Redesigning the Memory Hierarchy
for Memory-Safe Programming Languages

by

Yee Ling Gan

B.S., Massachusetts Institute of Technology (2016)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Engineering in Computer Science and Electrical Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2018

© Massachusetts Institute of Technology 2018. All rights reserved.

Author .................................................................
Department of Electrical Engineering and Computer Science
May 23, 2018

Certified by ............................................................
Daniel Sanchez
Associate Professor without Tenure
Thesis Supervisor

Accepted by ..........................................................
Katrina LaCurts
Chair, Master of Engineering Thesis Committee
Redesigning the Memory Hierarchy
for Memory-Safe Programming Languages

by

Yee Ling Gan

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2018, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Electrical Engineering

Abstract

We present Hotpads, a new memory hierarchy designed from the ground up for
modern, memory-safe languages like Java, Go, and Rust. Memory-safe languages hide
the memory layout from the programmer. This prevents memory corruption bugs,
improves programmability, and enables automatic memory management.

Hotpads extends the same insight to the memory hierarchy: it hides the memory
layout from software and enables hardware to take control over it, dispensing with the
conventional flat address space abstraction. This avoids the need for associative caches
and virtual memory. Instead, Hotpads moves objects across a hierarchy of directly-
addressed memories. It rewrites pointers to avoid most associative lookups, provides
hardware support for memory allocation, and unifies hierarchical garbage collection
and data placement. As a result, Hotpads improves memory performance and efficiency
substantially, and unlocks many new optimizations. This thesis contributes important
optimizations for Hotpads and a comprehensive evaluation of Hotpads against prior
work.

Thesis Supervisor: Daniel Sanchez
Title: Associate Professor without Tenure
Acknowledgments

I would like to thank my research advisor, Prof. Daniel Sanchez, and my mentor, Po-An Tsai. Hotpads was first proposed by Po-An and Daniel, and many of the material in this thesis is adapted from a paper that we submitted. I have learned a lot from their research philosophy. Their unbounded enthusiasm and endless energy in conducting research are very contagious and really inspiring.

I am grateful to Po-An for being a great mentor since my senior year. He has taught me all about our group’s infrastructure and a lot of good practices on how to conduct research. He has been patient and kind, I really appreciate that he allowed me to ask and always answered any question I had. In our meetings, he made sure that I fully understood all concepts and details. He has been forgiving of the mistakes that I made as I was learning, and during this work, helped me to not panic especially during critical moments.

I am thankful to Prof. Sanchez for giving me the opportunity to do research in his group. His range of knowledge in so many areas is amazing, I have learned a lot from our weekly meetings. I am thankful for the guidance, advice, and suggestions he has given throughout my various projects. He always provides time and help whenever we need it.

I would also like to thank members of my group: Anurag Mukkara, for the interesting discussions about research, his help and advice with graphs; Harshad Kasture and Nosayba El-Sayed for the late working nights and food outings; Mark Jeffrey, Suvinay Subramanian, and Guowei Zhang, for their help and advice; and Prof. Joel Emer, Maleen Abeydeera, Virginia Chiu, Nathan Beckmann, Ryan Lee, and Victor Ying—it has been a wonderful experience to grow and learn with everyone.

I would like to thank my academic advisor, Prof. Vladimir Bulovic, who has given me good advice on classes and research every year. I would also like to thank everyone that I have had a chance to meet on the 7th, 8th and 9th floors of CSAIL, especially Mary McDavitt, who takes care of us all on this corner of the 7th floor; Prof. Martin Rinard, who always gives friendly doses of life advice; and Malavika, Nadia and Shachar, for welcoming a newbie on the 7th floor.

