmarks. We observe that instructions used to prefetch data into the L2 data cache are often squashed because of high TLB miss rates and because the TLB does not usually have the translation information needed to prefetch the data into the L2 data cache. We also find that co-allocation of frequently used method tables can reduce the number of TLB misses and lower the cost of accessing type information block entries in virtual method calls and runtime type checking.

Keywords: Java, Workload characterization, Memory systems, SPECjvm98.

1. Introduction

The Java Programming Language [13] is gaining popularity as a language of choice for developing applications on a variety of platforms, ranging from servers to embedded systems. At the same time, the growing disparity between processor and memory speeds makes it important to study and optimize the memory behavior of programs. The goal of this paper is to understand the memory behavior of important Java workloads used in benchmarking JVMs and JIT compilers and to provide computer architects and the implementors of JVM components with structured information about the Java workloads (at different levels of detail), which may be useful in formulating their designs.

We first study high-level characteristics of these Java workloads, which are independent of the hardware configuration. We then obtain, based on simulation, a series of increasingly detailed data on the behavior of these programs with respect to the data TLB and caches. We also correlate observed memory behavior and performance problems, such as TLB and L2 cache misses, with the sources of the behavior. Our analysis provides important insights into understanding the key sources of performance loss in Java programs. Throughout the paper, we develop a set of recommendations to computer architects and JVM developers on

techniques to run Java programs more efficiently, and comment on some of the performance improvement techniques discussed in the literature.

2. Experimental Setup

In this section, we describe our experimental setup. We describe the JVM, including the run-time compiler, that was used to run the programs. We describe our methodology for collecting traces and running simulations. Finally, we describe the benchmarks targeted in this study.

A JVM with a Run-Time Compiler. We have performed our experiments under a state-of-the-art JVM, the Jalapeño JVM [1] [7], supported by the Quicksilver *quasi-static* compilation system [31]. Quicksilver performs the compilation of Java bytecodes into quasi-static images (which contain persistent forms of executable machine code and auxiliary information such as exception tables and garbage collection maps) in a separate, ahead of time run. During the actual execution of an application, it tries to reuse existing quasi-static images after performing validation checks and adapting them to the new execution context, to reduce the cost of run-time compilation. Since relatively low compilation overhead is incurred at run-time, our data reflect more closely the intrinsic characteristics of running applications.

Instrumentation and trace generation. In order to generate the trace of heap references, we have extended the run-time optimizing compiler to instrument code. (Since the Jalapeño JVM is itself written in Java, the JVM runtime code is also optimized and instrumented by the same compiler [2].) The instrumentation is performed after the machine-independent optimizations and immediately before register allocation.[1] All load and store instructions referencing object fields, arrays, and virtual method tables (all of which reside on a heap) are instrumented. References to the stack frames (which tend to have good locality) are not instrumented. The instrumentation is done during the *write* phase of the Quicksilver compiler [31]. In the write phase, a program is compiled as it runs. During compilation, the compiler performs the instrumentation steps and generates instrumented quasi-static images.

The data is collected in the *read* phase [31], which corresponds to a "production run" of the program. On each heap reference, a call is made

[1] Both the original and the instrumented versions of benchmarks have the same patterns of accesses to their heap-allocated data.

to the trace generator with the relevant information about a memory reference.

Trace processing and simulation. We process the collected traces off-line. During this step, we perform the simulation of the data TLB and L1 data cache for a Power PC 604e-like [16] configuration (which is similar to the configuration of other modern processors in the market). In addition, we simulate a fairly large (but simple and fast) L2 data cache. The parameters of the simulated memory subsystem are as follows:

- data TLB: 128-entry, 2-way set associative LRU, 4k pages.
- L1 data cache: 32KB, 4-way set associative LRU, 32-byte lines.
- L2 data cache: 4MB, direct-mapped, 32-byte lines.

This choice of parameters allowed us to verify the representativeness of collected traces and validate our simulation results with help of performance monitor counters on a system with a Power PC 604e processor.

We use virtual addresses in the simulation and assume that the operating system will employ one of the standard page mapping techniques to reduce the number of conflicts between pages. We used a heap size of 1 GB, which leads to relatively few garbage collections while executing these applications, and is consistent with the choice of large heap sizes in production environments. Hence, our measurements largely reflect the inherent memory behavior of these benchmarks with sufficient memory.

Benchmarks. We used the industry standard SPECjvm98 [33] benchmarks and the portable business object benchmark (pBOB) [5] to conduct the study. We chose to run the SPECjvm98 benchmarks with the largest data set size (set to 100 [33]), because we found that both size 1 and size 10 are not adequate for collecting meaningful data about the behavior of these applications. We ran each benchmark with the largest data size and simulated 2GB worth of compressed traces (for each benchmark) generated from the point where the benchmark starts the timing. This portion of the trace is representative of the studied workloads and excludes the overhead associated with initializing the JVM and the benchmark harness, and the initial run of a garbage collector. SPECjvm98 benchmarks run for 20-85 seconds on a 166MHz PowerPC system. We simulate 20-50% of actual program execution (depending on the benchmark) and do capture programs' behavior in the steady state.

3. Inherent Heap Access Behavior

In this section, we present the data on the characterization of heap accesses in terms of their distribution among different kinds of accesses and the "hotness" (i.e. the frequency) of various kinds of references.

3.1 Classification of heap accesses

We classify all heap accesses into three basic kinds: accesses to method table entries (which hold the starting address of virtual methods), object fields, or array elements. Fig. 1 shows that a majority of the accesses are to object fields. A surprising result is that for several benchmarks in the SPECjvm98 suite, the fraction of accesses to array elements is quite significant (more than 40% for _209_db and _228_jack). The smallest fraction of accesses are to the method tables, which are due to virtual method calls in the code. It is important to note that we have instrumented the optimized code, where the optimizing compiler has already inlined virtual methods. The compiler currently inlines virtual methods when it can identify a single target of the virtual call based on `final` declarations in Java code or a simple, local type analysis [2], and where inlining is considered to be beneficial for performance. Interestingly, only _213_javac has a substantial number of virtual method calls that could not be resolved at compile time. Most of the virtual methods have been resolved and inlined at compile time for the simpler benchmarks. Hence, the SPECjvm98 benchmark suite contains relatively few applications that are both sufficiently complex and written in a true object-oriented style, using polymorphism. In particular, _201_compress (which seems to be a port of a C benchmark from SPEC95) and _209_db (which has not been derived from an actual application) contain very few virtual method calls, and may therefore be questionable as benchmarks for an object-oriented language. Therefore, we believe that more truly object-oriented benchmarks should be added to the next version of SPECjvm to make the suite more useful for evaluating the ability of JVMs and architectures to efficiently handle object-oriented features.

