BarTLB: Barren Page Resistant TLB for Managed Runtime Environments

Author: Xin Tong

Supervisor: Andreas Moshovos

A thesis submitted in fulfillment of the requirements for the degree of Master of Applied Science and Engineering in the
Computer Group
Department of Electrical and Computer Engineering

August 2014

© Copyright by Xin Tong (2014)
This thesis observes that many translation look-aside buffer (TLB) misses in managed runtime language workloads originate from barren pages. That is, pages that contain mostly dead objects sprinkled with only a few live objects. The main contribution of this work is that it characterizes the barren page phenomenon. This work also proposes (1) a low-cost barren page identification technique, and (2) two simple, low-cost techniques for improving TLB performance: (a) The Barren Page First (BPF) replacement policy extends an existing TLB replacement policy to prefer barren pages on evictions. (b) Selective In-Cache Translation Caching (SICTC) avoids installing barren pages in the TLB by augmenting one way of a virtually-indexed, physically-tagged L1 data cache with virtual tags. Experiments find that for those Dacapo and SPECJBB2005 workloads that exhibit barren pages BPF and SICTC improve performance by 4% and 9.2% on average.
Acknowledgements

I would like to acknowledge the help and support I have received from a number of sources while working on this thesis. First, I would like to thank adviser, Andreas Moshovos without whom I could not have completed this work. Second, I would like to thank all the members of our research group for their support and feedback throughout the different stages of this work. Finally, I would like to thank my parents and my brother for all their help and support while I was completing this work.
Contents

Abstract ii

Acknowledgements iii

List of Figures vi

List of Tables vii

1 Introduction 1
  1.1 Memory Management by Garbage Collection 1
  1.2 TLB and Garbage Collection 2
  1.3 Contributions 2
  1.4 Thesis Organization 3

2 Background 5
  2.1 Virtual Memory 5
  2.2 Translation Look-aside Buffer 6
  2.3 Java Virtual Machine 7
  2.4 Garbage Collection 7

3 Simulation Methodology 9
  3.1 Workloads 9
  3.2 Functional Simulation 10
  3.3 Cycle Accurate Simulation 11

4 Workload Analysis 13
  4.1 How Much does Java Suffer from TLB Misses 13
  4.2 Barren Page Behavior Analysis 14
    4.2.1 Allocation to Survival Ratio 15
    4.2.2 How Java Applications Create Barren Pages 15
    4.2.3 Unique Block Reference Analysis 16
    4.2.4 Solitary Object Analysis 17
    4.2.5 TLB Misses Due to Barren Pages 18
    4.2.6 Barren Page Working Set Analysis 19
    4.2.7 Analysis Summary 20
  4.3 Potential Software Solutions to Barren Pages 20
    4.3.1 Concurrent Garbage Collection 21
# List of Figures

4.1 Performance Improvement with HugePage Support. .................. 13
4.2 Memory survival rate (fraction of bytes allocated) over a garbage collection cycle. .................................................. 15
4.3 Number of unique blocks accessed per L1 TLB entry installation. .... 17
4.4 Block access and L1 TLB miss counts for the four pages with the highest TLB miss count for SPECJBB2005. ......................... 18
4.5 Unique blocks necessary to capture 90% of the misses per page and their contribution to the overall L1 TLB miss rate. ............... 19
4.6 Distribution of page count vs. TLB miss rate. ........................ 20
4.7 Parallel GC vs Concurrent GC Performance. .......................... 21
4.8 Performance by Varying Nursery Heap Size. ......................... 22

5.1 Barren Page Identification Example ................................. 25
5.2 L1 Number of references per L1 TLB entry. ......................... 26
5.3 Barren page identification accuracy. ................................. 27

6.1 SICTC cache organization. ........................................... 30
6.2 L1 TLB Miss Reduction Averaged Over Three Application Cycles. ...... 31
6.3 Overall Performance Improvement Over Base TLB Averaged Over Three Application Cycles. ............................................ 32
6.4 L1 Data Cache MPKI with SICTC Averaged Over 3 Application Cycles. 33
6.5 Overall Performance Improvement Over Base TLB with an Infinite L3 Averaged Over 3 Application Cycles. .............................. 34
List of Tables

3.1 Workloads and Instructions Simulated Per Workload .......................... 10
3.2 Simulated Base System Configuration .............................................. 11
Chapter 1

Introduction

1.1 Memory Management by Garbage Collection

The last decade has witnessed the widespread adoption of managed programming languages [1]. Their popularity stems, in part, from the automatic management of memory where a garbage collector reclaims memory from dead objects, that is data elements that can no longer be referenced by the program. Compared to explicit memory management, garbage collection results in faster memory allocation, fewer programming and memory management errors, shortened development time [2], and concurrent de-allocation operations when the garbage collector runs as a separate thread. Among the top ten languages in terms of popularity, six are managed programming languages and all rely on garbage collection [1]. In addition to being used in desktop and server environments, managed programming languages have now become widely used in mobile devices where performance and power are even more important.

Garbage collection (GC) relieves the programmers from having to explicitly manage memory de-allocation. Instead, GC attempts to reclaim garbage, or memory occupied by objects that are no longer in use by the program. Garbage can be identified iteratively through a scanning process or via reference counting. In the iterative method, GC identifies dead objects starting from a root set of objects which is known to be alive. It iteratively scans for objects that are reachable from this root set marking them as alive. Once all reachable objects have been visited the remaining unmarked objects are dead. With reference counting each object maintains a count of the active references to it by other objects. When new pointers are created to the object, the reference counter is incremented and when pointers are made to point to other things or die, the reference counter is decremented. Once the reference count reaches zero, the object is dead. GC is triggered automatically when the amount of free memory falls below a defined threshold.
Therefore, the programmers are freed from the burden of having to consider when and where to de-allocate memory for objects.

1.2 TLB and Garbage Collection

Unfortunately, the “allocate and forget” convenience that GC offers does not always interact favorably with the virtual memory system. Specifically, this work demonstrates that several Java applications thrash conventional Translation Look-aside Buffers (TLBs) since they experience many barren pages with only a few solitary objects that are alive and that the application actively references. This work finds that: (1) Applications create barren pages unknowingly since they allocate objects with no regard to their expected lifetime (generational garbage collectors classify objects according to lifetime but only at garbage collection time [3]). Since most objects prove short lived, they push the few remaining long lived objects far apart from each other. (2) The working set of barren pages often exceeds the capacity of conventional L1 TLBs. (3) Accesses to barren pages are bursty with long inter-burst reference intervals. Coupled with the large working set of barren pages, this access pattern translates into a TLB miss followed by very few TLB hits before the inevitable TLB eviction. Overall, barren pages unnecessarily thrash conventional TLBs that rely on temporal and spatial locality to be effective.

TLB misses can greatly affect performance today and will likely grow to be more detrimental in the future. Depending on the application, TLB misses have been shown to degrade overall system performance by 5% to 14% or by up to 50% [4]. The performance loss due to TLB misses will likely increase in the future since TLB misses will also most likely increase for two reasons: (1) application working sets tend to increase over time, and (2) virtualization, in which TLB misses require multiple, nested page table walks, is becoming more popular [4]. While larger TLBs would accommodate larger application footprints, unfortunately, TLB size is limited by processor clock speeds and power restrictions making larger TLBs unfeasible. Accordingly, it is desirable to develop simple, low-cost mechanisms for reducing the impact of barren pages in Java applications.

