1-1-2006

TLB pre-loading for Java applications

Bashar Mahmoud Gharaibeh

Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

Recommended Citation

Gharaibeh, Bashar Mahmoud, "TLB pre-loading for Java applications" (2006). Retrospective Theses and Dissertations. 19014.
https://lib.dr.iastate.edu/rtd/19014

This Thesis is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
TLB pre-loading for Java applications

by

Bashar Mahmoud Gharaibeh

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
J. Morris Chang, Major Professor
Wensheng Zhang
Zhao Zhang

Iowa State University
Ames, Iowa
2006

Copyright © Bashar Mahmoud Gharaibeh, 2006. All rights reserved.
Graduate College
Iowa State University

This is to certify that the master's thesis of
Bashar Mahmoud Gharaibeh
has met the thesis requirements of Iowa State University

Signatures have been redacted for privacy
TABLE OF CONTENTS

LIST OF TABLES ......................................................... v
LIST OF FIGURES ......................................................... vi
ABSTRACT ................................................................. vii
1 INTRODUCTION ......................................................... 1
2 PROBLEM ANALYSIS ...................................................... 5
   2.1 Experimental Setup ............................................... 6
      2.1.1 Dynamic SimpleScalar (DSS) ................................. 6
      2.1.2 The Java Virtual Machine: JikesRVM ......................... 7
      2.1.3 Java Applications ........................................... 9
   2.2 Quantifying TLB Latency ......................................... 9
   2.3 TLB misses Distribution ........................................ 10
   2.4 Visualizing TLB Miss Patterns .................................. 11
   2.5 Conclusion ...................................................... 13
3 APPLICATION BEHAVIOR ............................................. 14
   3.1 Java Application’s Memory Access .............................. 14
   3.2 Reference Modification and TLB Misses ....................... 16
   3.3 Reference Modification Patterns ............................... 18
   3.4 Differences Between Consecutive Events ..................... 18
   3.5 Concluding Remarks ........................................... 20
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 2.1</td>
<td>DSS configuration parameters.</td>
<td>7</td>
</tr>
<tr>
<td>Table 2.2</td>
<td>Heap regions division</td>
<td>8</td>
</tr>
<tr>
<td>Table 2.3</td>
<td>Java Applications</td>
<td>9</td>
</tr>
<tr>
<td>Table 2.4</td>
<td>The effect of TLB miss latency on execution time.</td>
<td>10</td>
</tr>
<tr>
<td>Table 2.5</td>
<td>Distribution of TLB misses over heap regions</td>
<td>11</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>2.1</td>
<td>TLB misses. X-axis represents the number of executed instructions, while the Y-axis shows the page number to which the miss address belongs, starting from the MS region start address</td>
<td>12</td>
</tr>
<tr>
<td>3.1</td>
<td>Different memory management schemes, different colors means different page assignment, the dashed region represent a single heap region</td>
<td>16</td>
</tr>
<tr>
<td>3.2</td>
<td>Coverage and Inverse Coverage ratios</td>
<td>18</td>
</tr>
<tr>
<td>3.3</td>
<td>Write Barrier Addresses: X-axis represents the number of executed instructions, while the Y-axis shows the page number to which the target address belongs.</td>
<td>19</td>
</tr>
<tr>
<td>3.4</td>
<td>Offset CDF: The X-axis represents offsets values. The Y-axis represents cumulative probability.</td>
<td>21</td>
</tr>
<tr>
<td>4.1</td>
<td>Autocorrelation plots. X-axis represents the lag value, and the Y-axis shows the normalized correlation factor.</td>
<td>25</td>
</tr>
<tr>
<td>4.2</td>
<td>Effect of polynomial size on accuracy.</td>
<td>28</td>
</tr>
<tr>
<td>4.3</td>
<td>Prediction Accuracy</td>
<td>29</td>
</tr>
<tr>
<td>5.1</td>
<td>Linear Predictor Performance. Relative to the base JVM</td>
<td>32</td>
</tr>
<tr>
<td>5.2</td>
<td>Linear Predictor Accuracy vs. Performance. Each point represents a different benchmark</td>
<td>33</td>
</tr>
</tbody>
</table>
The increasing memory requirement for today's applications is causing more stress for the memory system. This side effect puts pressure into available caches, and specifically the TLB cache. TLB misses are responsible for a considerable ratio of the total memory latency, since an average of 10% of execution time is wasted on miss penalties.

Java applications are not in a better position. Their attractive features increase the memory footprint. Generally, Java applications TLB miss rate tends to be multiples of miss rate for non-java applications. The high miss rate will cause the application to loose valuable execution time. Our experiments show that on average, miss penalty can constitute about 24% of execution time.

Several hardware modifications were suggested to reduce TLB misses for general applications. However, to the best of our knowledge, there have been no similar efforts for java applications. Here we propose a software-based prediction model that relies on information available to the virtual machine. The model uses the write barrier operation to predict TLB misses with an average 41% accuracy rate.
1 INTRODUCTION

Java is emerging as one of the most popular paradigms for software development. According to the TIOBE programming community index listing on April 2006 [32], Java was considered the most popular mainstream programming language. Java employs a sophisticated run-time environment (i.e. Java Virtual Machines) to enable an array of advanced features, such as automatic memory management, cross-platform portability and enforced security checks. The versatility brought by this extra layer comes with the high cost of performance degradation.