Last but certainly not least, I am grateful to my family, who have always been
there to support me through my best times and my worst times, to allay my worries and fears. Their unconditional love, unwavering belief in me when I needed it the most, and their always sound advice has guided me to where I am today.
Contents

1 Introduction .......................................................... 13
  1.1 Motivation ......................................................... 13
  1.2 Hotpads ............................................................ 14
  1.3 Contributions ..................................................... 16

2 Background .......................................................... 17
  2.1 Background on Memory-Safe Languages ......................... 17
    2.1.1 Garbage collection (GC) .................................... 17

3 Related Work ......................................................... 21
  3.1 Cooperative cache scrubbing .................................. 21
  3.2 Architectural support for memory safety ....................... 22
  3.3 Cache hierarchies ................................................ 22

4 Hotpads Overview .................................................. 25

5 Hotpads ISA: Hiding the memory layout ......................... 29

6 Hotpads Microarchitecture ........................................ 33
  6.1 Pointer format .................................................... 33
  6.2 Canonical levels and invariants ................................ 34
  6.3 Pad organization ................................................ 34
  6.4 Steady-state operation ......................................... 36
    6.4.1 Performing memory accesses .............................. 36
    6.4.2 Performing other pointer operations ...................... 37
    6.4.3 Maintaining CE metadata ................................ 38
6.5 The collection-eviction (CE) process ........................................... 38
6.6 Supporting arbitrarily large objects ........................................... 42
6.7 Object-level coherence .......................................................... 42
6.8 Instruction pads vs. instruction caches ....................................... 43
6.9 Crosscutting issues ............................................................... 43

7 Experimental Methodology ......................................................... 45
7.1 Hardware ................................................................................. 46
  7.1.1 Core modifications ......................................................... 46
  7.1.2 Speculative execution .................................................... 46
  7.1.3 Cache scrubbing ........................................................... 46
7.2 Software ................................................................................. 47
  7.2.1 JVM .............................................................................. 47
  7.2.2 Workloads ..................................................................... 48

8 Evaluation .................................................................................. 49
8.1 Latency, energy, and area of caches vs. pads ............................ 49
8.2 Hotpads outperforms traditional hierarchies ......................... 50
8.3 Hotpads reduces data movement across the memory hierarchy ... 53
8.4 Pointer rewriting avoids most associative lookups ............... 53
8.5 CEs are fast and infrequent .................................................... 54
8.6 Hotpads performs well on multithreaded workloads ............. 56
8.7 Legacy mode incurs small performance overheads ............... 56
8.8 Cache scrubbing validation .................................................... 56
8.9 Effects of young generation size ............................................. 58
8.10 Sensitivity studies ............................................................... 61
  8.10.1 Allocation threshold for Hotpads’s L1 pad ....................... 61
  8.10.2 Baseline’s cache line size and Hotpads’s subobject threshold 64