1.3 Contributions

This work proposes a practical, low-cost and programmer-transparent mechanism that dynamically identifies barren pages in the Java heap. The mechanism relies on sampling to quickly identify barren pages based how many L1 TLB hits they experience. To demonstrate that barren pages can be exploited to improve modern systems, this
work presents two barren page resistant TLB (BarTLB) mechanisms that improve performance. The Barren Page First Replacement (BPF) enhances TLB replacement decisions making barren pages the preferred eviction victims. Selective In-Cache Translation Caching (SICTC) acts as a TLB filter for barren pages banishing them from re-entering the TLB. Instead, SICTC confines barren pages to one way of a conventional virtually-indexed, physically-tagged (VIPT) data cache. For this purpose, SICTC extends that cache way with an additional virtual tag and permission bits so that virtual address translations take place in the cache.

In summary, the two key contributions of this work are that:

1. It identifies the barren page phenomenon, its root causes and characteristics, demonstrating that it is responsible for a many TLB misses in certain, important Java workloads. Furthermore, this work investigates existing software solutions to barren pages and concludes that while they may be able to mitigate performance penalty of barren pages, they come with other performance problems.

2. It proposes a low-cost mechanism for identifying barren pages.

In addition, this work demonstrates two simple mechanisms that exploit barren page behavior:

1. BPF, which enhances existing state-of-the-art replacement policies, including those that change the insertion policy, with barren page information. BPF improves performance by 4% on average over the commonly used LRU replacement policy for workloads that suffer from barren pages. Over all workloads studied, BPF proves robust and provides an average performance of 1.75%.

2. SICTC, a novel yet simple and low cost extension to a conventional VIPT cache that allows accesses to objects within barren pages to bypass the TLB. SICTC improves overall performance by 9.2% on average for those workloads that suffer most from barren pages. Over all workloads studies SICTC also proves robust offering an average performance boost of 4.6%.

1.4 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 reviews virtual memory, TLB operation, Java runtime systems and garbage collection concepts. Chapter 3 details the measurement methodology used in this work. Chapter 4 quantitatively measures
and proves the existence of barren pages. Chapter 5 proposes hardware optimization techniques to mitigate barren pages. Chapter 6 evaluates the performance potential of the proposed techniques, chapter 7 discusses related work, while Chapter 8 concludes this thesis.
Chapter 2

Background

The techniques presented in this work mitigate the detrimental effects of certain virtual memory pages on the Translation Look-aside Buffer’s ability to accelerate virtual memory operations when running Java applications. Accordingly, this section reviews key background concepts that relate to the analysis and techniques presented in this work. Specifically, Section 2.1 reviews virtual memory concepts. Section 2.2 discusses how Translation Look-aside Buffers accelerate virtual memory operation. Section 2.3 reviews Java Virtual Machine concepts, while Section 2.4 discusses Garbage Collection.

2.1 Virtual Memory

The virtual memory system provides each process with the illusion of having a private, contiguous address space. As a result, each process sees its own private virtual address space. These virtual address spaces are then mapped onto the actual memory that the machine provides, that is the physical address space. To implement this mapping and to accommodate many virtual address spaces that may exceed the total physical address space available, both virtual and physical address spaces are conceptually partitioned into equally sized, power of two sized, continuous chunks called pages. For the system studied in this work the most common page size is 8Kb. Each virtual memory page is then mapped onto a physical memory page. This virtual to physical page mapping is commonly implemented using a page table, that is a table whose index is the virtual page number index (the lower bits constituting the offset within the page are not used as part of this index) and its contents are the physical page number plus additional bookkeeping information such as permission information. Since the total capacity of all or sometimes even one virtual address space may exceed the amount of physical memory available, pages may be evicted and temporarily paged out, that is stored in secondary
storage. As physical memory capacities have grown and their cost decreased such paged out pages are increasingly less frequent.

Since the page table itself can grow quite large, page tables are often implemented as hierarchical structures where each level points to a sub-table mapping a portion of the virtual address space. This way only those virtual pages that each process allocates need to allocate space in the page table.

With virtual memory, each program reference, be it for fetching an instruction or for reading or writing data values, needs to first be translated from the virtual address (VA) it refers to, to the physical address (PA) this VA is currently mapped to. Accordingly, the page table must be accessed on every memory reference. Even with a flat page table organization, this address translation would require at least one extra memory reference. In reality, given that page tables are often hierarchical, multiple, extra memory references are required. Performing this extra memory references would unduly slow down program execution especially if one considers that memory references are much slower compared to other operations in modern processing cores. Accordingly, modern processors incorporate virtual memory translation acceleration mechanisms. The next section reviews the most commonly used mechanism which is also the focus of this work.

### 2.2 Translation Look-aside Buffer

To obviate the need to consult the page table of the process on every memory access, a hardware structure called Translation Look-aside Buffer (TLB) is used. TLB speeds up the process of translating virtual memory to physical memory by caching part of the process’ page table. If the translation for the requested address is present in the TLB, then there is a TLB hit, otherwise there is a TLB miss. A miss in the TLB triggers a page table walk which can take many cycles depending on how page table are structured. On X86 Linux, page tables are laid out in four different levels, therefore a page table walk will need at least four memory accesses. While a page walk is in progress, the affected memory access and its dependent instructions are effectively stalled. Thus, this can pose great performance penalties on the program. Because of the cost of page table walks, TLB misses are to be avoided as much as possible. To extend the reach of the TLB and to accommodate known to be large data structures such as the display frame buffer, modern operating systems usually offer superpages or hugepages, that is pages of larger sizes. With a larger page, a larger amount of memory can be translated by a single TLB entry. However, superpages are not free of trade offs as Section 7.1.2 explains.
The TLB is on the critical path of every memory access and therefore has tight access time constraints. As a result, the size of the TLB is usually very small in comparison to all the pages used by the program. Modern processors usually use multi-level TLBs. The first level TLB is usually small and very fast and the second level TLB is slower and it too used to reduce how often page table walks are needed. However, the L2 TLB is not as fast as the L1 TLB and thus it does impose a performance overhead albeit one that is a much shorter than a page walk. TLB can be organized either as a fully associative structure or multiple set-associative structures in order to support multiple page sizes [5] [6].

2.3 Java Virtual Machine

A Java Virtual Machine (JVM) is a virtual machine that can execute Java bytecode; Java applications are compiled down to Java bytecode. Because bytecodes are architecture independent, Java allows application programs to be built once and run on any platform without having to be rewritten or recompiled. Bytecodes can either be interpreted or JIT (just-in-time) compiled. Interpretation offers faster start-up time and better portability whereas JIT compilation offers much better performance. One distinct feature of the Java language relevant to this work is its employment of garbage collection as explained in Section 2.4.

2.4 Garbage Collection

Garbage collection (GC) is the process that aims to free up occupied memory that is no longer referenced by any reachable Java object, and is an essential part of the Java virtual machine’s (JVM’s) dynamic memory management system. In a typical garbage collection cycle all objects that are still referenced, and thus reachable, are kept. The space occupied by previously referenced objects is freed and reclaimed to enable new object allocation. Garbage collection usually requires two stages. In the first stage, the dead objects are identified by means of mark and sweep or reference counting. In the second stage, the memory for the dead objects is reclaimed and an optional compaction of the live objects are performed. Garbage collection is usually ran when the amount of free memory falls below a specific threshold, with the exception of concurrent garbage collection as explained in Section 4.3.1. Ideally, GC would be ran frequently without impacting overall performance. Running GC frequently would reduce the amount of memory needed by the heap, the memory space where dynamic object allocation takes place. However, the more frequently GC is ran, the more time taken away from the
application by the garbage collector. Accordingly, the frequency of GC is chosen in order to balance the need for keeping the heap size in check while avoiding an excessive performance penalty. As a result, GC does not run as frequently, e.g. once every second, creating relatively long periods of time where dead objects accumulate in the heap.