Interestingly, the characteristics of pBOB are about the average of the characteristics of SPECjvm98 applications. In addition, although pBOB models a database-like workload, it is more object-oriented than _209_db. It has relatively fewer array accesses and many more virtual method calls. Furthermore, pBOB seems to be a more balanced benchmark than _209_db in terms of the variety of operations performed on the database records. Most of the time in _209_db is spent in a sorting routine which sorts an array of references to records (i.e. objects).

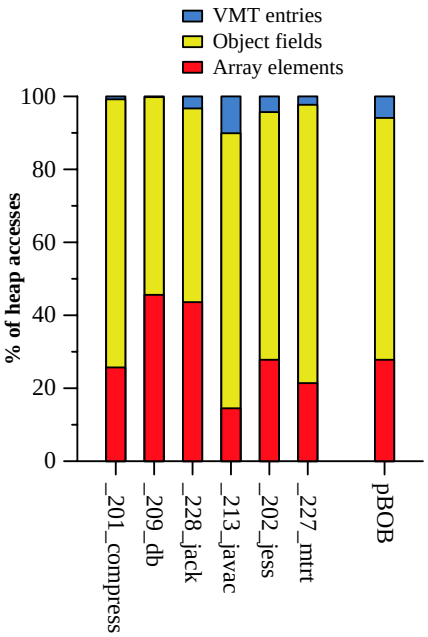

Figure 1: Accesses to VMT entries, object fields, and array elements as % of heap accesses

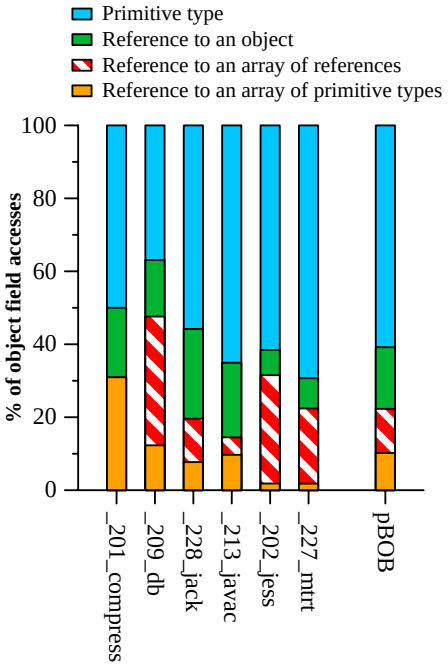

Figure 2: Characteristics of object field accesses

3.2 Classification of accesses to object fields

We classify all accesses to object fields (which is one of the three regions in Fig. 1) into four basic kinds: (i) accesses to fields containing an entity of a primitive type (i.e. short, int, boolean, etc.), (ii) a reference to an object, (iii) a reference to an array of primitive types, or (iv) a reference to an array of references (Fig. 2). The applications that access object fields of primitive types more often are likely to have a better data TLB, data cache, and virtual memory behavior than other applications that access data through one or more references (i.e. *pointer chasing*).

About 30% to 65% of the field accesses are to fields containing references. These measurements suggest that _209_db, in which 65% of fields accesses are to reference fields, is likely to put considerable strain on the memory subsystem (we shall investigate this prediction further in Section 4). Surprisingly, except for two programs (_213_javac and _228_jack), we observe more references to fields containing references to arrays than to fields containing references to objects. For most programs (other than _201_compress and _213_javac), most of the references to ar-

rays are to arrays of references rather than arrays of primitives. This suggests that arrays are used to keep track of large quantities of data and that the data is encapsulated into objects.

Some systems implications of these observations are as follows. First, since pointer chasing is a real issue with these Java benchmarks[2], the compiler writer should consider object inlining as an important optimization that could have a positive impact on the memory performance of these benchmarks. Object inlining in the context of C++ has been studied in [12]. Second, because many accesses are to arrays, and possibly more predictable, prefetching may also be beneficial. We consider the opportunities for prefetching elsewhere in this paper and in our technical report [32]. On the hardware side, a load instruction does not know whether the data loaded from memory contains a reference (a pointer) or some value. If such a functionality were available (e.g. if load instructions could be tagged appropriately), the hardware could, for example, give a higher priority to load instructions retrieving references as opposed to load instructions retrieving values.

3.3 Hotness of fields and virtual methods

In this section, we verify the traditional hypothesis that small sections (so called hot-spots) of a program are typically responsible for most of the execution time and performance problems. This has implications for the effectiveness of run-time compilation strategies employed by compilers like the Sun Hotspot compiler [14] and the IBM Jalapeño adaptive optimization system [2].

We calculated the frequency of accesses to object fields and sorted the frequencies in descending order. We then set an *importance threshold* to a small value (0.1%) and discarded all the fields whose contribution to the total number of field references was smaller than the importance threshold. We then calculated the number of fields that remained standing and computed their cumulative contribution to field accesses (Fig. 3). We obtained similar information (with the same threshold) for the references to virtual method tables (Fig. 4).

For some of the benchmarks (_201_compress and _209_db), very few fields are responsible for a large fraction of field references, and very few

[2]Java, unlike C++, does not have nested objects. A "has-a" relationship between a container object and a contained object is implemented as a reference to the contained object. As a result, Java programs have high overhead associated with dereferencing pointers.

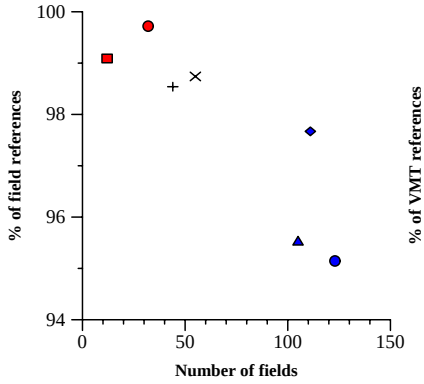

Figure 3: A few fields are responsible for a large fraction of field references (most frequently referenced fields, i.e. Hot Fields)

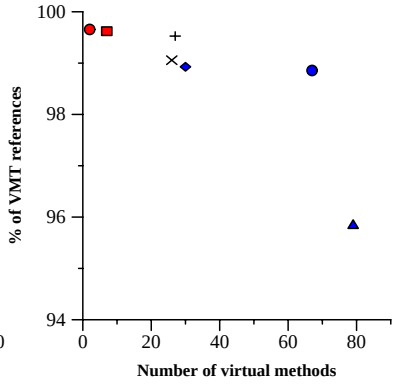

Figure 4: A few virtual methods are responsible for a large fraction of virtual method calls (i.e. Hot Virtual Methods)