9 Future Work and Conclusion ...................................................... 67
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1</td>
<td>Generational GC example.</td>
<td>18</td>
</tr>
<tr>
<td>4-1</td>
<td>Hotpads hierarchical memory system.</td>
<td>25</td>
</tr>
<tr>
<td>4-2</td>
<td>Pad organization.</td>
<td>25</td>
</tr>
<tr>
<td>4-3</td>
<td>Hotpads’s key features.</td>
<td>26</td>
</tr>
<tr>
<td>6-1</td>
<td>Hotpads pointer format.</td>
<td>33</td>
</tr>
<tr>
<td>6-2</td>
<td>In-pad object format.</td>
<td>35</td>
</tr>
<tr>
<td>6-3</td>
<td>Canonical tag entry format.</td>
<td>35</td>
</tr>
<tr>
<td>6-4</td>
<td>In-pad metadata format.</td>
<td>35</td>
</tr>
<tr>
<td>6-5</td>
<td>Steps in L1 pad access.</td>
<td>36</td>
</tr>
<tr>
<td>6-6</td>
<td>Eviction that changes the canonical level of a non-evicted object.</td>
<td>40</td>
</tr>
<tr>
<td>6-7</td>
<td>Dual-ended compaction.</td>
<td>41</td>
</tr>
<tr>
<td>8-1</td>
<td>Simulation results for single-threaded workloads.</td>
<td>51</td>
</tr>
<tr>
<td>8-2</td>
<td>Breakdown of bytes read and written per level.</td>
<td>54</td>
</tr>
<tr>
<td>8-3</td>
<td>Allocated, evicted, and collected bytes per pad level.</td>
<td>54</td>
</tr>
<tr>
<td>8-4</td>
<td>Breakdown of access types across pad levels.</td>
<td>54</td>
</tr>
<tr>
<td>8-5</td>
<td>CDFs of lengths and intervals for L1 CEs.</td>
<td>55</td>
</tr>
<tr>
<td>8-6</td>
<td>Simulation results for multithreaded workloads.</td>
<td>56</td>
</tr>
<tr>
<td>8-7</td>
<td>Cache scrubbing read traffic.</td>
<td>57</td>
</tr>
<tr>
<td>8-8</td>
<td>Cache scrubbing write traffic.</td>
<td>57</td>
</tr>
<tr>
<td>8-9</td>
<td>Performance of different cache scrubbing optimizations normalized to the baseline.</td>
<td>57</td>
</tr>
</tbody>
</table>
8-10 How the average execution time, memory traffic and energy changes with the baseline and cache scrubbing schemes as the young generation size increases. .......................................................... 58
8-11 Young generation size sweep for lusearch. ........................................... 59
8-12 Young generation size sweep for xalan. ............................................... 59
8-13 Young generation size sweep for coloring. .......................................... 59
8-14 Gmean of the ratio of cache scrubbing metrics to baseline metrics. ........ 61
8-15 L1 pad allocation threshold–Distribution of object lifetime. ............... 62
8-16 L1 pad allocation threshold–Number of CEs. ...................................... 62
8-17 L1 pad allocation threshold–Number of bytes moved from a pad level. 63
8-18 Cache line sweep vs. subobject threshold sweep. ............................. 63
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Hotpads ISA</td>
<td>30</td>
</tr>
<tr>
<td>7.1</td>
<td>System configuration</td>
<td>45</td>
</tr>
<tr>
<td>7.2</td>
<td>Configured heap size for applications</td>
<td>48</td>
</tr>
<tr>
<td>8.1</td>
<td>Pads configuration</td>
<td>49</td>
</tr>
<tr>
<td>8.2</td>
<td>CE duration and frequency across pads.</td>
<td>55</td>
</tr>
<tr>
<td>8.3</td>
<td>Summary single threaded Hotpads improvements</td>
<td>61</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation

Computer systems still cater to early programming languages like C and Fortran. These languages expose a flat memory address space to the programmer and allow memory-unsafe operations, such as performing arbitrary pointer arithmetic and accessing arbitrary memory locations. A flat address space was a natural interface for the memories of the earliest computers but it is a poor interface for modern memory systems, which are organized as deep hierarchies. To preserve the illusion of a flat address space, these hierarchies rely on expensive translation mechanisms, including associative caches and virtual memory.

These early languages also require programmers to manage memory manually. This allows a whole slew of software engineering and security bugs, such as double-freeing memory, use-after-free bugs, dangling pointers, memory leaks, and buffer overflows.

Fortunately, languages have evolved. Most modern languages, like Java and Go, are memory-safe: they do not expose raw pointers or allow accessing arbitrary memory locations. Memory safety greatly improves programmability: it avoids memory corruption and simplifies memory management.

Memory safety adds overheads in current systems, but these costs largely stem from mismatched memory system and language semantics. By redesigning the memory hierarchy to cater to memory-safe languages, we can avoid many of the overheads of conventional hierarchies and unlock new optimizations.
Prior work has sought to bridge the semantic gap between memory-safe languages and architectures by accelerating common compute operations, such as type checks or object-based addressing [18,93] and protection [28,99] (Section 2.1). But they do so within a conventional memory hierarchy. By contrast, this work aims to redesign the memory hierarchy to cater to memory-safe languages.

The key insight we exploit is that memory-safe languages hide the memory layout from the programmer. Programmers never deal with raw memory addresses, but see pointers as abstract data types (ADTs [49]) that may only be dereferenced or compared. In software, hiding the memory layout is key to enabling automatic memory management, i.e., garbage collection (GC). We extend this insight to hardware through Hotpads, a novel memory hierarchy designed for memory-safe languages.