It is well known that Java objects die young [3]. As a result, generational garbage collection is built to scan certain areas more often than others. In generational garbage collection, the heap is divided into two areas or generations: the nursery or young space, and the old space. Initially, objects are allocated in the nursery space. If an object proves to be long lived, for example, if it survives size garbage collection cycles by default in OpenJDK, it is moved to the old space. By separating objects based on their lifetime, the time used for garbage collection can be used more effectively by running on the nursery heap where most dead objects exist and more free memory can be recovered.
Chapter 3

Simulation Methodology

This chapter details the experimental methodology used in this work to conduct the barren page phenomenon analysis and to estimate the potential of the proposed performance optimization techniques. This work first uses functional simulation to confirm the existence of barren pages as shown in Section 3.2. This work then uses cycle accurate simulation to measure the performance potential of the proposed techniques as shown in Section 3.3.

3.1 Workloads

This work uses the Dacapo [7] and SPECJBB2005 [8] workloads. The applications are ran over the OpenJDK 7 Java runtime system. Table 3.1 reports the input data sets and the number of instructions simulated per workload. Here, the term application cycle describes the execution sample between two garbage collection cycles. The first one billion instructions are skipped to avoid application start-up and traces are collected for three randomly selected application cycles. The traces are then simulated separately and the reduction in L1 TLB misses and the performance variation are reported averaged over the three samples. Where appropriate error bars show the range of the average measurement values per application cycle. The results of Chapter 4 showing the analysis of the barren page behavior are taken over one application sample. The results of Chapter 5 demonstrate the performance of the two barren page exploiting mechanisms and use all three application cycles. The results of this latter section confirm that barren pages persist over multiple application cycles.
### Table 3.1: Workloads and Instructions Simulated Per Workload.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>h2</td>
<td>large</td>
<td>564M</td>
<td>561M</td>
<td>564M</td>
</tr>
<tr>
<td>lusearch</td>
<td>large</td>
<td>301M</td>
<td>312M</td>
<td>317M</td>
</tr>
<tr>
<td>tradebeans</td>
<td>large</td>
<td>2,178M</td>
<td>2,281M</td>
<td>2,045M</td>
</tr>
<tr>
<td>tradesoap</td>
<td>large</td>
<td>1,123M</td>
<td>1,056M</td>
<td>1,086M</td>
</tr>
<tr>
<td>specjbb</td>
<td>1 out of 4 warehouses</td>
<td>2,121M</td>
<td>2,039M</td>
<td>2,271M</td>
</tr>
<tr>
<td>Others</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>avrora</td>
<td>large</td>
<td>542M</td>
<td>511M</td>
<td>539M</td>
</tr>
<tr>
<td>batik</td>
<td>large</td>
<td>613M</td>
<td>621M</td>
<td>639M</td>
</tr>
<tr>
<td>eclipse</td>
<td>large</td>
<td>564M</td>
<td>549M</td>
<td>542M</td>
</tr>
<tr>
<td>fop</td>
<td>default</td>
<td>561M</td>
<td>552M</td>
<td>499M</td>
</tr>
<tr>
<td>jython</td>
<td>large</td>
<td>545M</td>
<td>541M</td>
<td>545M</td>
</tr>
<tr>
<td>luindex</td>
<td>default</td>
<td>655M</td>
<td>661M</td>
<td>661M</td>
</tr>
<tr>
<td>pmd</td>
<td>large</td>
<td>493M</td>
<td>481M</td>
<td>478M</td>
</tr>
<tr>
<td>sunflow</td>
<td>large</td>
<td>289M</td>
<td>318M</td>
<td>321M</td>
</tr>
<tr>
<td>tomcat</td>
<td>large</td>
<td>585M</td>
<td>531M</td>
<td>552M</td>
</tr>
<tr>
<td>xalan</td>
<td>large</td>
<td>583M</td>
<td>511M</td>
<td>592M</td>
</tr>
</tbody>
</table>

### 3.2 Functional Simulation

Functional simulation is used to model the hit and miss behavior of the TLB and thus to identify and analyze the barren page phenomenon. Functional simulation models the effects of instructions and the hit or miss behavior of the memory hierarchy without detailed modeling of the timing of events. It is commonly used in architecture studies as it is at least an order of magnitude faster than timing simulation. For this reason, functional simulation enable the exploration of a larger portion of the design space. Since functional simulation does not model the exact timing of events it will not accurately model contention and its effect on program execution. However, the hope is that the trends observed will be correct. It is for this reason that at the end, timing simulation is used to confirm the validity of the observations made with functional simulation for a few relevant configurations.

The functional simulation experiments are used to confirm the existence of barren pages. The experiments use Intel’s PIN to trace the execution of the OpenJDK 7 running Dacapo and SPECJBB2005. As the amount of time spent in the kernel space in these workloads is very small, only user-level instructions are simulated. A two-level TLB is simulated and the relevant statistics are used to estimate the amount of barren pages existing in the system as discussed in Section 4.2. The simulated TLB closely resembles that of the Intel Sandy Bridge architecture [9] as it includes a 64-entry, 4-way set
associative L1 TLB, and a 512-entry, 4-way set associative L2 TLB both with LRU replacement. The experiments assume 4KB pages and 64-byte L1 cache blocks (each page contains 64 blocks). These are commonly used sizes in x86 systems.

The functional simulation also gathers traces which are then used in the cycle accurate simulation as shown in Section 3.3.

### 3.3 Cycle Accurate Simulation

<table>
<thead>
<tr>
<th>Processing Core</th>
<th>OoO, 2.5GHZ, 6-wide Fetch/Decode/Issue, 128-entry ROB, 80-entry LSQ, BTAC (4-way, 512-entry), TAGE (5-tables, 512-entry, 2K-bimod)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 ICache</td>
<td>64KB, 64B cache blocks, 8-way, 3-cycle load-to-use</td>
</tr>
<tr>
<td>L1 DCache</td>
<td>64KB, 64B cache blocks, 8-way, 3-cycle load-to-use</td>
</tr>
<tr>
<td>Unified L2 Cache</td>
<td>512KB, 64B cache blocks, 8-way, 16-cycle load-to-use</td>
</tr>
<tr>
<td>L1 ITLB</td>
<td>64-entry, 4-way, 3-cycle load to use.</td>
</tr>
<tr>
<td>L1 DTLB</td>
<td>64-entry, 4-way, 3-cycle load to use.</td>
</tr>
<tr>
<td>L2 TLB</td>
<td>512-entry, 4-way, 5-cycle hit latency.</td>
</tr>
<tr>
<td>Main Memory</td>
<td>DDR3 1.6GHz, 800MHz Bus, 100ns latency 2 Channels / 1 Rank / 8 Banks 8B Bus Width, Open Page Policy</td>
</tr>
<tr>
<td>Optional Unified L3 Cache</td>
<td>Infinite size, 40-cycle load-to-use.</td>
</tr>
</tbody>
</table>