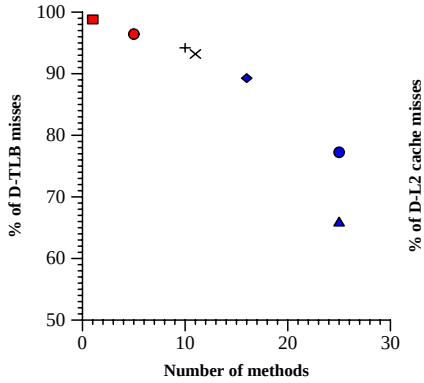

Figure 5: A few methods are responsible for a large fraction of D-TLB misses

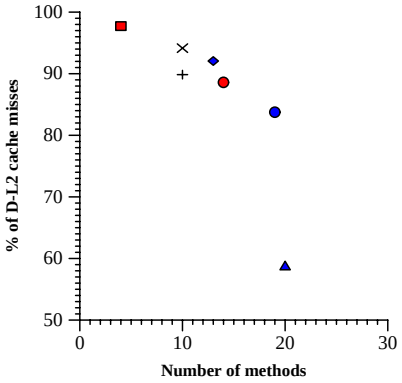

Figure 6: A few methods are responsible for a large fraction of D-L2 cache misses

methods account for a large fraction of all references to method table entries.[3]

On the other hand, the number of important (hot) fields and methods in pBOB and _213_javac is substantially larger. The smaller the number of hot methods and fields, the narrower the scope of optimizations is

[3] A field is associated with a class in which it is declared and is counted as one field for all instances of that class and its subclasses.

and the easier it is to do certain optimizations, especially those that are applied during run-time compilation. For example, if profiling is used to find contemporaneously accessed fields to guide field reordering, then only a small number of hot fields need to be monitored.

The number of hot fields and methods is likely to be a good indicator of how complex or rich a program is. Fig. 3 and Fig. 4 demonstrate a significant disparity in the inherent complexity of the benchmarks. When evaluating the performance of a JVM, perhaps the more complex benchmarks should get a higher weight than the kernels, so that the JVMs that perform better on these benchmarks get a higher score.

4. Memory System Interactions

In this section we discuss the simulation results on cache and TLB behavior of the benchmarks. We will compare some of the collected data with the data gathered by other researchers,[4] and point out some of the surprising results. We shall also discuss the implications of the obtained results and suggest opportunities for optimizations. Note that the reported miss rates are only for heap accesses. Most of the misses during program execution are due to heap accesses since stack frames are usually small ($\approx$ 16K) and stack accesses tend to have good locality. We also found that both I-cache and I-TLB had little impact on the performance of the benchmarks under consideration (with miss rates well below 0.1% for most of the benchmarks)[5].

4.1 Data TLB

Fig. 7 shows the percentage of references that result in misses in data TLB. It comes as no surprise that _201_compress has a good TLB behavior. Its simple structure and little pointer chasing make it TLB friendly. At the same time, _209_db really pushes the memory subsystem to the limit and thrashes the TLB. Its high TLB miss rate seems to be due to a shell sort algorithm that ignores the principles of locality. Overall, the TLB miss rate of other Java programs in this suite is about 2%, which is much higher than 0.1% reported for SPEC95 benchmarks and desktop workloads (written in C and C++) [19].

[4] We have tried to be careful in making the comparisons as objective as possible, by choosing those results in the literature that apply to a memory subsystem with parameters closely resembling those used by us. However, we note that due to architectural differences, and inevitably, due to some differences in the parameters of the memory subsystem, these comparisons should be taken as rough indicators of trends, rather than being taken literally.

[5] Instruction cache miss rates for _228_jack and _213_javac where slightly higher: 0.5% and 0.7%, respectively.

Cvetanovic and Bhandarkar [11] observe that commercial workloads tend to have high TLB misses, but point out that a number of SPECfp applications do have a similar behavior. Although they report the percentage of time spent in the PAL (Privileged Architecture Library) code that performs other functions besides handling of TLB misses (9%-17%), they do not mention the actual TLB miss rates.[6] For most technical workloads, the percentage of time spent in the PAL code is below 1%, but ranges from 5% to 13% for some.

4.2 L1 data cache

Fig. 8 shows the percentage of references that result in misses in the L1 data cache. Again, _209_db has an unusually high miss rate that is likely to be due to pointer chasing.[7] The higher miss rate of _227_mtrt can be explained by a large number of accesses to data through references and by low data locality. Overall, the L1 data cache miss rate of these benchmarks is around 4%, which is higher than 1% for SPEC95 benchmarks and desktop workloads (written in C and C++) [19].

4.3 L2 data cache

Fig. 9 shows the percentage of references that result in misses in the L2 data cache (note that these references also result in misses in the L1 data cache). These misses cost more than a hundred cycles on many processors, and are becoming increasingly costlier with the growing disparity between processor and memory speeds. It seems that _201_compress has a very small working set. Again, a large working set of _209_db and its poor locality result in a very high miss rate. The miss rate of other benchmarks is around 1.5%. This is a fairly high miss rate for the cache size we are simulating.

From the data presented in Fig. 8 and Fig. 10 of [11], we estimate the miss rate to the L2 cache for commercial applications (TPC-A and disk-to-disk sort) at around 5% and for technical/scientific applications at less than 0.2%.

The L2 cache miss rates of Java workloads are very close to the miss rates of commercial applications reported in [22] (2.7% for TPC-A and 1.2% for TPC-C) and [4] (2.7% for TPC-B).

[6] Because our metrics and memory subsystem are different from those used by Cvetanovic and Bhandarkar, it is not meaningful to compare our data to the data reported in [11].

[7] _209_db spends most of the time sorting a very large array pointing to records further pointing to vectors and then arrays.