### 1.2 Hotpads

Hotpads hides the memory layout and takes control over it, dispensing with the flat address space abstraction. The Hotpads ISA (Chapter 5) prevents programs from reading or manipulating raw pointers, enabling Hotpads hardware to safely rewrite them under the covers. This avoids the need for associative caches.

Instead, Hotpads is a hardware-managed hierarchy of directly-addressed memories similar to scratchpads, which we call pads. Each pad has two contiguous regions of allocated objects and free space, and is managed using techniques similar to generational GC (Chapter 6). Specifically, Hotpads relies on four key features:

- **Implicit, object-based data movement.** All data movement happens implicitly, in response to memory accesses. If the core initiates an access to an object not currently in the L1 pad (analogous to the L1 cache), the object is copied to the L1 pad, using some of its free space, and the access is performed.

- **Pointer rewriting to avoid associative lookups.** Objects copied into the L1 pad may have pointers to objects beyond the L1 pad. When the program dereferences one of these pointers, hardware automatically rewrites the pointer with the newly-brought-in-object’s L1 location. This way, subsequent dereferences of the same pointer do not incur an associative lookup. Pads still require some associative lookups, e.g., for non-L1 pointers. But whereas caches perform an associative lookup
on every access, pointer rewriting makes associative lookups rare.

- **In-hierarchy object allocation.** New objects are allocated in the L1 pad’s free space (or, if they are large, in a higher-level pad). New objects require no backing storage in main memory so they can be accessed cheaply, without misses.

- **Unified hierarchical garbage collection and evictions.** When a pad fills up, it triggers a process similar to GC to free space. This process both detects dead objects and reclaims their space, evicting live (i.e., referenced) but not-recently-accessed objects to the next-level pad. This process, which we call *collection-eviction (CE)*, leverages both the locality principle that underpins caches and the generational hypothesis (most objects die young) that underpins generational GCs [92]. CEs happen concurrently with program execution and are hierarchical. While small pads incur more frequent CEs, their small size makes each CE very cheap.

Hotpads manages pads entirely in hardware but leaves management and garbage collection of main memory to software. Hotpads supports arbitrarily large objects, which may not fit in pads, by caching them in small chunks called subobjects. Hotpads maintains coherence at object granularity using standard protocols. Finally, Hotpads includes a compatibility mode to support memory-unsafe programs with minor slowdowns.

We evaluate Hotpads using detailed simulation and a heavily-modified research Java Virtual Machine (JVM) running Java benchmarks (Chapter 7). Hotpads substantially outperforms cache hierarchies for three key reasons. First, operating in variable-size objects instead of fixed-size cache lines uses on-chip capacity more efficiently. Second, pointer rewriting avoids most associative lookups, making L1 pads 3.3× more efficient than L1 caches. Third, CEs dramatically reduce GC overheads, by 8× on average. Overall, Hotpads improves performance by 41% and reduces memory hierarchy energy by 2.8×. Finally, Hotpads slows down memory-unsafe programs by only 4%.

Beyond these gains, Hotpads opens up new and exciting avenues to improve the memory hierarchy, including improved security by avoiding cache side-channels, and new isolation, resource management, and concurrency techniques. We leave these and other techniques to future work (Chapter 9).
1.3 Contributions

Hotpads is a collaborative project. This thesis describes the full details of Hotpads and emphasizes the author’s key contributions to the project:

- **Implementing pointer rewriting and shadow subobject registers**: Pointer rewriting and shadow subobject registers are used in the first level pad as optimizations to reduce associative lookups.
- **Evaluating Hotpads against prior work**: We make a detailed comparison of Hotpads to a baseline with cooperative cache scrubbing (Section 3.1, Section 8.8).
- **Sensitivity studies and design space exploration for Hotpads**: These studies help find the best performing Hotpads configuration (Section 8.10).
Chapter 2

Background

2.1 Background on Memory-Safe Languages

It is the right time to redesign memory systems for memory-safe languages, also known as managed languages. These languages rely on a combination of type systems, static analysis, and runtime checks to prevent programs from manipulating memory directly. They prevent large classes of bugs, such as buffer overflows and use-after-free errors, and enable garbage collection.