Table 3.2: Simulated Base System Configuration

Trace-based cycle-accurate simulation on a modified version of the Zesto x86 architecture simulator [10] is used to measure the performance effects of barren pages and of the optimizations that this thesis proposes. Table 3.2 details the baseline architecture which uses a hardware page walker. The traces were collected over two garbage collection cycles and the total number of simulated instructions is shown in Table 3.2. As per the results of Sections 6.3 and 6.4, the workloads are classified in (1) those that barren page optimizations improve (Barren Page Sensitive, or Sensitive) and (2) those do not (Others). The former group includes applications that are meant to be representative of commercial, transaction-like workloads. The techniques proposed in this work should improve performance for the sensitive group. They should also prove robust by not hurting performance for the insensitive group. The experiments demonstrate that the technique proposed meet both requirements improving performance considerably
for the sensitive group of applications while not only being robust but also improving performance for the insensitive group as well.
Chapter 4

Workload Analysis

This section analyzes several Java workloads showing that (1) TLB misses can greatly impact performance, and (2) a significant fraction of TLB misses are due to barren pages. It then analyzes the behavior and characteristics of barren pages explaining why existing software solutions are challenged by these pages. Specifically, Section 4.1 uses measurements on an actual system to show that Java applications suffer significantly from TLB misses. Section 4.2 analyzes barren pages using functional simulation as discussed in Section 3.2. Section 4.3 and its subsections discuss potential software solutions to barren pages.

4.1 How Much does Java Suffer from TLB Misses

![Performance Improvement with HugePage Support](image)

*Figure 4.1: Performance Improvement with HugePage Support.*
Chapter 4. Workload Analysis

If barren pages are indeed problematic for the TLB, it must be that the TLB misses do hurt performance for Java workloads. Accordingly, to measure the extent Java workloads suffer from TLB misses, we measure the performance of the Dacapo [7] and SPECJBB2005 [8] with and without HugePage enabled on Linux. Using HugePage significantly reduces the number of TLB misses and thus focusing HugePage use provides a rough low bound of the time overhead spend in TLB misses in the default configuration where most pages are 4KB. All workloads are ran from start to finish. As shown in Figure 4.1, there is a performance improvement of 12% with HugePage enabled for the Java heap. This shows that there is at least a 12% performance penalty due to TLB misses in the Java programs, given that only the heaps are backed by huge pages and there are a small number of TLB misses on huge pages. Unfortunately, indiscriminately using HugePages is not free of trade offs as will be explained in Section 7.1.2.

4.2 Barren Page Behavior Analysis

It is well known that most Java objects die young e.g., [11]. One key reason why this happens is that many objects are created and used temporarily inside methods. As soon as the method exits, these objects die but are left littering memory until the next GC invocation reclaims them. Such objects affect the degree of spatial locality the application exhibits as they separate objects that remain alive; what otherwise could be references that would fall onto the same page, now become references to objects forced onto different pages due to several, intervening dead objects. As a result, this reduces the effectiveness of TLB entries which now observe fewer references while residing in the TLB.

To quantify the impact barren pages have on TLB performance for Java applications, this section runs workloads on a simulated a two-level TLB as detailed in Section 3.2. This section and the rest of this work focuses on the L1 data TLB behavior. The L2 TLB is much larger and the the effect of barren pages on the L2 TLB is much less noticeable for the specific workloads.

The rest of this section is organized as follows: Section 4.2.1 corroborates that most Java objects die quickly by measuring the number of bytes allocated and survived over a garbage collection cycle. To provide some intuition as to why many of the objects die young, Section 4.2.2 shows an example method from the Dacapo tradebeans workload that allocates shortly lived objects and thus results in barren pages. Section 4.2.3 shows that many TLB misses are due to pages that have only a few blocks accessed every time their entry is brought into the TLB. Section 4.2.4 further demonstrates that these pages have only a few live objects in them and thus are barren. Section 4.2.6 analyzes the
working set of barren pages and shows that barren pages are actively accessed thrashing the TLB. Finally, Section 4.3 discusses software solutions for mitigating the effect of barren pages and explains their shortcomings.

### 4.2.1 Allocation to Survival Ratio

Figure 4.2 shows the number of bytes allocated vs. the number of bytes survived on the Java heap between two garbage collection cycles. Only 8.9% of the allocated memory survives on average.

![Figure 4.2: Memory survival rate (fraction of bytes allocated) over a garbage collection cycle.](image)

While there are few live objects at any given point of time, it is possible that these objects are collocated in a few normal pages. Sections 4.2.3 through 4.2.6 show that this is not true for certain workloads. Instead, many of these are solitary objects inside barren pages. Before discussing these results, the next section shows an example illustrating how Java applications end up with barren pages.

### 4.2.2 How Java Applications Create Barren Pages

Listing 4.1 shows one of the methods from the tradebeans workload. This method allocates a large number of objects. Some of the allocated objects, i.e., conn, stmt, and rs, and objects allocated in getStatement() and executeQuery() are local to the getOrders() method and die upon exit from the method. On the contrary, the objects allocated and stored in the orderDataBeans structure remain alive and are used extensively in the rest of the execution. The objects are allocated in execution order which is oblivious to whether objects are short- or long-lived. As a result, long-lived objects may be
allocated in between short-lived objects. These become solitary objects that live inside barren page, which may only get compacted after a GC cycle. Many other methods in tradebeans exhibit a similar allocation pattern. In particular, methods that retrieve data from the underlying database create transaction statements and database connections objects that prove short-lived. Given this observation, it is likely that the barren page phenomenon is not specific to Java but it may be inherent to many other object-oriented languages that use GC.

```java
public Collection getOrders(String userID) {
    Collection orderDataBeans = new ArrayList();
    Connection conn = getConn();
    PreparedStatement stmt = getStatement(conn, ...);

    stmt.setString(1, userID);
    ResultSet rs = stmt.executeQuery();

    // return top 5 orders for now.
    int i = 0;
    while ((rs.next()) && (i++ < 5)) {
        OrderDataBean orderData = new OrderDataBean(...);
        orderDataBeans.add(orderData);
    }

    // close statement and connection to the database.
    // all allocated objects for the database connection
    // and the statement become dead at this point.
    stmt.close();
    commit(conn);

    return orderDataBeans;
}
```

Listing 4.1: Object survival and death example.

4.2.3 Unique Block Reference Analysis

Barren pages are pages that have few live objects. Thus, when accessing a barren page there should few unique blocks within it that are touched. Accordingly, Figure 4.3 shows the number of unique cache blocks that are accessed in each page while the corresponding entry remains TLB resident. The results are weighted according to the number of L1 TLB misses each entry is responsible for. On average about 15% of the TLB misses are
for pages that experience accesses to just one of their blocks. This percentage is much higher at approximately 23%, 22%, 18% and 20% for h2, tradebeans, tradesoap, and SPECJBB2005 respectively. Similarly, over all workloads 50% of the TLB misses are to pages that have at most 12 unique blocks (less than 1/5 of the page) accessed each time, with the fraction being at 65%, 65%, 61%, and 71% respectively for the aforementioned workloads. In summary, a large fraction of TLB misses is due to pages that have only a few blocks accessed every time their TLB entries are installed.