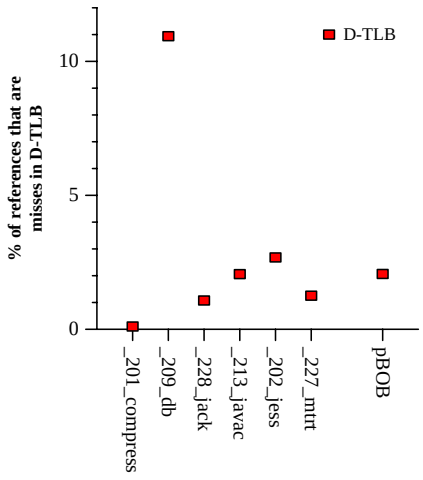

Figure 7: D-TLB misses

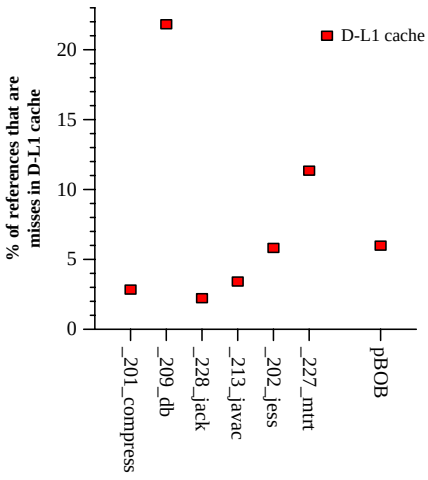

Figure 8: D-L1 cache misses

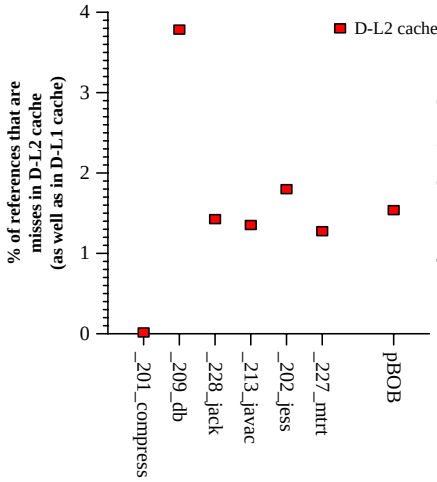

Figure 9: D-L2 cache misses

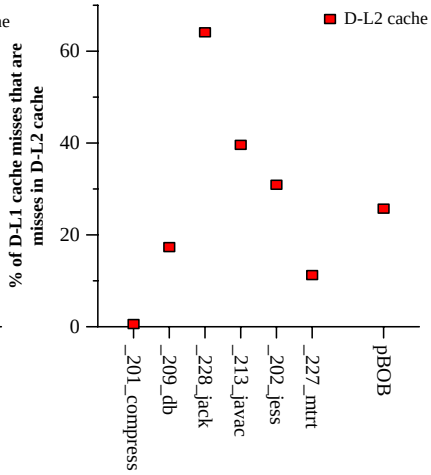

Figure 10: Effectiveness of D-L2 cache

4.4 The effectiveness of the L2 data cache

Fig. 10 shows how effective the L2 data cache is for reducing the number of times a request has to go all the way to memory. This graph plots the percentage of L1 data cache misses that miss in the L2 data cache. As we show in section 4.8 and Fig. 11 and 13, the number of compulsory/cold misses is small and should not bias the analysis of effectiveness of the L2 cache significantly. It turns out that for some benchmarks (like _227_mtrt), the L2 data cache is very effective and filters out most (90%) of the data L1 misses. However, for others (like _213_javac and _228_jack), it is not. For example, in _228_jack, over 60%

of references that miss in the L1 data cache also result in misses in the L2 data cache. This is important in designing cost effective (and low-power) systems for running Java workloads. Namely, even the complete absence of an L2 cache may not have a huge impact on the performance of a system whose major workload does not benefit from an L2 cache. Fig. 10 also tells us that a large L2 data cache is not going to solve all the performance problems motivating us to evaluate opportunities for prefetching data into the L2 cache later in this paper and in [32].

Overall, the effectiveness of the L2 cache for Java workloads seems to be worse than its effectiveness for commercial workloads written in C. For example, for TPC-B [4], 19% of accesses that reach the L2 cache lead to a miss. Similarly, for TPC-C [17], 12% of accesses that miss in the L1 cache, go all the way to memory. The L2 cache is shown to be somewhat effective [4] in capturing the footprint of TPC-D workload (with 13%-31% of local miss rates depending on a query). At the same time, the L2 cache is fairly ineffective [4] for the web index search workload (the Altavista search engine) where more than 40% of accesses that reach the off-chip cache result in a miss.

4.5 Methods causing data TLB and L2 data cache misses

In this section, we test the hypothesis that very few methods of a program are responsible for most of the performance problems. In this context, the performance problems we focus on are misses in the data TLB and L2 data cache.

We have sorted the list of methods according to their contribution to the total TLB and L2 cache miss rates and then discarded all those whose contribution is smaller than an importance threshold which is set to a small number (0.1%). The number of methods that cause most of the performance problems and their contribution to the total miss rate can be seen in Fig. 5 and Fig. 6.

It seems that in most benchmarks, the number of methods causing a lot of TLB and L2 misses is fairly small. However, the difference between _209_db and _213_javac is quite pronounced. These graphs show once again the inherent complexity of _213_javac and the simplicity of _209_db. The L2 cache miss rate of _201_compress is very low, and it appears that many methods in _201_compress contribute to a large fraction of all misses.

4.6 The relative complexity of the benchmarks

Interestingly, the seven benchmarks we are evaluating in this paper can be grouped into three clusters (based on the number of hot fields and virtual methods and on the number of methods that cause most of the performance problems). These clusters can be seen in Fig. 3-6. _201_compress and _209_db are really simple kernels. _213_javac and pBOB are truly complex programs. Finally, the remaining three are the benchmarks of medium complexity.

4.7 Reducing the number of L2 data cache misses may NOT be easy

We have also observed that many instructions that incur a L2 data cache miss also cause a data TLB miss. This fact is important but has been ignored probably due to low TLB miss rates of other benchmarks (e.g. benchmarks with regular memory access patterns and small data sets). An important implication is that non-blocking data prefetch instructions that are aimed at bringing in the data from memory will be squashed by processor hardware if the TLB does not contain appropriate translation information.[8] Hence, such a prefetching request will be of no use to an executing program.

A possible solution to this problem is not to discard a prefetching request on a TLB miss but to prefetch the translation information from a page table if it is not available in the TLB and then use it to prefetch the data [23, 10]. Obviously, such TLB prefetching can only work for the data residing on pages that can be accessed and are available in memory.

4.8 Classification of data related misses

We classify the cause of a miss to TLB and L2 cache that occurs on an access to heap-allocated data into the following categories: accesses to frequently referenced object fields and method table entries (i.e. those that are above the importance threshold, as depicted in Fig. 3 and Fig. 4); accesses to infrequently referenced object fields and method table entries; and accesses to array elements (for which the frequency of accesses to individual elements is less meaningful). Finally, an access can be categorized as a load or a store.

[8] For example, *data cache block touch* (DCBT) and *data cache block touch for store* (DCBTST) instructions in the PowerPC 604e [16] are treated as *no-ops* when there is a TLB translation miss.