Nearly all languages introduced in the last 30 years are memory-safe, including Java, JavaScript, Go, Rust, Scala, Swift, C#, and Python. Among the top 20 languages in four popularity indices [24, 68, 72, 90], only C++, C, and assembly are not memory-safe. Although many applications and most operating systems are still written in C/C++, several projects like the Singularity OS [29], Verve [101], and Redox [73] feature an entire software stack written in memory-safe languages. Therefore, Hotpads targets memory-safe languages first, and includes a slower compatibility mode for legacy applications. (Section 6.9).

2.1.1 Garbage collection (GC)

Garbage collection, also known as automatic memory management, frees programmers from the burden of manually freeing memory. Instead, the system automatically reclaims memory occupied by dead (i.e., unreferenced) objects. There are two main types of GC: tracing [52] and reference counting [16].
Tracing GC algorithms periodically scan the heap to make space available. They start with a set of root pointers that are outside of the managed heap (e.g., static, stack, or register variables). Starting from the roots, tracing GC traverses heap objects to find all live (i.e., reachable) ones. It then reclaims the space taken by dead (i.e., unreachable) objects.

Reference counting keeps a count on the number of references to an object. When that count falls to zero, the object is considered dead and can be immediately collected.

Though both styles have pros and cons [7, 51], tracing GC is more general (e.g., it supports cyclic references among objects) and it is more widely used. Hotpads leverages the principles behind tracing GC to manage the memory hierarchy.

Tracing algorithms can be moving or non-moving [97]. Moving GCs move all live objects on each collection to leave a contiguous free space region and avoid fragmentation. By contrast, non-moving GCs leave live objects in-place and use freelists or other data structures to track free space. Moving GC is more common because it simplifies memory allocation. We find that moving GC is a natural match for Hotpads.

Prior work has proposed many techniques to reduce GC overheads [4, 6, 9, 10, 67]. We focus on two dimensions: generational and concurrent GC.

Generational GC algorithms exploit the generational hypothesis, the empirical observation that most objects die young [92]. They use separate heaps for objects of different ages. Fig. 2-1 shows an example with two heaps, young and old. New objects are allocated in the small young heap. When the young heap fills up, it is GC’d and its live objects are moved to the old heap. When the old heap fills up, both heaps are GC’d.
Generational GCs improve performance because each GC of the small young heap is cheap, and filtering objects that die young greatly reduces the frequency of expensive full GCs. Generational GCs also improve cache locality [7]. For these reasons, most runtimes use generational GC [55,64,69,74].

Generational GCs and cache hierarchies share many similarities: both build on analogous empirical observations (generational hypothesis vs. locality principle), adopt a multi-level structure, and seek to make the common case fast. Hotpads unifies generational GC and hierarchical data placement.

Concurrent GC algorithms reduce the long pauses that arise in conventional or stop-the-world GC, where the program is stopped while GC takes place. Long pauses have traditionally hindered the adoption of tracing GC in environments where real-time or low-latency operation is important. Concurrent GCs reduce pauses by running most GC phases concurrently with the program [3,20,46,88]. However, they have higher overheads to handle races between program and GC threads, and still incur some pauses (e.g., to interrupt threads and produce their root sets). For example, ZGC [46] reduces throughput by 15% and requires pauses of a few milliseconds. Thus, concurrent GC is used selectively, when long pauses are detrimental.

The collection-eviction process of Hotpads, which encompasses GC, is concurrent. It incurs minimal overheads and requires negligible pause times (tens of cycles, Section 7.1).

Hardware techniques to accelerate GC date back to Lisp machines [56]. Recent work includes HAMM [33], which accelerates reference counting to reduce young heap GC overheads. Cooperative cache scrubbing [81] extends the ISA with scrubbing instructions to recycle dead space in caches without incurring memory traffic. Finally, several concurrent GC implementations exploit hardware transactional memory to reduce overheads [3,53,75,88].