![Figure 4.3: Number of unique blocks accessed per L1 TLB entry installation.](image)

### 4.2.4 Solitary Object Analysis

To provide additional insight on solitary page behavior, this section takes a more detailed look at the block-level access frequency for the four most frequently missing in the TLB pages from SPECJBB2005. These pages account for 12.3% of all the L1 TLB misses. Specifically, these measurements show that it is only very few specific blocks within each page that are actively accessed. Given that this section looks at only a fraction of the overall pages, the results cannot be used to draw the conclusion that other pages behave in this way. However, they do provide additional evidence that for certain frequently accessed pages, not only few unique blocks are touched every time the page’s translation is brought into the TLB, but also that these are the same blocks. This result suggests that the remaining blocks are indeed dead. Reporting the same results for all pages or even all barren pages is impractical as they are too many.
Figure 4.4 reports the number of accesses (squares) and the number of TLB misses (diamonds) for each block (0 through 63) in the page. The results show that very few blocks in each page – at most six – are being actively accessed. In addition, many of the blocks have a TLB miss to access ratio that is close to one; every time such a block is accessed it results in a TLB miss and no other block gets accessed before the TLB entry is evicted. The graph also marks (circles) those blocks that are alive just before the next garbage collection cycle. With a few exceptions, the blocks that are not accessed are also dead. These measurements demonstrate that the few blocks that are actively accessed are solitary blocks.

**Figure 4.4:** Block access and L1 TLB miss counts for the four pages with the highest TLB miss count for SPECJBB2005.

### 4.2.5 TLB Misses Due to Barren Pages

It is difficult to report a single result that clearly identifies the fraction of all TLB misses that is due to barren pages across all pages. The key difficulty is that a page becomes barren only after most of its objects die. All objects get accessed initially and they die at different points in time. The next experiment indirectly identifies such pages by measuring the frequency of TLB misses per block per page. Blocks that are rarely accessed and rarely cause TLB misses are considered *practically* dead. Pages where most blocks are practically dead are most likely barren pages. Accordingly, Figure 4.5 ranks pages along the x-axis according to the number of unique blocks that are responsible for at least 90% of the TLB misses for this page. Pages with few such objects are most likely barren. The y-axis shows the cumulative distribution of the total number of TLB misses.
the corresponding blocks are responsible for. For example, for SPECJBB2005 37.5% of all TLB misses are from pages where just up to five blocks cause 90% of the TLB misses per page. Pages where just a few blocks account for the vast majority of the TLB misses are most likely barren pages where most of the objects are practically dead (meaning they are dead or rarely accessed) surrounding a few solitary objects. h2, tradesoap, lusearch, tradebeans and SPECJBB2005 exhibit a large fraction of such pages. In summary, these results hint that many misses are due to barren pages. The next experiment investigates the number of barren pages and its impact on TLB performance.

![Figure 4.5: Unique blocks necessary to capture 90% of the misses per page and their contribution to the overall L1 TLB miss rate.](image)

### 4.2.6 Barren Page Working Set Analysis

Figure 4.6 shows the distribution of TLB misses over the number of pages that cause them demonstrating two clear application groups. In applications that suffer from barren pages, there are relatively few barren pages that cause most of the misses. However, even at 256 pages, there is at least 20% to 33% of the TLB misses that are caused by less frequently missing pages. Coupled with the previous experiments, this experiment demonstrates that while there are often relatively few barren pages, they thrash the TLB and contribute disproportionally to the overall TLB miss rate.
4.2.7 Analysis Summary

This section demonstrated that a large fraction of TLB misses in Java applications is due to pages with severe internal fragmentation. Accesses to these pages thrash a conventional TLB. While their TLB entry is resident, these pages experience very few accesses to very few of their blocks. This behavior is at odds with conventional TLBs that rely on locality to remain effective. Accordingly, identifying the barren pages can help improve virtual memory system performance. The next chapter presents a practical mechanism for identifying barren pages at run-time. However, before this hardware mechanism is presented, the rest of this chapter discusses potential software solutions. A software solution is desirable as it can be readily deployed without needing to change the hardware. However, as explained next, effective software solutions are challenging. Moreover, while changing software is in principle easier, in practice it may not be practically possible due to the dynamics of software development and scale.

4.3 Potential Software Solutions to Barren Pages

Existing garbage collection techniques can be used to mitigate the effect of barren pages on TLB performance. However, they all suffer from other drawbacks this section explains.
4.3.1 Concurrent Garbage Collection

Garbage collection could run *periodically* or *concurrently* with the application. A periodic GC runs when the amount of available heap memory falls below a certain threshold. A concurrent GC runs in parallel with the application. Periodic GCs completely pause application execution, thus guaranteeing that new objects are not allocated and objects do not suddenly become unreachable while the GC is running. They are simpler and faster than concurrent GCs. Concurrent GCs induce shorter application pauses by splitting each GC pass into several phases, with program execution permitted between and sometimes within each phase. Specifically, concurrent GCs pause program execution only while scanning the program’s execution stack. Concurrent GC can reclaim dead objects as the application is running. Therefore, a concurrent GC may partially mitigate the effects of barren pages. Unfortunately, concurrent GCs are slower than periodic GCs for the following reasons: (1) The sum of the concurrent GC phases usually takes longer to complete than one periodic GC pass. (2) Synchronization is required between the application and a concurrent GC. (3) Concurrent GCs thrash the cache more often than periodic GCs; Either GC fetches lots of data that is not re-referenced, but concurrent GCs run more frequently [12].

This work used OpenJDK configured with the default periodic GC that proves to be simpler and higher performing than existing alternatives. Figure 4.7 shows the performance of the programs using the concurrent garbage collector vs. the parallel stop-the-world garbage collector. On average, there is a performance degradation of 5% using the concurrent garbage collector. The highest performance degradation of 18% is observed for specjbb2005.

![Figure 4.7: Parallel GC vs Concurrent GC Performance.](image-url)
4.3.2 Control the Size of Nursery Heap

Previous work has shown that TLB misses can be reduced in some cases with smaller nursery heap sizes [13]. However, having a smaller nursery heap size usually results in more frequent garbage collections.

Every garbage collection needs to scan the entire nursery heap and the everything in the card table. A card table is a data structure that the generational garbage collector uses to enumerate objects in the old generations that refer to objects in the nursery space so that it does not need to scan the entire old generation. This may involve pulling in megabytes of uncached data from the nursery and old generation and it makes garbage collection a non-trivial operation. Additionally, garbage collection is detrimental to application performance as it pulls in a large amount of data that have little re-use and therefore pollute the cache and the TLB.

Figure 4.8 shows how performance varies with different nursery heap sizes. On average, limiting the nursery heap to 8MB reduces performance by 9% compared to using a nursery heap size of 256MB. Increasing the nursery size above 256MB did not improve performance further.

![Figure 4.8: Performance by Varying Nursery Heap Size.](image)

4.4 Summary

This section has shown that TLB misses account for an average 12% performance penalty and that a significant fraction of TLB misses are due to barren pages. Furthermore, even though existing software solutions could mitigate the effects of barren pages, they can hurt performance due to other overheads.
Chapter 5

Identifying Barren Pages

This section presents the proposed mechanism for identifying barren pages on-the-fly. It also shows that despite being simple and low cost the identification mechanism is accurate. Specifically, Section 5.1 details the identification mechanism and Section 5.2 measures its accuracy.