The speed gap between processors and main memory will continue to widen. Recent studies have shown that memory latency can constitute about 50% of Java applications execution time [5]. Several components contribute to the latency, such as cache and TLB misses. Although researchers studied the problem of cache misses in great detail [1, 15, 22], TLB misses [30] are studied less frequently. Recent studies have shown that about 10% of the execution time is contributed to TLB misses in commercial applications (written in C/C++) [26]. However, our experiments indicate that Java applications (SPECJVM98 benchmarks) can spend, on average, about 24% of their execution time resolving TLB misses (see Table 2.4). This can be attributed to the large memory footprint required for Java applications [30] compared to applications written in C/C++. To the best of our knowledge, TLB miss behavior and potential solutions for Java applications have not been investigated. Here we study the TLB miss patterns for Java applications first, then propose a scheme to manage the TLB misses in Java Virtual Machines.
Several schemes were proposed to reduce the effect of TLB misses, those approaches can be divided into two directions. The first focuses on studying the effect of TLB structures (e.g. associativity, block sizes and multi-level) on TLB misses [17]. The second approach aims at reducing TLB misses through prediction. In [21, 26], a predictor is proposed to predict future translation misses based on previous ones. Then predicted address translations are fetched. However, all these approaches require hardware modifications, thus limiting the benefits to processors having these modifications. In this thesis, we propose a software-based solution to reduce the number of TLB misses. The proposed approach does not require hardware modifications, and relies on monitoring the application behavior to predict TLB misses.

Our scheme aims at reducing the number of TLB misses caused by the application rather than those caused by the Java virtual machine services. We employ a software-based predictor within Java virtual machine to preload the TLB with predicted future misses. Previously proposed prediction schemes relied on monitoring TLB misses to predict future misses. This monitoring process requires support from hardware.

The proposed prediction scheme is based on the application memory access pattern. We argue that the memory access patterns can be correlated to TLB misses. Our analysis in chapter 3 on the memory access patterns and TLB misses reveals the existence of the correlation. Based on the correlation, we propose a prediction scheme that monitors memory access patterns from the Java Virtual Machine.

In Java, objects are connected through references (i.e. object pointers). These references are stored in object’s fields. Whenever these references change, they would cause a change in the memory access pattern. The application accesses objects by following references, this procedure is called object traversal. When a reference changes, the new value will point to another set of objects. The new reference will allow the application to traverse another set of objects. The traversal change causes the memory access patterns to change, which affects the TLB miss pattern. A typical way of reference
change is done through writing a new value into an object's reference field.

The proposed modification to virtual machine, needs to process access patterns and issue TLB predictions accordingly. Java virtual machine provides the functionality of monitoring reference modifications using a *write barrier*. It has been developed to assist generational garbage collectors to record references between different memory regions. This type of garbage collectors is widely used within modern Java virtual machine implementations because of its short pause time [11, 22]. The write barrier helps with remembering references between different memory regions (generations), and it is invoked when the application tries to modify a reference field in an object. Generally, the write barrier receives two addresses. The first is the address of the object to be modified, the second is the address being written into the reference field, which is the value we are interested in. The use of the write barrier has two benefits: Firstly, it gives us the opportunity to monitor the application without extra instrumentation or profiling for the application. Secondly, it provides us with the needed reference without requiring any modifications.

Our prediction algorithm is based on the *linear prediction* model, which is a mathematical model known to the field of DSP (Digital Signal Processing) [34]. The typical use of linear prediction is to analyze a set of observed events to generate a prediction function. This function is used with a smaller set of observed events to predict future events. In our implementation, references from the write barrier are supplied to the linear predictor to predict future TLB misses. Simulation results showed that, for SPECjvm98, the predictor has an average prediction accuracy of about 41%, while previously proposed hardware-based predictors [21, 26] has an average accuracy of 36% for the same benchmarks.

The rest of this thesis is organized as follows: The experimental platform and Detailed analysis of TLB misses is given in Chapter 2. Chapter 3 presents key aspects of the application behavior. Results from the previous two chapters are used in Chapter 4 to
propose a solution. The implementation of the predictor within Java virtual machine is evaluated in Chapter 5. Chapter 6 shortly reviews related work. Finally, we state our conclusion in Chapter 7.
2 PROBLEM ANALYSIS

The use of virtual memory allowed applications to increase their memory footprint beyond the size of the physical main memory. Current processors and operating systems support virtual to physical mapping using different techniques. Although virtual memory provides many benefits from the application perspective, it is noted however, that translating virtual addresses to physical memory increases execution time. The translation is required on every memory reference, causing the processor to request the translation from the physical memory through the Memory Management Unit (MMU). Requesting the translation information adds a considerable overhead on each memory operation, since it needs two memory accesses. The first to get the translation information. The second is for the issued operation.

To reduce the overhead of the translation process, processors incorporate a special buffer that holds recently used translations. This buffer, called Translation Look-aside Buffer (TLB), would reduce the translation overhead by reducing the number of memory accesses to get translation information. Having the TLB within the processor means that translations can be accessed with the processor speed, rather than the slower main memory speed.

Even with the TLB existence, the translation process can still cause latencies if the TLB does not hold the needed translation. TLB misses have a non-negligible overhead, we have found that about 24% of Java application execution time is attributed to TLB miss latencies. Usually, separate TLBs exist for instruction and memory references. However, the instruction TLB (i-TLB) is known to have a negligible effect on execution
Because of its role in reducing translation latency, the TLB was targeted by several optimization efforts. These efforts can be divided into hardware and software optimizations. Hardware optimizations targeted the TLB structure to reduce the number of misses and miss latencies. Such as investigating the effect of the TLB size and associativity on reducing the number of misses, or the use of multi-level TLB to reduce the miss latency [6, 17]. Moreover, Software-based optimizations were also proposed to reduce misses by enhancing data locality. Although those schemes have their direct effect on data caches, a better locality can also reduces TLB misses.

For the remainder of this chapter, we will first discuss the tools used for our experiments. Then we will analyzes different aspects of TLB misses, such as the overhead of TLB misses, TLB miss targets and TLB miss patterns.