This prior work reduces GC overheads in conventional cache hierarchies. By contrast, Hotpads is a new memory hierarchy that exploits the key principle behind memory-safe languages, hiding the memory layout, to improve efficiency further.
Chapter 3

Related Work

This chapter reviews related work in hardware and software support for modern languages. Section 3.1 reviews cooperative cache scrubbing, a hardware/software technique to reduce memory traffic that we compare Hotpads against. Section 3.2 describes architectural support for memory safety. Section 3.3 describes related work on cache hierarchy optimizations.

3.1 Cooperative cache scrubbing

Sartor et al. proposed cooperative hardware/software cache scrubbing to reduce memory traffic and save energy for managed languages [81]. They found that 10–60% of writebacks are useless because they contain dead objects that are guaranteed to never be read again. Furthermore, zero initialization during memory allocation also wastes memory traffic, as cache lines are brought in from memory only to be written over immediately.

Cache scrubbing introduces instructions to reduce memory traffic. Among these, the combination of \texttt{clzero} and \texttt{clinvalidate} is able to attain the full memory traffic savings. \texttt{clzero} reduces read traffic from main memory by zero-initializing cache lines directly in the cache without fetching them from memory. \texttt{clinvalidate} reduces write traffic from main memory by invalidating dead cache lines without having to write them back to main memory.

The cache scrubbing instructions significantly reduce application (i.e. mutator)
memory traffic and energy, but do not alter the cost of garbage collection.

3.2 Architectural support for memory safety

Much prior work has focused on bridging the semantic gap between memory-safe languages and architectures. Object-oriented systems [17, 18, 57, 63, 93] reduce virtual call overheads and accelerate object references. Capability-based systems [19, 28, 43, 99] provide object-based memory protection and isolation. Typed architectures [2, 14, 38, 39, 86] accelerate dynamic type checks. Whereas this prior work focuses on core design and uses a standard cache hierarchy, we focus on redesigning the memory hierarchy.

Though we focus on GC-based languages, prior work has proposed software [1, 5, 61] and hardware [11, 60, 94] techniques to make languages with manual memory management memory-safe, e.g., by tracking bounds for all pointers. Hotpads could be combined with these techniques to work on C/C++ programs. These programs often have sizable memory allocation costs [34, 35], which Hotpads would avoid.

3.3 Cache hierarchies

Hotpads leverages program-level information and object-level data movement to reduce traffic and avoid most associative lookups. Prior work has also sought to achieve some of these benefits in the context of cache hierarchies.

Adaptive-granularity cache designs like sector caches [48, 84] and Amoeba [42] improve utilization and reduce traffic, but they have more involved tag lookups and require predictors to fetch data at the right granularity [32, 42]. Hotpads manages variable-sized objects instead of fixed-sized cache lines, achieving many of these benefits without the overhead of these techniques.

The V-Way cache [71] decouples tag and data arrays, and improves associativity by oversizing the tag array. Like the V-Way cache, pads use an auxiliary tag array (the c-tags array) that is decoupled from the data array. Hotpads adopts this organization to manage the data array independently.

GPUs and many accelerators [12, 13, 23, 25, 95] use software-managed scratchpads to avoid the inefficiencies of caches. But scratchpads are hard to use—they require
programmers or compilers to manage data placement and movement. As a result, only regular programs can use them well [45].

Stash [41] seeks to combine the benefits of scratchpads and caches. Programmers can map a global memory region onto the Stash and access it like a scratchpad. Hits achieve scratchpad-like efficiency, and misses automatically fetch data like a cache. Like Stash, Hotpads achieves cheap direct accesses. However, Hotpads does not require programmers to explicitly map data into on-chip memories.