5.1 Barren Page Identification

Before applying any optimizations, barren pages have to be accurately identified. A straightforward way to identify barren pages would be to identify how many unique blocks are touched while the corresponding TLB entry is resident. As the results of Section 4.2 suggest, barren pages experience a few accesses to only a few blocks every time they become TLB-resident. A straightforward barren-page identification mechanism would be to extend every TLB entry with a bit vector with one bit per block marking which blocks are touched. Upon eviction, the mechanism could then check whether the blocks touched are below some pre-specified threshold and thus whether the page was barren. An extra barren bit in the L2 TLB, and if so desired in the page table, could mark barren pages and can be used to predict that the page will be barren next time it is touched. A page’s barren bit can be cleared when the page is de-allocated and it can be updated by the barren page identification mechanism next time the page is brought in. While simple this mechanism would require an extra 64-bit field per TLB entry, a considerable overhead for a relatively small and power sensitive structure.

This work proposes a simpler, lower overhead mechanism that relies on the results of Section 4.2 to simplify the identification mechanism in two ways. Section 4.2 demonstrated that when a page is barren, not only a few of its blocks are touched, but also
these blocks get referenced only a few times. Accordingly, the first simplification is to track how many times each TLB entry is touched instead of which blocks exactly are touched. As long as the reference count remains below a threshold, the corresponding page is likely barren. Section 5.1.1 explains how to set this threshold. This simplification reduces the overhead per TLB entry.

The second simplification is to rely on sampling to identify the barren pages one at a time. Sampling will quickly identify the barren pages since Section 4.2 demonstrated that (1) barren pages occur frequently, (2) they get few references and thus are short lived in the TLB, and (3) they are relatively few. Accordingly, instead of using a separate counter per TLB entry, the mechanism uses a single tagged counter for the whole TLB.

The counter’s tag identifies the page that is being sampled at any given point of time. In the beginning, the counter is empty. When the first TLB entry is brought in, it reserves the counter by setting the tag to its virtual page number. The counter counts the number of references the page receives until, at some point, either (1) the counter exceeds a pre-specified threshold and thus the page is classified as normal, or (2) the TLB entry has to be evicted. In the latter case, the page got only a few TLB hits, and hence it is classified as barren. In either case, the counter is released and acquired by the next page to enter the TLB. This way, a single counter is used to incrementally profile the behavior of the program’s pages. For the applications studied in this work, a 5-bit counter coupled with a 36-bit virtual address tag were sufficient.

Figure 5.1 shows an example of the barren page identification process. Assume that virtual pages (VA)X,Y,Z and W arrive in that order at different times at the TLB. Each reserves the profiling counter on entry to the TLB. VAX, VAY and VAW remain in the L1 TLB long enough for the profiling counter to saturate, therefore, they are identified as normal pages and the profiling counter is released at the point of saturation. However, VAZ experiences fewer than 32 accesses before being evicted from the L1 TLB. As a result, it is marked as barren page.

Barren bits can be stored in the L2 TLB and in the page table. An alternative to propagating barren bits to the pages table is to store them only in the L2 TLB. In this case, the page table of the program needs not be modified and thus barren page optimizations can operate completely transparent to the operating system. However, when an entry is evicted from the L2 TLB, the barren information is lost and the page has to be re-identified as such. Section 6.5 demonstrates that the loss in accuracy is negligible for the studied workloads. The advantage of this method is that no changes are needed to the page table and the operating system. A third alternative, would use a dedicated, hash table of barren bits.
This work targets the Java heap since it contains most the barren pages and since one of the example optimizations relies on the absence of homonyms and synonyms in the Java heap. Accordingly, the mechanism uses two address registers identifying the beginning and the end of the Java heap. The Java runtime keeps this range updated at all times. In case of dis-contiguous Java heaps, additional registers can be used to identify them separately.

5.1.1 Setting the Threshold

The barren page identification mechanism relies on a threshold $\epsilon$. With too small a threshold many barren pages would sneak by, while, with too large a threshold many innocent pages would be falsely flagged. For this work using a static threshold proved sufficient. As Figure 5.2 shows, for those workloads where barren pages are most problematic, a fixed threshold of 31 would capture most of the relevant pages. That is close to 70% of the overall TLB misses for tradebeans, h2, lusearch, tradesoap and SPECJBB2005. The figure shows the number of references each page receives every time
it becomes TLB resident and the corresponding fraction of overall TLB misses the pages are responsible for. The identification mechanism uses this as a proxy for the number of unique block touched which Figure 4.3 reported.

As the next section demonstrates experimentally, for this work there was no need to consider a mechanism for dynamically adjusting the threshold. However, the runtime system can adjust the threshold as needed. While deploying JVMs on new systems, developers often have to tune several virtual machine parameters so that it can deliver the most throughput on the underlying system. An appropriate barren page identification threshold can be set in this process.

![Figure 5.2: L1 Number of references per L1 TLB entry.](image)

### 5.2 Identification Accuracy

It is important that pages are not incorrectly identified as barren pages and vice versa. A mis-identification of a barren page, i.e. a barren page identified as a normal page, would reduce the efficiency of the proposed techniques. A mis-identification of a normal page, i.e., a normal page identified as a barren page, has much worse consequences for the proposed hardware optimizations, i.e. the translation for the normal page will be constantly preferred for replacement even though it may contain many live objects in case of BFP and all cache blocks on the mis-identified normal pages will be competing for the precious virtual address augmented cache line in SICTC.
Figure 5.3 shows a breakdown of the predictions produced by the fixed threshold identification mechanism. For each workload and for each TLB miss there are four possibilities depending on whether: (1) the page is barren or not (B or N), and (2) the page is predicted to be barren or not. The four categories are labeled as "actual-predicted". A perfect identification mechanism would only produce B-B and N-N outcomes. The results show that the simple, static threshold mechanism which uses a single register to sample and identify barren pages rarely misidentifies pages.

5.3 Summary

This section has presented a barren-page identification mechanism that works by measuring the number of hits a TLB entry received before being evicted. As long as this number is below a statically chosen threshold, the page can be safely identified as barren. A simple, one page at a time sampling technique was proposed to detect barren pages with little additional hardware cost. Experiments confirmed that the proposed technique misses only a small amount of barren page opportunities and rarely mis-identifies normal pages as barren pages.
Chapter 6

Exploiting Barren Page Detection

This section uses cycle accurate simulation as detailed in Section 3.3 to evaluate two optimizations that reduce the performance impact of barren pages on the L1 TLB. The first optimization (BPF) enhances the replacement policy of a conventional TLB, whereas the second (SICTC) avoids installing barren pages in the TLB altogether. Specifically, Sections 6.1 and 6.2 detail the two barren page optimization techniques. Section 6.3 evaluates the effectiveness of the BPF replacement policy enhancement over several, state-of-the-art replacement policies. Section 6.4 studies the performance of SICTC. Sections 6.5 and 6.6 show that performance improvements persist when barren bits are not propagated to the page table and when using a deeper memory hierarchy respectively.

6.1 Barren-Page-First Replacement Policy (BPF)

Barren page access behavior is at odds with conventional TLBs that rely on spatial and temporal locality to be effective. Barren pages exhibit little of both. The Barren Page First replacement policy (BPF) enhances an underlying replacement policy by preferring barren pages as eviction victims. With BPF, if there is a barren page in the target set, that barren page is replaced first, otherwise, the underlying replacement policy is used. If there are multiple barren pages in the target set, one is randomly chosen as the victim.