2.1 Experimental Setup

2.1.1 Dynamic SimpleScalar (DSS)

DSS [14] is a variation of the known simulator SimpleScalar [4]. The key difference is that DSS support applications that uses Just-In-Time(JIT) compilers, which optimizes the application code and rewrites it during runtime. This type of compilers is widely used within Java virtual machines, but it requires special support from the simulator to handle changing instructions. The simulator replaces instructions with handlers to functions that simulate a particular instruction. If instructions were changed or moved by JIT, the simulator should update the previous mapping. Other features are also supported by DSS, such as thread synchronization and virtual memory management through special instructions and system calls. Furthermore, the simulator allows the user to configure most of its functional units, such as caches and TLBs. Table 2.1 lists parameters used to configure DSS for all consequent experiments.
Table 2.1 DSS configuration parameters.

<table>
<thead>
<tr>
<th>Unit</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data TLB</td>
<td>256 entries, 2-way set associative, 4k pages</td>
</tr>
<tr>
<td>Instruction TLB</td>
<td>128 entries, 2-way set associative, 4K pages</td>
</tr>
<tr>
<td>L1 Data cache</td>
<td>512 entries, 2-way set associative, 32 byte block</td>
</tr>
<tr>
<td>L1 Instruction cache</td>
<td>265 entries, direct mapped, 32 byte block</td>
</tr>
<tr>
<td>L2 unified cache</td>
<td>4K entries, 2-way set associative</td>
</tr>
</tbody>
</table>

We have modified DSS to report addresses that caused D-TLB misses. When an address supplied to the data TLB for translation causes a miss, the event will be written to a trace file. The event information consists of the following: The address divided by the page size to record the page number, and the number of instruction executed so far. Recorded events will be processed later on to remove TLB events that are not generated by the application through examining the miss address.

2.1.2 The Java Virtual Machine: JikesRVM

JikesRVM [2] is a research-oriented Java virtual machine developed by IBM. Most of the virtual machine code is written in Java, and it uses an aggressive optimizing compilers that compiles both the application byte-codes and the virtual machine code. Allowing the virtual machine to inline parts of its code within the Application to enhance performance. Moreover, JikesRVM can be configured to use one of several memory management policies. Each policy consists of an allocation mechanism responsible for allocating objects into memory, and a collection mechanism (i.e. Garbage Collection) responsible for deleting unneeded objects.

The memory management policy used in our experiments is GenMS (Generational Mark-Sweep), which uses a Mark-Sweep collection mechanism for its mature space. The Generational collector works by assigning the objects into different memory regions(spaces) based on their age, Table 2.2 gives the heap layout for GenMS. New objects
are allocated into the *Nursery* space until it fills up. Then, the collection mechanism scans the nursery for objects still needed by the application and promotes them to the *Mature* space. Some objects in the mature space may reference objects in the nursery before the collection starts. After promoting referenced objects from the nursery, references within mature objects should be updated to point to the promoted object’s new location. The update procedure needs a list of references that points to the nursery space from the mature space. This list is maintained by the *write barrier*, since it captures reference modifications, and will further process references that connects mature objects to nursery objects.

**Table 2.2 Heap regions division**

<table>
<thead>
<tr>
<th>Region</th>
<th>Start address</th>
<th>End address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boot</td>
<td>0x31000000</td>
<td>0x40FFFFFF</td>
</tr>
<tr>
<td>Immortal</td>
<td>0x41000000</td>
<td>0x42FFFFFF</td>
</tr>
<tr>
<td>Meta data</td>
<td>0x43000000</td>
<td>0x44FFFFFF</td>
</tr>
<tr>
<td>Large object space (LOS)</td>
<td>0x45000000</td>
<td>0x4c7FFFFF</td>
</tr>
<tr>
<td>Nursery</td>
<td>0x74c00000</td>
<td>0x7FFFFFFF</td>
</tr>
<tr>
<td>MS</td>
<td>0x4c800000</td>
<td>0x723FFFFF</td>
</tr>
</tbody>
</table>

JikesRVM contains an implementation for the write barrier, which is invoked when the application (*Mutator*) changes a reference value within an object. The write barrier is also available within other commercial JVMs. We have modified the write barrier to supply the value being written into the reference field to the simulator. The barrier usually process reference updates that points outside the mature space. However, the modified version supplies all reference values regardless of the target region they point to (including the mature space). Communication between the virtual machine and the simulator is done through virtual devices. Those devices are allocated by ordering the simulator to reserve a page in memory. Any access (read or write) to this page will be captured by the simulator, and interpreted as a device access. We have implemented a virtual device within DSS to receive values from the write barrier and print them into
Table 2.3  Java Applications

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_201_compress</td>
<td>An implementation of the LZW compression algorithm</td>
</tr>
<tr>
<td>_202_jess</td>
<td>Expert shell system, based on NASA’s CLIPS expert shell system</td>
</tr>
<tr>
<td>_205_raytrace</td>
<td>Scene rendering</td>
</tr>
<tr>
<td>_209_db</td>
<td>Perform Database functions on a memory resident database</td>
</tr>
<tr>
<td>_213_javac</td>
<td>Java compiler from JDK 1.0.2</td>
</tr>
<tr>
<td>_222_mpegaudio</td>
<td>Compresses audio files into MPEG layer-3 standard</td>
</tr>
<tr>
<td>_228_jack</td>
<td>Parser generator</td>
</tr>
</tbody>
</table>

the trace file.

2.1.3  Java Applications

The set of Java application considered belong to the SPECJVM98 suite [31]. They represent a wide range of application classes. Table 2.3 list those applications along with a brief description.

2.2  Quantifying TLB Latency

Our first step in analyzing the TLB problem is to study its effect on execution time. Recent studies have shown that about 10% of execution time is contributed to TLB misses in commercial applications written in C/C++ [26]. However, our experiments indicates that Java applications can spend, in average, about 24% of their execution time resolving TLB misses. The results were obtained by simulating the SPECJVM98 benchmarks using DSS. Each benchmark was simulated twice, the first using a perfect TLB that hits on every access. The second using the TLB configuration described in Table 2.1. In all experiments, the simulator keeps track of the number of cycles spent on the garbage collection and the mutator phase.