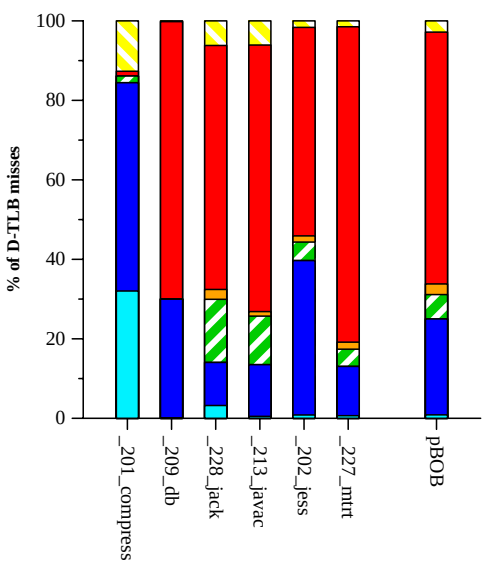

Figure 11: The distribution of D-TLB misses

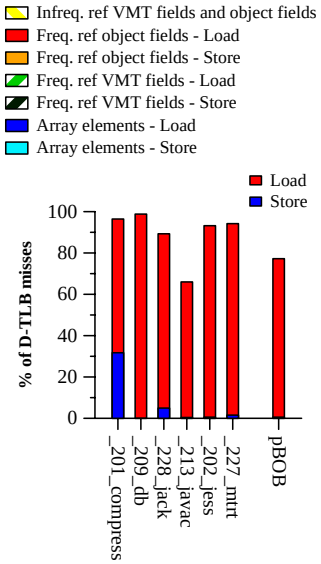

Figure 12: The types of misses in methods causing the most D-TLB misses

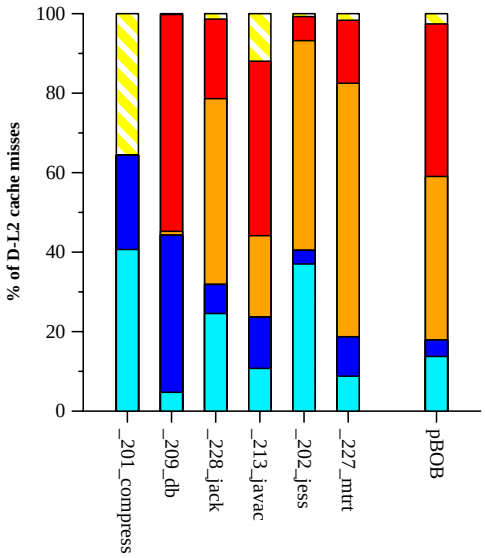

Figure 13: The distribution of D-L2 cache misses

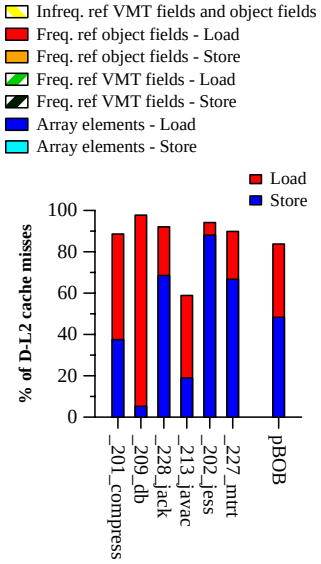

Figure 14: The types of misses in methods causing the most D-L2 cache misses

Sources of data TLB misses. As we expected, very few TLB misses are due to accesses to infrequently referenced object fields and method table entries (Fig. 11). The miss rates associated with accesses to object fields are larger than those associated with accesses to arrays.

This is probably due to better page locality of accesses to array elements as opposed to objects. Co-allocation of objects that are referenced at about the same time on the same page, done at allocation time [8] as well as at garbage collection time [9], can be useful. _201_compress has quite a few misses on accesses to infrequently referenced object fields and method table entries. This is probably because most of its working set consists of large arrays. _209_db seems to be accessing very few fields most of the time, and those accesses cause a lot of misses.

To our surprise, an unusually large fraction of TLB misses were due to accesses to method tables. The benchmarks, whose hot virtual methods reside in a fairly large number of classes, appear to suffer from this problem the most. This is probably due to the fact that the method tables are created only when a class is loaded. Because classes are loaded on demand, they end up far away from each other in the heap space and get placed on different pages, each of which has a separate entry in the translation table. A TLB miss would make a virtual method call even more expensive than it already is. A solution to this problem is to co-allocate hot method tables to ensure that they occupy a small number of pages to achieve higher utilization of TLB entries. Co-allocation of hot method tables can be done at GC time by combining the techniques reported in [9] and [30].

Interestingly, most of the misses to object fields and arrays are due to loads. There are no significant stores to the method tables, because once they are installed, they do not have to be modified. It is also interesting that most of the misses in the data TLB are due to loads. _201_compress is an exception, probably because it has to access elements of two large arrays sequentially about the same number of times while compressing and then uncompressing the content of one array into another.

Fig. 12 shows the types of misses in methods that contribute to the total TLB miss rate above a threshold (i.e. those that constitute Fig. 5). From this graph, we can see that most of the TLB misses happen on a load. The costs of servicing TLB misses that happen on a load or a store are comparable.

Sources of L2 data cache misses. Again, most of the L2 data cache misses are due to accesses to frequently referenced object fields and arrays (Fig. 13). The miss rates for object field accesses are larger than those for array accesses. _201_compress serves as an exception – it has quite a few misses on accesses to infrequently referenced object fields and method table entries. Almost all of the remaining misses in _201_compress are due to array accesses. This is probably because

most of its working set consists of large arrays, and all hot objects fit comfortably in the L2 data cache.

Interestingly, there were virtually no L2 cache misses due to accesses to the method table entries, even though TLB misses due to the same type of accesses were high. This is probably because the number of hot method table entries is small and all of them fit (and can co-reside with the rest of the working set) in the L2 data cache. Hence, field reordering within the method tables need not be necessary. However, it may become important for classes with a large number of virtual methods.

Fig. 14 shows the types of misses in methods that cause the largest numbers of misses (i.e. those that constitute Fig. 6). From this graph, we can see that for the majority of the benchmarks, most of the L2 cache misses happen on a store. This is a good sign because load misses in a cache are much harder to tolerate and hide. For some benchmarks (e.g. pBOB), load and store misses contribute about the same to the total miss rate. For others (e.g. _209_db), most of the misses in L2 happen on a load.