Barren pages experience, short, bursty accesses thus the little spatial and temporal locality they exhibit can be exploited even if they stay in the TLB for a very short time. This improves the chance that other pages will get hits. BPF can be implemented as a layered extension over an underlying replacement policy and requires only one additional bit in each TLB entry to identify a barren page.
Section 6.3 investigates the interaction of BPF with other, recently proposed replacement policies for caches including those that use different insertion policies, e.g., [12, 13]. These policies were originally proposed for data caches and are scan and/or thrash resistant. Existing insertion policies, as proposed, rely on aggregate measurements over many blocks or some other non-block-specific measurement for their decisions. This is appropriate for the data caches that experience many long reference sequences with similar behavior. BPF adopts the key concept of intelligent insertion decisions and enhances the performance of existing policies by providing hints on a per page basis.

### 6.2 Selective, In-Cache Translation Caching

Prevention is often the best cure, so instead of having barren pages wreak havoc in the TLB, a filter may prevent them from re-entering the TLB. Barren pages could instead use another structure for caching their translations. As Figure 6.1 shows, in Selective In-Cache Translation Caching (SICTC), one way (out of eight in the studied configuration) of a virtually-indexed, physically-tagged L1 data cache is extended with a virtual address and an extra valid bit that complement the existing physical tag and valid bit. When a TLB miss happens on a barren page, the translation is brought in from the L2 TLB or the page table. However, instead of installing the translation in the L1 TLB, the translation installed in the cache along with the block. The block now contains both the virtual and physical tag for the block. Subsequent accesses to the same block will miss in the TLB but will find the translation in the cache itself. If another block from the page is accessed, the same process is followed and the block and its address translation are installed in the extended way of the cache. SICTC improves performance in two ways: (1) It hides barren pages from the TLB preventing them from thrashing it. (2) It confines solitary blocks to one way of the cache preventing them from competing with all other blocks. Blocks from normal pages can install in any of the ways including the extended one. When a normal block installs in the extended way it marks the virtual address tag as invalid. However, accesses to different blocks in a barren page will now experience a TLB miss per unique block. However, as it has been shown, barren pages have very few active blocks.

The physical tag is still needed for cache coherence. In case of TLB shootdowns, the virtual tags in the extended cache set are invalidated such that accesses to those blocks will not produce a virtual address match and will force the access to consult the L2 TLB or the page table. The blocks remain valid and hence there is no need to forcibly writeback any dirty blocks. Since barren identification is currently restricted to Java heap pages there is no need to install permission bits in the extended way; by default,
the Java heap has read and write permissions only. If necessary, the permission bits can be copied from the page table or the L2 TLB at installation time. Similarly, if the architecture supports address space identifiers, these would have to be copied as well.

SICTC is a hybrid virtually-indexed, physically-tagged and partially virtually-tagged cache. Virtually-tagged caches have been considered in the past, however, they have to contend with synonyms and homonyms. A synonym occurs when multiple virtual addresses map to the same physical address, whereas a homonym occurs when the same virtual address maps to different physical pages. The workloads studied exhibit neither synonyms nor homonyms in the Java heap. If necessary the mechanism can be disabled when synonyms or homonyms may occur.

![SICTC cache organization](image)

**Figure 6.1:** SICTC cache organization.

### 6.3 BPF Replacement Policy Enhancement

This work compares the performance of the BPF replacement policy enhancement with the following state-of-the-art replacement policies: least recently used (LRU), least frequently used (LFU), not recently used (NRU), LRU+LFU (LRFU) [14], least recently used insertion policy (LIP), bimodal insertion policy (BIP), static re-reference interval prediction hit-priority (SRRIP-HP or simply SRRIP), bimodal re-reference interval prediction (BRRIP), and the dynamic re-reference interval prediction insertion policy (DRRIP) [15] [16]. These policies have been demonstrated for caches and this work adapts them for the TLB.
Figure 6.2: L1 TLB Miss Reduction Averaged Over Three Application Cycles.

The policies are as follows:

- **LRU** replaces the page which has not been referenced for the longest time. It is susceptible to thrashing.

- **LFU** counts how often each page is accessed. Those that are accessed least often are discarded first. LFU is resistant to thrashing, but it does not take recency into account.

- **NRU** evicts pages that have not been recently used. It uses a reference bit per entry which is set after every reference and cleared periodically. NRU prefers to replace pages with cleared bits. Its primary advantages are that it is simple and inexpensive.

- **BIP** uses a probability function to chose between the MRU and LRU positions.

- **SRRIP** improves NRU by enhancing the granularity of re-reference prediction used for each cache block. Instead of inserting a block always at a near-immediate or distant re-reference, SRRIP inserts a block at an intermediate re-reference and promotes it to near-immediate upon re-reference. This allows blocks brought in by scanning to be evicted quickly.

- **BRRIP** Bimodal RRIP (BRRIP) tries to ameliorate thrashing by inserting the majority of cache blocks with a distant re-reference interval prediction while, it infrequently (with low probability) inserts new cache blocks with a long re-reference interval prediction.

- **DRRIP** uses set-dueling to dynamically select among SRRIP and BRRIP. DRRIP has been proven to be both scan and thrash resistant.
Chapter 6. Exploiting Barren Page Detection

The four leftmost bars of Figure 6.2 show the L1 TLB MPKI (misses per 1000 instructions) for several replacement policies including policies enhanced with BPF (marked as “policy+BPF”). The figure presents a subset of the policies studied for clarity. The policies that are omitted performed worse than SSRIP and DRRIP and their BPF-enhanced versions. The figure includes additional techniques which will be explained in subsequent sections. The discussion in this section focuses on the BPF mechanism that stores the barren bit information in the page table. Section 6.5 shows that the proposed techniques remain effective even when the barren bits are stored only in the L2 TLB and thus forgotten on eviction from the L2 TLB.

As expected, the state-of-the-art replacement policies that modify the insertion policy improve MPKI over the baseline LRU policy. The higher the baseline MPKI the higher the improvements. Enhancing the replacement policies with BPF improves MPKI further. Enhancing the baseline LRU with BPF results in the best MPKI on the average and for clarity the figure omits the other BPF enhanced policies. However, for certain workloads such as h2 and batik, SRRIP with BPF performs best. For batik LRU+BPF is worse than all insertion-based policies. However, the MPKI is low for this workload to start with and thus TLB performance barely affects overall performance. For the barren page ameliorated workloads LRU+BPF greatly reduce L1 TLB MPKI. While the alternate insertion policies are scan and thrash resistant their decisions are oblivious to individual entry behavior. BPF enhances their ability to insert pages at the right spot by providing relevant, page-specific information.

Figure 6.3 reports overall performance relative to the baseline system that uses LRU replacement in the L1 TLB for the various replacement policies. For the sensitive
workloads LRU+BPF improves performance by 4% on average. On average over all workloads, LRU+BPF proves robust and improves performance by 1.75%. Given the simplicity and the minuscule cost of BPF, these performance improvements can be justified.

### 6.4 Selective In-Cache Translation Caching

This section evaluates the performance of SICTC. Figure 6.2 includes the reduction in L1 TLB MPKI with SICTC. SICTC improves L1 TLB performance more than the replacement policies as it prevents correctly identified barren pages from thrashing the TLB. As hoped, the reduction in L1 TLB MPKI is more pronounced for the applications that suffer most from barren pages. Figure 6.3 also reports the overall relative performance improvement over the baseline architecture. SICTC not only proves robust but also improves overall performance by 4.6% on average. For the barren page sensitive group of workloads, the performance improvements are 9.2% on average. SICTC improves performance in two ways. It reduces the L1 TLB misses by not installing barren pages, and it improves the average latency in the L1 by restricting solitary objects in only one of the ways. Figure 6.4 shows that SICTC increases the L1 MPKI slightly. Solitary objects content for space in the one way they are limited to. However, while this increases the L1 MPKI, other objects get to stay longer in the L1 and they prove to be more performance critical.