Comparing the number of mutator cycles between the two TLB configurations shows the percentage of time wasted on TLB misses caused by the mutator. This is calculated
Table 2.4 The effect of TLB miss latency on execution time.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Mutator TLB Misses Overhead</th>
<th>Miss Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compress</td>
<td>2.1%</td>
<td>0.0012</td>
</tr>
<tr>
<td>Jess</td>
<td>23.8%</td>
<td>0.0074</td>
</tr>
<tr>
<td>DB</td>
<td>56.5%</td>
<td>0.2310</td>
</tr>
<tr>
<td>Jack</td>
<td>33.4%</td>
<td>0.0104</td>
</tr>
<tr>
<td>Mpeg</td>
<td>6.9%</td>
<td>0.0013</td>
</tr>
<tr>
<td>RayTrace</td>
<td>26.5%</td>
<td>0.0030</td>
</tr>
<tr>
<td>Average</td>
<td>24.9%</td>
<td>0.058</td>
</tr>
</tbody>
</table>

using the following equation:

\[ T_{TLB} = \frac{C_{Base} - C_{PerfectTLB}}{C_{Base}} \times 100\% \]

where \( T_{TLB} \) is the percentage of time wasted due to TLB misses, \( C_{Base} \) is the number of mutator cycles using the TLB configuration in Table 2.1, and \( C_{PerfectTLB} \) is the number of mutator cycles using a perfect TLB. Table 2.4 presents the results for individual benchmarks.

Although TLB miss rates shown in Table 2.4 are considered low (i.e. an average of 58 misses per 1000 references), their effect on execution time is not to be ignored because of the high cost associated with the misses. Furthermore, the miss ratio and accordingly the effect on execution time is higher for Java applications compared to applications written in C/C++ [30].

2.3 TLB misses Distribution

Our second step of analysis is to examine what regions in memory do references that caused a TLB miss target. TLB miss events captured by the simulator were classified based on the heap region they target. All events that occurred during GC were neglected, since we are interested in the events generated by the application rather than the memory management policy. The ratio for each memory region in is shown in Table 2.5.
Table 2.5 Distribution of TLB misses over heap regions

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Boot</th>
<th>Immortal</th>
<th>Meta-Data</th>
<th>LOS</th>
<th>Nursery</th>
<th>MS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compress</td>
<td>36.8%</td>
<td>0.73%</td>
<td>1.13%</td>
<td>40.3%</td>
<td>7.6%</td>
<td>13.5%</td>
</tr>
<tr>
<td>Jess</td>
<td>27.6%</td>
<td>2.4%</td>
<td>0.59%</td>
<td>2.2%</td>
<td>35.8%</td>
<td>31.4%</td>
</tr>
<tr>
<td>DB</td>
<td>7%</td>
<td>0.25%</td>
<td>0.27%</td>
<td>1.18%</td>
<td>2.38%</td>
<td>88.9%</td>
</tr>
<tr>
<td>Jack</td>
<td>44.5%</td>
<td>1.94%</td>
<td>3.15%</td>
<td>3.38%</td>
<td>20.5%</td>
<td>26.6%</td>
</tr>
<tr>
<td>Mpeg</td>
<td>46.8%</td>
<td>7.63%</td>
<td>0.87%</td>
<td>3.76%</td>
<td>20.7%</td>
<td>20.2%</td>
</tr>
<tr>
<td>RayTrace</td>
<td>17.5%</td>
<td>1.11%</td>
<td>0.65%</td>
<td>6.22%</td>
<td>6.73%</td>
<td>67.8%</td>
</tr>
<tr>
<td>Average</td>
<td>30.0%</td>
<td>2.35%</td>
<td>1.11%</td>
<td>9.51%</td>
<td>15.6%</td>
<td>41.4%</td>
</tr>
</tbody>
</table>

Table 2.5 shows that TLB events are not distributed evenly over all memory regions. For example, the Immortal and Meta-Data regions have insignificant contribution on the total TLB misses, and except for Compress, large object space (LOS) region has minor contribution on total TLB misses. Most of TLB events are caused by references to the other three regions: Mature space (MS), Nursery and Boot. The MS region has the highest average contribution over the Nursery and the Boot regions. On the other hand, write barrier targets should be distributed over the LOS, Nursery and MS regions.

2.4 Visualizing TLB Miss Patterns

Figure 2.1 plots TLB miss events that target the MS region, since it is the target for a high ratio of TLB misses. For each plot, the X-axis represents the number of executed instructions, while the Y-axis gives the page number of the miss address. Although it seems that several misses occur for a single instruction. However, this is not the case here, since the X-axis spans millions of instructions. For visibility purposes, the plots represents events occurring between two GC invocations.

Several benchmarks such as Compress (Figure 2.1(a)), DB (Figure 2.1(c)) and RayTrace (Figure 2.1(e)) show a clear repetitive pattern in their TLB miss addresses. It can be also noted that TLB misses are not evenly distributed over the whole MS region. All
Figure 2.1 TLB misses. X-axis represents the number of executed instructions, while the Y-axis shows the page number to which the miss address belongs, starting from the $MS$ region start address.
plots shows a range in the Y-axis that does not have TLB miss events (i.e. Figure 2.1(f)). These address gaps are caused by the allocation policy for the MS region. The policy segregates objects based on their size, objects that belong to the same size class are allocated in a reserved block in memory. Some of the blocks can be empty, since the application is not allocating objects that belong to that size class. This means that some memory block will never be referenced, and consequently, will never have TLB misses. As seen in the plots, the address gaps are different between benchmarks, since the gap depends on the unused blocks in memory, which in turn, depends on the size of objects allocated by the application.