4.9 Assessment of opportunities for prefetching into data TLB and the L2 data cache

In this section, we determine the strategy for prefetching page table entries into the data TLB (assuming such a mechanism is available to us) and for prefetching data into the L2 data cache. Note that an instruction that causes a miss may not always do so. For example, if an instruction is executed a hundred times and causes a miss every time it is executed, then its own miss ratio is 100% [25]. On the other hand, if an instruction that is executed a hundred times causes a miss 10 times and 90 times it enjoys a hit, then its miss ratio is 10%. Prefetching all instances of the first kind of instruction is a good bet. The prefetch request is going to be useful and will bring in the data that otherwise would not be available when it is needed. Issuing prefetches for the second kind of instruction is more difficult because it is hard to know when such an instruction is going to experience a hit anyway, leading to a wasteful prefetch.

Compiler-based prefetching techniques for array-based and pointer-based applications have been studied in [24] and [21], respectively. The predictability of misses in the L1 data cache has been studied in [25]. In contrast, this work investigates the opportunity for prefetching into the data TLB and the L2 data cache. A complete set of graphs can be found in [32].

Opportunities for prefetching page table entries into data TLB.
Fig. 15-18 show how frequently different dynamic instances of instructions that contribute the most to data TLB misses[9] actually cause a miss. The histograms (marked (1)) show that the relevant loads fall under different categories in terms of their tendency to lead to a miss. However, the contribution of loads that almost always cause a miss is fairly high in most benchmarks – this can be seen by the steep slope, closer to the right hand side, of the line (marked (3)) showing the cumulative contributions of these instructions to the total TLB miss rate. This implies that once the important instructions which require prefetching are identified, relatively straightforward prefetching techniques could be used, as very few prefetch operations would be wasted (on account of instructions that lead to a hit).

Comparing the plots for loads and stores for each program, both the number of relevant loads and their contributions to the overall TLB miss rate are much higher. (In each plot, the horizontal dotted line shows the percentage contribution of loads (or stores) to the total number of TLB misses). The spectrum of miss causing store instructions, in terms of their miss rate distributions, is much narrower.

Opportunities for prefetching program data into L2 data cache.
Fig. 19-22 show how frequently different dynamic instances of instructions that contribute the most to L2 data cache misses[10] actually cause a miss. Most instances of the store instructions cause a miss, making it easier to prefetch stores without wasting prefetches on stores that hit in the L2 cache. While a similar behavior was observed for stores causing TLB misses, the narrow distribution is much more pronounced for L2 cache misses. Furthermore, unlike the case with TLB misses, the store instructions account for a higher percentage of L2 misses than loads (with a few exceptions – most notably, _209_db and _213_javac).

In contrast, the important load instructions show a wider distribution with respect to L2 cache misses over different instances. For some benchmarks (e.g. _227_mtrt), loads that miss less than 80% of the time

[9] We only consider methods whose contribution to the total D-TLB miss rate is greater than 1%. In those methods, we select those instructions whose individual contribution to the miss rate is greater than 0.25%. Often, there are relatively few instructions in the program that meet this criterion (the actual number is shown in each plot).

[10] As with the plots for TLB misses, we only consider methods whose contribution to the total L2 data cache miss rate is greater than 1%, and instructions whose contribution to the miss rate is greater than 0.25%. The number of instructions that meet this criterion is shown in each plot.

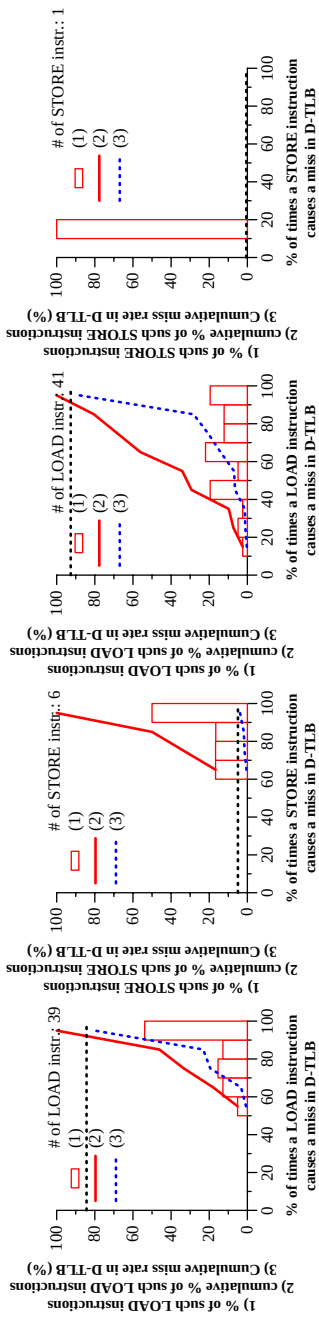

Figure 15: Frequency with which LOADs causing most of the misses in D-TLB lead to a miss _228_jack

Figure 16: Frequency with which STORESs causing most of the misses in D-TLB lead to a miss _228_jack

Figure 17: Frequency with which LOADs causing most of the misses in D-TLB lead to a miss _202_jess

Figure 18: Frequency with which STORESs causing most of the misses in D-TLB lead to a miss _202_jess

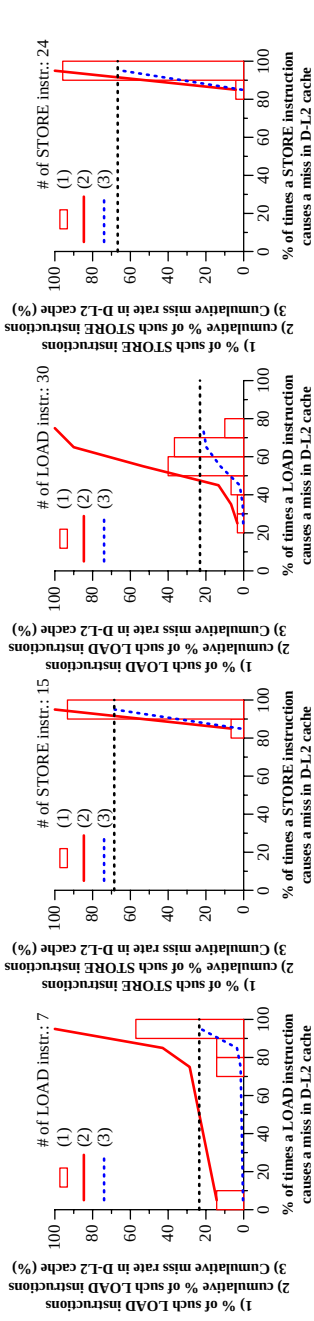

Figure 19: Frequency with which LOADs causing most of the misses in D-L2 cache lead to a miss _228_jack

Figure 20: Frequency with which STORESs causing most of the misses in D-L2 cache lead to a miss _228_jack

Figure 21: Frequency with which LOADs causing most of the misses in D-L2 cache lead to a miss _227_mtrt

Figure 22: Frequency with which STORESs causing most of the misses in D-L2 cache lead to a miss _227_mtrt

account for most of the misses. Issuing useful prefetches for loads in these programs is likely to be more difficult.