**Figure 6.4:** L1 Data Cache MPKI with SICTC Averaged Over 3 Application Cycles.
6.5 Avoiding Changes to the Page Table

The discussion in the previous sections focused on a barren page detection mechanism that allows barren pages to be identified in the L2 TLB and the page table. Changes to the page table while possible are often undesirable. Figures 6.2 and 6.3 also include results for LRU+BPF and SICTC when the barren bits are not stored in the page table (marked as "L2TLB BarrenBit"). The results shows that preventing barren bits from reaching the page table has a negligible impact on the performance benefits of BPF and SICTC. This can be explained by the number of hits barren pages receive while resident in the L2 TLB. Specifically, translations in the L2 TLB get replaced at a much slower rate than in the L1 TLB. As a result, when a page is identified as barren, it can usually be used many times from the L2 TLB before it is evicted. For the sensitive workloads, barren pages experience, on average, 17.3, 9.7, 13.9, 11.7, and 12.2 hits in the L2 TLB before being evicted for h2, lusearch, tradebeans, tradesoap, and specjbb respectively.

6.6 Performance With Larger On-Chip Caches

The baseline system used thus far includes two levels of cache hierarchy that may be representative of power-optimized processor cores. Modern, higher-end cores often include larger L3 caches. Accordingly, this section demonstrates that the proposed techniques remain effective even with a three-level hierarchy. Figure 6.5 shows performance with an infinitely large L3 with a 40-cycle latency. An infinitely large L3 provides a low bound on the amount of time lost during page walks and thus, barring other effects, provides a
limit on the performance improvements that are possible with the proposed techniques. Overall, performance improvements persist. For example, performance with SICTC improves on average by 10.5% for the sensitive applications and by 5.24% overall.

6.6.1 Summary

This section evaluates performance potential of the proposed BPF and SICTC techniques. For all workloads considered BPF and SICTC not only prove robust but also improve performance by 1.75% and 4.6% on average and by up to 4.52% and 11.32% respectively. In addition, this section shows that similar performance improvement is achieved without modification to the page table of the operating system.
Chapter 7

Related Work

7.1 Related Work

Sections 6.1 and 6.3 commented on state-of-the-art replacement policies and demonstrated that barren identification enhances their replacement decisions by providing per-page specific information. This section reviews other techniques that may alleviate the impact barren pages have on performance and the TLB. These include alternate garbage collection policies and superpages. This section also summarizes other TLB performance enhancing techniques.

7.1.1 Garbage Collection and Virtual Memory

Previous work demonstrated that GC interacts poorly with virtual memory managers by spreading the application’s working set over far more pages than necessary, causing many page faults to the disk [17]. To reduce these page faults, a *bookmarking* garbage collector that cooperates with the virtual memory manager to guide its eviction decisions was proposed. This work focuses on the impact memory-resident pages have on TLB performance. As system memories are getting larger, TLB performance has become more important.

7.1.2 Superpages

Superpages increase the reach of TLB entries by mapping much larger chunks of memory. They were originally proposed to efficiently handle special regions of memory such as the framebuffer. A TLB with superpages would suffer less from barren pages; many barren pages would map onto a single TLB entry thus reducing the working set of
barren pages as observed by the TLB. Superpages are not free of tradeoffs and it is well understood that they should be used with care. Specifically: (1) Dedicated OS and application code are required to support superpages [18] [19]. (2) Superpages must be aligned in both virtual and physical memory and the processing required to ensure alignment hurts performance. (3) The longer a system remains operational the higher the possibility that fragmentation would make superpage allocation increasingly expensive. As a result, systems may end up having to reserve a significant amount of memory just for superpages [20]. (4) Swapping superpages to disk is problematic due to their size. This is of particular concern for Java heaps where many pages contain dead objects and are natural candidates for swapping. (5) Superpages can negatively affect system performance when occupying large portions of memory by forcing excessive swapping for other pages [20]. (6) If not fully-utilized, superpages can increase the amount of I/O traffic and increase page initialization and fault latency. Each superpage often uses a single dirty bit; if set, the entire superpage must be written back to disk even if only one base page has been modified.

7.1.3 Other TLB Performance Enhancing Techniques

Previous work investigates the possibility of TLB prefetching for Java applications on embedded systems, e.g., [21]. While being able to reduce the TLB misses by 5 - 25%, TLB prefetching requires expensive hardware structures. Additionally, prefetching degrades performance for some workloads as the prefetched entries pollute the TLB. This work reduces the need for prefetching by reducing TLB misses through better replacement decisions or through avoiding the installation of pages with poor locality. Moreover, this work does not require a predictable TLB miss stream. Previous work investigates the benefits of triggering garbage collection proactively to increase cache and TLB efficiency [22]. While triggering garbage collection proactively results in better utilization of the cache and the TLB, it also introduces unnecessary garbage collection overhead in comparison to invoking garbage collection only when there is little free memory. Jones and Ryder find that a several allocation sites allocate objects with very similar lifetimes and therefore garbage collector should be made aware of the expected lifetimes of objects based on their site of allocation [23]. However, doing so requires the runtime to profile the lifetime of the objects created at each allocation site and thus introduces additional performance overhead. Furthermore, it cannot adapt to dynamic behavior changes in the application.

The Speculative TLB exploits the predictable order in which certain memory allocators manage memory to predict the virtual to physical mapping on TLB misses [24]. The Coalesced Large-Reach TLB (CoLT) extends TLB entries to cover multiple continuous
virtual pages that happen to be continuous in the physical space as well [25]. The techniques proposed in this work reduce the impact of contention due to barren pages and do not rely on a predictable allocation order nor on whether barren pages appear next to one another. For this reason, the techniques presented in this work are mostly complementary to the aforementioned works. Basu et al., revisit virtually-tagged caches in order to avoid looking up the TLB [26]. SICTC applies virtual tagging selectively and only for those pages that otherwise hurt TLB performance. In addition it maintains the physical tags which remain consistent for all blocks.

Basu et al., finds that most of the allocated memory in big data workloads are contiguous regions with the same page permissions [27], as a result, they advocates segmentation registers being used for such regions to reduce the impact of TLB misses on performance. However, this requires specific memory usages, i.e. contiguous regions of used memory with same permissions, and is therefore orthogonal to our work.
Chapter 8

Conclusion

This work identified that, as a result of automated dynamic memory allocation and de-allocation, many Java applications end up with many barren pages in their heap. These are pages containing mostly dead objects and a few actively referenced ones. Object creation order in Java is oblivious to the respective object’s expected lifetime; short-lived objects end up separating long-lived objects. The access behavior of barren pages was analyzed and the results guided a simple, low-cost barren page identification mechanism. The mechanism uses a single, saturating, tagged counter to incrementally identify barren pages in the Java heap. This dynamically collected information was used to enhance the TLB replacement policy or to avoid placing barren pages in the TLB improving overall performance. For those applications that exhibited many barren pages, the proposed mechanisms improved performance considerably while requiring minimal changes at the hardware level.

The key contributions of this work is that it exposes the barren page phenomenon along with a simple mechanism to dynamically detect them. Barren pages may have applications beyond those presented in this work both at the hardware and at the software levels. It is likely that the barren page phenomenon is not specific to Java but it may be inherent to many other object-oriented languages that use GC.
Bibliography