2.5 Conclusion

As noted, TLB misses have a considerable effect on Java application’s execution time. We found that misses tend to have repeated patterns. However, till this point, we have not looked into what is the reason behind these patterns. Determining the reason behind the patterns can provide us with more insight about the miss behavior. Which allows as to better predict them in advance.
3 APPLICATION BEHAVIOR

A TLB miss occurs when an address supplied for translation does not exist in the TLB, therefore, misses and addresses should have some common characteristics. Those addresses are a result of the running application memory access behavior. By studying the application memory access patterns, we can have an insight over the underlying TLB misses.

Previously, several researchers studied the application's memory reference pattern to help lower data cache misses. For example, in [7] references were analyzed to find frequently repetitive patterns. While in [15], references were analyzed to discover common strides between consecutive references. In both cases, the analysis information were used for prefetching future references. A comprehensive survey on several related approaches can be found in [3].

In this chapter, we are going to study Java application's memory access patterns. First, we will generally discuss factors that shape and affect the reference pattern. Then we will analyze the reference patterns for the simulated applications and study their relation to TLB misses. Finally, we will lay out the basis for our proposed preloading algorithm.

3.1 Java Application's Memory Access

Objects in Java are connected through references (i.e. object pointers). Let us consider an object graph $G$ depicted in Figure 3.1. The graph consists of vertices $O$
that represent objects, and directional edges $R$ between the vertices, where the edges represent references between objects. The application accesses the objects by traversing those references until it reaches the needed object. The graph characteristics, such as the number of objects, the number of references in each object and the connectivity between objects, are all determined by the application. However, reference values reflect the target object's location in memory, which is dependent on the memory management policy. Generally, references are managed by the following operations:

- Initialization, by allocating an object and assigning a reference field to the created object location.  

- Mutation, where references are changed to reference other objects.

- Garbage collection, where objects are copied from one space to another.

Those operations are responsible for changing the reference field's value. When a reference changes, connections between objects will also change, causing a restructuring of the object graph.

Objects are allocated into pages in memory, and different memory management policies may assign the same object to a different page based on the allocation policy. Furthermore, the division of the memory into regions is dependent on the memory management policy used, so the same application object may get assigned to a different location if the virtual machine uses a different memory management policy. Although the application decides which objects to traverse, the resulting memory access sequence depends on the memory management policy used. Formally, we can represent the memory access sequence using the following relation:

$$R = F_{MM}(V)$$

\(^1\text{We can think of reference first assignment as modification from null to another value}\)
Where $R$ is the memory access sequence, $V$ is the set of object to be traversed, and $F_{MM}()$ is a function that maps an object to its location in memory based on the memory management policy ($MM$). If the application does not change $V$, then the memory access sequence would remain the same. However, if the application changes $V$ through reference mutation, the memory access sequence will change accordingly.

### 3.2 Reference Modification and TLB Misses

Because we are mainly interested in TLB misses, we can neglect references that do not cross memory pages (i.e. the pointing and the target objects are in the same page), we will call these references *page-local references*. A TLB miss occurs when the needed page translation is not available in TLB. However, if the application traverse page-local references, it will only cause a single TLB miss if the page translation was not available when the application started the traversal. On the other hand, references that cross memory pages are more likely to cause TLB misses than page-local references, since they require a page translation to access the target page.
The object graph in Figure 3.1 represents a snapshot of how objects are connected together at a certain point of time. If the graph remains constant through the application execution, and if the application keeps following the same references, memory access sequence will also be static. However, some references between objects can change to point to other objects as required by the application. This would change the memory access sequence each time a reference is changed. In order to measure the impact of reference mutations into TLB misses, we first need to study the relation between TLB miss addresses and reference changes.

To find out whether reference mutation matches TLB misses, we need to find the coverage ratio between TLB misses and modified references. The coverage ratio gives the fraction of TLB miss addresses that matches a reference mutation. This ratio is calculated using the following formula:

$$\frac{|R \cap^* T|}{|T|}$$

Where $R$ denotes the set of modified references page numbers, and $T$ is the set of TLB miss page numbers. Each similar address between $R$ and $T$ is multiplied by its number of occurrences in $T$. On the other hand, inverse coverage would measure the ratio of reference mutations that match a TLB miss. The formula for calculating the inverse coverage is:

$$\frac{|R \cap^* T|}{|R|}$$

And similar address between $R$ and $T$ is multiplied by its number of occurrences in $R$. Both $R$ and $T$ are taken from the trace file generated from simulating the SPECJVM98 benchmarks (trace file details can be found in Section 2.1). Figure 3.2 shows the coverage and inverse coverage percentage for individual benchmarks. The average coverage ratio is 36%, while the average inverse coverage reaches 80%.

The results in Figure 3.2 provides us with two observations. First, not all TLB miss addresses matches(covered by) reference mutations, with an average coverage ratio of
36% . However, the ratio is proportional to the percentage of TLB miss that occur in memory regions that stores application objects, mainly, the MS region (Table 2.5). Secondly, most reference mutations (average of 80%) matches a TLB miss address, which means that monitoring reference mutations can provide us with TLB miss addresses.

### 3.3 Reference Modification Patterns

To obtain a better insight into reference mutation patterns, write barrier events recorded into the simulator trace file are extracted and plotted. Figure 3.3 presents the write barrier events for the simulated benchmarks. The plots have similar properties and patterns to those in Figure 2.1 that represents TLB miss addresses, which confirms the observation from Figure 3.2 that reference mutation targets are related with TLB miss addresses.