5. Related Work

Kim and Hsu [18] have investigated the memory system behavior of Java programs executed with a JIT. They study the lifetime characteristics of objects in Java programs, the temporal locality, and the impact of cache associativity on cache miss rate. They also study the performance impact of garbage collection under different heap sizes, and the effect of a heap size on the cache and the VM performance of applications.

Li et al. [20] use a complete system simulation to study SPECjvm98 benchmarks. Their study focuses on the OS activity during the execution of these benchmarks with a JIT compiler and without it (i.e. under interpretation). They find that most of the kernel activity is due to TLB miss handling and attribute many of those TLB misses to JIT compilation. In our work, we obtain results for a larger data size (recommended as the default size) and a more sophisticated run-time compilation strategy with a smaller cost. Hence, our work ends up reflecting more closely the user-level activity of Java applications.

Radhakrishnan et al. [26, 27, 29] study a wide variety of issues (at the bytecode and microarchitectural levels) associated with efficient execution of Java programs with an interpreter and a JIT compiler. Much of their detailed data is presented for SPECjvm98 programs with very small data sets (size 1, rather than the default size of 100). As a result, most of the activities they register are due to JIT compilation, which we believe is not representative of typical Java workloads. We use the default data set size (-s100) to study the behavior of longer-running applications.

Hsieh et al. [15] have studied cache and branch performance issues in the context of Java. The goal of their work was to compare the performance of the original C/C++ programs from SPEC95 and SPEC92 benchmarks with the performance of equivalent Java versions executed with an interpreter and compiled with their static compiler.

Bowers and Kaeli [6] study the SPECjvm98 benchmarks under a JVM with an interpreter and measure the relative frequencies of executed bytecode instructions (to determine what bytecodes need to be optimized first) and the distribution of the stack depths at different points in time (to estimate the number of processor registers needed to hold the content of stack elements).

Romer et al. [28] analyze several interpreters (including one for Java) from the software and hardware perspectives. They study the instruction

mix, performance bottlenecks, and the cache behavior of interpreters executing a set of common benchmarks.

Barisone et al. [3] analyze SPECjvm98 benchmarks executed with an interpreter, a JIT compiler, and the HotSpot compiler. They compare the bytecode and the native execution profile of these benchmarks as well as the overall behavior of a memory subsystem and a branch prediction mechanism under these three execution environments on a system with an UltraSparc-I processor.

6. Conclusions

We now summarize the main results from our study. A surprising result of our study is that for several benchmarks in the SPECjvm98 suite, the fraction of accesses to method tables is very small, while the fraction of accesses to array elements is quite significant. Those benchmarks are likely to be less object-oriented than others. Therefore, we believe that more truly object-oriented benchmarks (e.g. pBOB) should be added to the next version of SPECjvm to make it more useful. We have observed that some of the benchmarks have similar characteristics and can be clustered and defined as kernel benchmarks, benchmarks of medium complexity, and truly complex benchmarks. Only a few of the studied benchmarks (_213_javac and pBOB) fall in the last category.

Inherent Program Behavior. For the benchmarks studied, we can make the following observations about their behavior. The number of hot fields is very small in a few (kernel) benchmarks, and is substantially larger in two (more complex) benchmarks, although even in those two benchmarks, the number of hot fields is still small relative to the total number of fields. So are the number of hot virtual methods and the number of hot methods responsible for a large fraction of the data TLB and L2 data cache misses. However, because the number of hot spots is fairly small, even for complex benchmarks, relatively expensive algorithms can still be used to perform various run-time optimizations if the hot spots are identified effectively.

TLB Performance. Most of the misses in the data TLB are due to accesses to frequently referenced object fields. Overall, the data TLB performance is poor. A purely software approach to mitigate this problem would be to employ an intelligent object co-allocation scheme. On the hardware side, one could make data TLB larger, or add some support for large pages (e.g. 4M pages) to widen the TLB's coverage. Another possibility could be to add a special instruction that would prefetch a page table entry corresponding to a virtual address into an

L2 cache or some buffer from which it can be serviced on a TLB miss. We observed that loads typically account for most of TLB misses. We also found that different dynamic instances of instructions that make the largest contribution to data TLB misses rarely result in data TLB hits upon their execution. Hence, if prefetching of page table entries were supported by hardware, relatively simple prefetching strategies could prove to be effective.

Virtual Method Tables A noticeable fraction of data TLB misses are due to accesses to virtual method tables, even though the number of important (hot) virtual method tables is quite small. This observation indicates the importance of co-allocation of virtual method tables (and other `class` data) for different classes to reduce the cost of accessing the TIB (type information block) entries in virtual method calls and runtime type checking (such as `instanceof, checkcast, checkstore`). Interestingly, we have not observed many L2 data cache misses on references to entries in virtual method tables. At least for these benchmarks, hot virtual method tables can fit comfortably in an L2 cache of a reasonable size.

Cache Performance. Many data references miss in L1 data cache. One possible approach to mitigate this problem is to use prefetching.

Given a 32 kB L1 data cache, even a fairly large L2 data cache is not very effective for half of the benchmarks, and a system with a smaller or no L2 cache may not be significantly slower than a system with a large L2 cache. This observation should be important for the designers of cost conscious and low-power systems.

Most of the misses in the L2 data cache are due to accesses to frequently referenced object fields. Overall, most of the instructions causing L2 data cache misses are stores, which is interesting because store misses are easier to tolerate compared to load misses.

The variation in the behavior of different instances of important loads that contribute the most to L2 misses indicates that in some programs, relatively sophisticated strategies would be needed to avoid wasting prefetch operations on load instances that result in a hit.

Relationship between Cache and TLB Misses. We have observed that many loads and stores that miss in the L2 data cache also miss in the data TLB. As a result, a cache line prefetch instruction that one could issue for such a load or store miss would be squashed by hardware. Therefore, we believe that when doing prefetching into the L2 cache, the hardware should provide an option where the appropriate page table entry is prefetched if it is not available in the TLB.