### 3.4 Differences Between Consecutive Events

For all plots in Figure 2.1 and Figure 3.3, the horizontal scale is too small to clearly separate two consecutive points. This reduced view deprives us from discovering the
Figure 3.3 Write Barrier Addresses: X-axis represents the number of executed instructions, while the Y-axis shows the page number to which the target address belongs.
general trends in consecutive events values, such as whether the addresses tend to increase or decrease, and whether they change in small or large quantities. Figure 3.4 presents the CDF (Cumulative Distribution Function) plots for three of the simulated benchmarks. CDF plots give the cumulative probability of a certain event, for example, for a point \((x, y)\), the probability of having a value \(\leq x\) would be \(y\). In order to generate these plots, we have calculated the difference in page numbers between consecutive TLB misses and consecutive reference mutations that targets the MS heap region. For example, a difference of zero means that the consecutive misses or reference writes targets the same memory page.

Our first observation is that differences are symmetrical around zero, which means that the next event have an equal probability to occur in a higher or a lower address. The impact of this observation is that, predicting next events by going in a single direction (increase or decrease) will approximately miss half of the events. We also observe that the probability of having small offsets is higher than having large offsets between consecutive events. This is apparent from the nearly-vertical lines for offsets close to zero.

3.5 Concluding Remarks

The similarity between reference mutation and TLB miss patterns confirm the cause-effect relation noted at the beginning of this chapter. This relation implies that we can know about misses behavior by monitoring reference mutations. Furthermore, by predicting future reference mutations, we can predict future TLB misses. Several schemes were proposed to monitor TLB misses to predict future misses. However, none have evaluated the use of reference mutation instead of previous misses. In the next chapter, we will review some of the previously proposed scheme to predict TLB misses. Then we will evaluate the benefits of using previous reference mutations rather than previous misses for prediction.
Figure 3.4 Offset CDF: The X-axis represents offsets values. The Y-axis represents cumulative probability.
4 PREFETCHING MECHANISMS

Previously proposed hardware-based TLB preloading algorithms were designed to predict future TLB misses using gathered information from previous TLB misses. This hardware-only view does not take into account the application view about misses. As we have seen in the previous chapter, some aspects of the application behavior have strong correlation with TLB misses. Therefore, previously proposed hardware-based preloading schemes have two drawbacks: first, the application behavior is not part of their prediction algorithm. Secondly, the monitoring of previous misses requires hardware modifications, which may not be available to all architectures.

Here we propose a software-based TLB preloading algorithm. We have already discussed several aspects of the application behavior and their effect on TLB misses. Concluding that reference mutation patterns are similar to TLB miss patterns, which leads us to our proposal: to use reference mutation pattern to predict TLB miss pattern. In other words, instead of using previous TLB misses to predict future misses, we will use reference mutation information to predict TLB misses. Using reference mutations does not require any form of hardware support, as opposed to monitoring TLB misses needed by previously proposed hardware-based TLB preloading algorithms.

In this chapter, we will discuss recently proposed Hardware-based prediction models used to preload TLB entries. Java applications will be used to evaluate the prediction accuracy of those schemes. We will also discuss the reasons for their low accuracy in predicting TLB misses based on reference mutations. Finally, we propose our linear prediction model, and compare its accuracy to the previous models.
4.1 Existing Preloading Schemes

In the past, several schemes were proposed to reduce the effect of cache misses. One of the commonly investigated schemes is prefetching, where data items that will cause a cache miss in the future are loaded into the cache before their actual use-time. A thorough survey on prefetching techniques can be found in [33]. Influenced by prefetching schemes, few have proposed prefetching schemes that target TLB misses. To the best of our knowledge, only two schemes were recently proposed: Recency-based and Distance prefetching.

Recency-based prefetching [26] is based on monitoring the temporal locality of TLB misses. The model maintains an LRU (Last Recently Used) stack of misses by modifying the page table structure. Each page entry will contain two additional pointers, the first is to the page entry that caused the previous miss, the other is to the page entry that caused the next miss. The model will attempt to prefetch neighboring misses into a prediction buffer, which is a small 8-16 entries fully-associative cache. When a miss occur, its page entry will be moved to the top of the stack, and the next and previous pointers will be updated. This scheme is discussed here for completeness purposes.

Distance prefetching [21], on the other hand, assume that offsets (distances) between misses, rather than page addresses, shows high temporal locality. A special cache indexed by the difference between the current and the previous miss is used to provide a prediction for the next offset. The predicted offset is added to the current missed page number as a prediction for the next miss.

Both of the previously discussed schemes have a common feature: they require monitoring TLB misses from the hardware perspective. However, the application might have a different view from the hardware on what TLB entries should be prefetched. For example, TLB miss patterns for Java applications, as noted in Section 3, are a product of the application behavior.
The relation between TLB misses and the application behavior is the basis of our proposed preloading algorithm. In which we rely on monitoring reference mutations rather than TLB misses. However, before formalizing our prediction algorithm, we should analyze the predictability of reference mutation patterns compared to TLB reference patterns.

### 4.2 Events History: The Past is The Key to Future

The relation between previous events and the future can be quantified using a mathematical procedure called *Autocorrelation*. Basically, it will analyze a given set of events of length $N$, and correlate each point with a point that is $k$ observations away in the past. This process will be repeated for all possible $k$'s. Mathematically, the autocorrelation for a distance $k$ and a sequence $X$ is given by the following formula [34]:

$$A_k = \sum_{i=0}^{i=N-1} X_i X_{i+k}$$

and all subscripts are of modulo $N$. The value of $k$ is called the *lag*, and for each lag value, there is an correlation value. Figure 4.1 shows the autocorrelation plots for differences between consecutive reference writes and TLB misses.

In all plots in Figure 4.1, the correlation value decrease as we increase the lag value. This indicates that contribution from previous events decreases as we go further into the past. We also notice that in general reference mutation events are less correlated compared to TLB miss events. Lower correlation means that we can not predict future events using a small number of previous events. While on the other hand, TLB misses a better correlated and thus, a small set of previous events may suffice to predict future misses.

The plot for DB in Figure 4.2 has a distinctive pattern. The spikes are caused by the periodicity of the events shown in Figure 3.3, and the gap between spikes is equal to the number of events in each period.
Figure 4.1 Autocorrelation plots. X-axis represents the lag value, and the Y-axis shows the normalized correlation factor.
4.3 Proposal: Linear Prediction

Before discussing the details of our prediction algorithm, let us first consider the components needed for any prediction system. In order for any prediction system to work, several key components should be available. Those components will be responsible for processing the input to the prediction system, producing predictions, and evaluating predictions accuracy. Generally, we can think of any prediction system as having these basic components:

- **Input:** which is responsible for monitoring past events.
- **Processing:** which is responsible for processing monitored events to produce predictions of the future events.
- **Output:** responsible for loading the predictions into their target.

Our proposed prediction scheme will monitor the application behavior as it's input component. It will issue predictions based on an algorithm called *Linear Predictive Coding*. Here we will evaluate this prediction algorithm against previously proposed schemes, and show that it can achieve comparable accuracy rates.

4.3.1 The Linear Predictive Coding Model

Linear prediction is long known to the field of digital signal processing. It is mainly used to mathematically analyze a set of past events in order to predict the future. Several researchers used this algorithm to predict program phases [18, 27, 28] based on observations collected from various hardware units, such as cache misses and execution rate. In this chapter, we will propose an implementation to be used to predict TLB misses.

Linear prediction uses equations to relate future events with the previous ones. The main concept behind linear prediction, is that different previous events do not have equal
effect on future events. This effect is expressed in the form of correlation between the events. A high positive correlation value means that the two events increase or decrease in proportion, and a correlation value of zero represents the lack of any correlation between the events. Several correlation values from different points in the past can be used mathematically to calculate the future event value.

The linear prediction algorithm uses the autocorrelation procedure and a certain number of observations to construct a prediction polynomial \( P \), more details on the procedure used to construct this polynomial can found in [25]. This polynomial will have \( n \) coefficients, based on the user choice. When the polynomial is applied to a set of events, it will take the last \( n \) observations to predict the next event. This is illustrated in the following formula:

\[
X_i = \sum_{j=1}^{n} X_{i-j}P_j
\]  

Our prediction scheme uses the linear prediction algorithm to predict future TLB misses. The input to the algorithm would consist of reference write events as captured by the write barrier. In order to construct the prediction polynomial, we need to analyze a certain number of those events. The use of a high number of events can increase the polynomial accuracy, but on the expense of increased storage requirement and analysis time. In Section 5.2 we evaluate different set sizes. After the prediction polynomial is constructed, the predictor will start calculating future reference write events as described in Equation 4.2. The predictions are then loaded into the TLB in a scheme that will be described in Section 5.1.

### 4.4 Accuracy Evaluation

The accuracy of several variations of the linear predictor were compared to the distance predictor described previously. Those variations are generated by changing one or more of the following parameters:
- Initial history: the set size used to generate the prediction polynomial

- Polynomial degree

- Accuracy Threshold: when to recalculate the prediction polynomial.

While for the distance prefetcher, we use a 1024-entree prediction cache, where each entree supplies us with two predictions. The prediction models were evaluated by simulating them over TLB misses and reference mutation traces collected by DSS.

Another parameter that can be varied within our predictor is the use of offsets between reference mutations rather than the absolute values. However, in all of our experiments, we have found that the prediction accuracy when using offsets is much higher than using the absolute values. Predicting offsets means working with a smaller range of values. Using smaller values within the predictor reduces its prediction error. For this reason, we will use offsets for our predictor.

Figure 4.2 shows the effect of the polynomial size on accuracy. It is noted that polynomial size of more than four does not increase accuracy. The figure to the left shows the prediction accuracy for polynomials generated using a set of 512 entries, while the other shows it for a set of 1024 entries.
In figure 4.3, the accuracy of predicting misses using the linear predictor is compared to the distance prefetcher. The linear predictor uses reference mutations to predict misses, while the distance prefetcher will use previous TLB misses to predict future misses. It is worth to note that the distance prefetcher has a very low accuracy in predicting misses using reference mutation (less than 1% in all benchmarks). The reason is that distance prefetcher does not use sufficient previous events, and as noted in Figure 4.1, predicting reference mutation needs more previous events than predicting misses.
5 IMPLEMENTATION DETAILS AND PERFORMANCE EVALUATION

In this chapter, we will discuss several problems associated with TLB prefetching, Section 5.1 presents those problems and proposes a solution. We will also describe our prediction model implementation within Jikes RVM in Section 5.2. Finally, the performance of our proposed scheme is evaluated in Section 5.3.

5.1 TLB Prefetching

TLB prefetching models need to load the target with their prefetches. In the case of Hardware-based models, this is usually done by a specialized prefetch unit, and predictions are stored in a special buffer. However, in the case of software-based models, such facilities are simply unavailable. Most architectures does not provide any special instructions to handle the TLB as it provides to caches. Furthermore, page table entries are stored in the kernel space, therefore, the application can not access those entries directly. As a result, software-based prediction models need to provide their own schemes to load predictions into the TLB.

An intuitive approach to load TLB entries is to issue a load instruction that reads any address within the needed page, this would cause a TLB miss, and the translation mechanism will load the entree to the TLB. However, this solution will send us back to the main problem, since the TLB miss caused by the load instruction\(^1\) will slow down

\(^1\)The load will most likely cause a TLB miss, since it is accessing a page predicted to cause a miss!
the application.