Comparison with other Workloads. Overall, the TLB and cache behavior of Java programs seems to be worse than that of technical benchmarks and is comparable to the behavior of commercial benchmarks. Therefore, enterprise servers whose memory subsystems are tuned for commercial workloads should be able to handle the demands of Java workloads well. At the same time, pointer chasing and large working sets of some Java programs can make an L2 cache somewhat ineffective. Consequently, cost sensitive and power conscious devices may often be able to perform just as well without an expensive and power-hungry L2 cache.

References

[1] B. Alpern, A. Cocchi, D. Lieber, M. Mergen, and V. Sarkar. Jalapeño - a compiler-supported Java virtual machine for servers. In *Workshop on Compiler Support for Software System (WCSSS 99)*, May 1999.

[2] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proc. of OOPSLA 2000*, Oct. 2000.

[3] A. Barisone, F. Belliotti, R. Berta, and A. Gloria. Ultrasparc instruction level characterization of Java virtual machine workload. In *2nd Annual Workshop on Workload Characterization (WWC) for Computer System Design*, pages 1–24. Kluwer Academic Publishers, 1999.

[4] L. A. Barroso, K. Gharachorloo, and E. Bugnion. Memory system characterization of commercial workloads. In *Proc. of ISCA-25*, pages 3 – 14, 1998.

[5] S. J. Baylor, M. Devarakonda, S. Fink, E. Gluzberg, M. Kalantar, P. Muttineni, E. Barsness, R. Arora, R. Dimpsey, and S. J. Munroe. Java server benchmarks. *IBM Systems Journal*, 39(1):57–81, 2000.

[6] K. R. Bowers and D. Kaeli. Characterizing the SPEC JVM98 benchmarks on the Java virtual machine. Technical report, Northeastern University, Dept. of ECE, Computer Architecture Group, 1998.

[7] M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño dynamic optimizing compiler for Java. In *Proc. of ACM SIGPLAN 1999 Java Grande Conference*, June 1999.

[8] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *Proc. of PLDI 1999*, pages 1 – 12, May 1999.

[9] T. M. Chilimbi and J. R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *Proc. of the*

1998 International Symposium on Memory Management (ISMM), Oct. 1998.

[10] D. Culler, J. P. Singh, and A. Gupta. *"Parallel Computer Architecture: A Hardware/Software Approach"*. Morgan Kaufmann Publishers, 1998.

[11] Z. Cvetanovic and D. Bhandarkar. Characterization of Alpha AXP performance using TP and SPEC workloads. In *Proc. of ISCA-21*, pages 60–70, Apr. 1994.

[12] J. Dolby and A. Chien. An automatic object inlining optimization and its evaluation. In *Proc. of PLDI 2000*, pages 345 – 357, June 2000.

[13] J. Gosling, B. Joy, and G. Steele. *The Java$^{(TM)}$ Language Specification*. Addison-Wesley, 1996.

[14] The Java Hotspot Performance Engine Architecture. `http://java.sun.com/products/hotspot/whitepaper.html`.

[15] C.-H. Hsieh, M. T. Conte, J. C. G. T. L. Jonson, and W. W. Hwu. A study of the cache and branch performance issues with running Java on current hardware platforms. In *Proc. of IEEE COMPCON*, pages 211–216, 1997.

[16] IBM Corp. *PowerPC 604e RISC Microprocessor User's Manual*, Mar. 1998. G522-0330-00.

[17] K. Keeton, D. Patterson, Y. He, R. Raphael, and W. Baker. Performance characterization of a Quad Pentium Pro SMP using OLTP workloads. In *Proc. of ISCA-25*, pages 15–26, 1998.

[18] J.-S. Kim and Y. Hsu. Memory system behavior of Java programs: Methodology and analysis. In *Proc. of SIGMETRICS 2000*, pages 264 – 274, June 2000.

[19] D. Lee, P. Crowley, J.-L. Baer, T. Anderson, and B. Bershad. Execution characteristics of desktop applications on Windows NT. In *Proc. of ISCA-25*, pages 27–38, 1998.

[20] T. Li, L. John, V. Narayanan, A. Sivasubramaniam, J. Sabarinathan, and A. Murthy. Using complete system simulation to characterize SPECjvm98 benchmarks. In *Proc. of ICS 2000*, May 2000.

[21] C.-K. Luk and T. Mowry. Compiler-based prefetching for recursive data structures. In *Proc. of ASPLOS-VII*, pages 222–233, Oct. 1996.

[22] A. Maynard, C. Donnelly, and B. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *Proc. of ASPLOS VI*, Oct. 1994.

[23] T. Mowry. *Tolerating latency through software-controlled data prefetching*. PhD thesis, Stanford University, Mar. 1994.

[24] T. Mowry, M. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proc. of ASPLOS-V*, pages 62–73, Oct. 1992.

[25] T. Mowry and C.-K. Luk. Predicting data cache misses in non-numeric applications through correlation profiling. In *Proc. of Micro-30*, Dec. 1997.

[26] R. Radhakrishnan, J. Rubio, L. John, and N. Vijaykrishnan. Execution characteristics of just-in-time compilers. Technical Report TR-990717-01, Department of Electrical and Computer Engineering, University of Texas at Austin, 1999.

[27] R. Radhakrishnan, N. Vijaykrishnan, L. K. John, and A. Sivasubramaniam. Architectural issues in Java runtime systems. In *Proc. of HPCA-6*, pages 387–398, Jan. 2000.

[28] T. Romer, D. Lee, G. Voelker, A. Wolman, W. Wong, J. Baer, B. Bershad, and H. Levy. The structure and performance of interpreters. In *Proc. of ASPLOS VII*, pages 150–159, Oct. 1996.

[29] J. R. R.Radhakrishnan and L. John. Characterization of Java applications at bytecode and Ultra-SPARC machine code levels. In *Proc. of IEEE ICCD*, pages 281–284, Oct. 1999.

[30] W. Schmidt, R. Roediger, C. Mestad, B. Mendelson, I. Shavit-Lottem, and V. Bortnikov-Sitnitsky. Profile-directed restructuring of operating system code. *IBM Systems Journal*, 37(2):270, 1998.

[31] M. Serrano, R. Bordawekar, S. Midkiff, and M. Gupta. Quicksilver: A quasi-static compiler for Java. In *Proc. of OOPSLA 2000*, Oct. 2000.

[32] Y. Shuf, M. J. Serrano, M. Gupta, and J. P. Singh. Characterizing memory behavior of Java workloads: A structured view and opportunities for optimizations. Technical report, IBM T.J. Watson Research Center, Yorktown Heights, NY, 2000.

[33] Standard Performance Evaluation Council. *SPEC JVM98 Benchmarks*, 1998. `http://www.spec.org/osg/jvm98/`.