One solution to overcome the previous problem is to use a helper thread. Helper threads were mainly proposed to help the application by prefetching cache lines in parallel with the application [23, 24, 35]. The advantage of using a helper thread is that TLB misses caused by the thread will not stall the application. However, inorder to get the full benefit from the helper thread, it should be executed in parallel with application, and most importantly, it should share the TLB unit with the application. Such architectures are available in the form of SMT (Simultaneous Multi Threading) processors [9], an example is Intel Hyper-Threading architecture [16]. SMT processors allow multiple threads to run concurrently on the same processor core, sharing the functional units and caches (including the TLB). By having a helper thread in the virtual machine on an SMT processor, the thread will load TLB predictions in parallel with the Java virtual machine execution. Since the helper thread and the application share the TLB unit, TLB preloaded entries will be available to the application when needed.

5.2 Implementation Details

We have modified Jikes RVM by adding a helper thread to the virtual machine. This thread will execute the prediction algorithm and issue TLB loads accordingly. We have also modified the write barrier to allow it to send reference write events to the helper thread. Furthermore, the helper thread will be notified on the start and end of each garbage collection phase. The helper thread is designed to not produce or issue predictions in the garbage collection phase, since the main focus of the algorithm is TLB misses caused by the application rather than the garbage collector.
Figure 5.1 Linear Predictor Performance. Relative to the base JVM

5.3 Results

Figure 5.1 shows the performance of the linear predictor. The implementation uses a polynomial size of four, and a 512 events history. The first column represents the performance of the predictor compared to the base virtual machine. The second shows the predictor overhead, where we perform all steps from reference mutation monitoring and polynomial generation, but instead of loading the predicted value, we issue a load to random page number. The last column shows the expected performance compared to overhead.

In most benchmarks we can see an average improvement of 3% compared to overhead. However, the overhead of this implementation limits the expected benefits. The negative performance gain for RayTrace can be attributed to its low prediction accuracy as seen in Figure 4.2. Generally, it is noted that the expected performance is correlated to accuracy rates. Figure 5.2 shows that as the accuracy increases, the expected performance increases accordingly.
Figure 5.2  Linear Predictor Accuracy vs. Performance. Each point represents a different benchmark.
6 RELATED WORK

Several studies have noted the negative impact of TLB misses on execution time [17, 20, 30]. Jacob et al. [17] evaluated the effect of different TLB design schemes on performance, such as block size or associativity. They also studied the effect of multi-level caches and TLB management policy.

TLB preloading schemes were only recently proposed in [21, 26], and discussed in [20]. Preloading schemes depends on feeding TLB misses to a special hardware unit in order to predict future misses. In [26], prediction is based on the observation that a sequence of TLB misses will be repeated in the same order at some point in time. It relies on a history table that stores the order of TLB misses. While in [21], they rely on the observation that offsets between misses, rather than the actual miss-addresses, repeat the same sequence. A special cache unit stores addressed by the last offset provides a prediction for the next TLB miss. Both of these prediction schemes relies on monitoring TLB misses via a special hardware unit, which limits the applicability of those schemes into prevalent architectures. Alternatively, our proposed scheme does not require knowledge about TLB miss events, but rather uses the running Java application behavior to predict TLB misses, thus eliminating the need for added logic to monitor misses.

Java application’s interaction with the underlying hardware was targeted by many studies. Several studies evaluated the role of automatic memory management on performance [10, 12] as opposed to explicit memory management techniques used in other languages such as C/C++. Concluding that automatic memory management requires
a larger memory footprint as opposed to explicit memory management. Others studied the memory behaviors for Java applications [22, 30], where in [30], the poor TLB performance was noted for Java applications. In [8], the interaction between Java application and the virtual machine and its effect on the underlaying hardware were analyzed. Generally, these studies agree on the observation that Java application suffer from poor memory performance compared to applications that uses explicit memory management policies.

Several papers aimed at enhancing performance by reducing the memory stall for Java applications. Different techniques were evaluated, such as the use of prefetching [1, 5, 15] to reduce the data cache miss penalty. Others investigated the use of different object layout schemes to increase locality [13, 29], thus reducing cache and TLB misses. However, to the best of our knowledge, TLB preloading for Java has not been proposed.
7 CONCLUSIONS

TLB miss effect on execution time is only going to increase with the widening speed gap between processors and the memory. The increase in applications memory requirement will also contribute to the problem by increasing the number of misses. Java applications with their large memory footprint can be within the first victims as the speed gap increases. Although our experiments show that TLB misses constitute a considerable 24% of java application's execution time, it is not unusual to see higher ratios in the near future. Furthermore, TLB misses can degrade the benefits from schemes that implement data cache prefetching.

We have studied the relation between the application behavior and TLB misses. Concluding that the application behavior represented by reference mutations shapes and affects TLB misses. The interesting outcome of this relation is that reference mutations can represent TLB misses. They can be monitored instead of TLB misses, and more importantly, predicting mutations can help us in predicting misses. Generally, this relation can be the basis for a scheme that uses application-based monitoring rather than hardware-based monitoring. The use of application-based monitoring makes enhancements portable, while on the other hand, hardware-based monitoring will restrict enhancements to specific architectures.

Based on the relation between the application behavior and TLB misses, we have proposed a software-based predictor. As apposed to hardware-based predictor, the proposed scheme did not require any hardware modification while achieving a comparable accuracy rate. Our proposed scheme relied on monitoring reference mutations as cap-
tured by the write barrier, which is already available to the virtual machine. This reduces monitoring overhead while maintaining accuracy.

Our actual implementation relied on a helper thread to provide predictions. The choice of using another thread is based on the lack of proper support to manage TLB entries from the application. However, as seen in Chapter 5, the high overhead of this implementation limits the expected benefits. A possible future research can be guided to investigate the usefulness of having special instruction to prefetch TLB entries as it is the case with regular caches.
BIBLIOGRAPHY
Bibliography