Virtual memory and caches conventionally use two separate associative lookups, on TLBs and cache tags. TLC [83], D2D [82], and cTLB [44] fold cache tag information into the TLB to reduce or eliminate cache tag lookups. While they reduce overheads, they still require an associative lookup (to the TLB) on every access and introduce other complexities. By contrast, Hotpads avoids associative lookups on most accesses.
Chapter 4

Hotpads Overview

Fig. 4-1 shows the general structure of Hotpads. Hotpads is a hardware-managed hierarchy of directly-addressed memories similar to scratchpads, which we call pads. Unlike in software-managed scratchpad hierarchies with explicit data movement, in Hotpads all data movement happens implicitly, in response to memory accesses. Fig. 4-1 shows a hierarchy with three levels of pads, but Hotpads supports an arbitrary number of levels.

**Pads:** Fig. 4-2 shows the internal structure of each pad. Most space is devoted to the *data array*, which is managed as a circular buffer. The data array has a contiguous block of *allocated objects* followed by a block of *free space*. The data array uses simple *bump pointer* allocation: fetched or newly-allocated objects are placed at the end of the allocated region.

Fig. 4-2 also shows that pads have some *metadata* (e.g., to record whether each word holds a pointer) and a *canonical tags* (*c-tags*) array, which is needed for a minority of the accesses. We will later see how these auxiliary structures are used.

![Figure 4-1: Hotpads is a hierarchical memory system with multiple levels of pads.](image1)

![Figure 4-2: Pad organization.](image2)
Key features: As explained in Chapter 1, Hotpads relies on four novel features: (1) implicit, object-based data movement, (2) pointer rewriting to avoid associative lookups, (3) in-hierarchy object allocation, and (4) unified hierarchical GC and evictions.

Fig. 4-3 illustrates these features through a simple example showing a single-core system with two levels of pads. Only the data array of each pad is shown. 1 shows the initial state of the system: the core’s register file holds a pointer to object A in the L2 pad, and A points to B in main memory. The L1 and L2 pads also hold other objects (shown in solid orange) that are not relevant to this example.

**Implicit data movement and pointer rewriting:** 2 shows the state of the system after the core issues an access to A. First, A is copied into the L1 pad, taking some free space. Second, the pointer in the register file is *rewritten* to point to this L1 copy.
This way, subsequent dereferences of this pointer access the L1 copy directly.

Pointer rewriting applies not only to registers, but to pad data as well. 3 shows the state of the system after the core dereferences A’s pointer to B. B is copied into the L1 pad, the core is given this L1 address, and the L1 A’s pointer to B is rewritten to point to B’s L1 copy. Further dereferences simply access the L1 data array, avoiding any associative lookups.

Pointer rewriting avoids most but not all associative lookups. For example, given the state in 3, if some other object in the L1 had a pointer to A’s L2 copy, the L1 must detect that a copy of A is already in the L1, then rewrite that pointer. This is the role of the canonical tags, which we explain later.

In-hierarchy object allocation: 4 shows the state of the system after the core creates a new object C. C is allocated directly in the L1 pad’s free space, and requires no backing storage in main memory or other pads.

Unified hierarchical garbage collection and evictions: In 5 the L1 pad has filled up, so the pad starts a collection-eviction (CE) to free L1 space. Similarly to GC, a CE walks the data array to detect live vs. dead objects. In addition to GC, a CE evicts live but not-recently-accessed objects to the next-level pad. In this example, C is dead (i.e., unreferenced) and a new object D is referenced from B, and thus live. Note that B’s L1 copy has been modified, so the main memory data is now stale. Only B has been accessed recently in the L1.

6 shows the state after the CE: First, C has been collected. Second, A and D have been evicted to the L2 pad. Since A already had an L2 copy and was not modified, this is a silent eviction and requires no writeback (pointer rewrites are not modifications). By contrast, D is allocated new space in the L2 pad. Third, B has been kept in the L1 and moved to the start of the array. Like moving GC, live objects are compacted into a contiguous region to simplify space management.

CEs happen concurrently with program execution and are hierarchical, i.e., each pad can perform a CE independently from larger, higher-level pads. In our example, the L1 pad performs its CE independently from the L2 pad. To ensure this, we enforce a key invariant: objects at a particular level may only point to objects at the same or higher levels (Section 6.2). For example, an L2 object may point to objects in the L2 or main memory, but not to L1 objects.
Chapter 5

Hotpads ISA: Hiding the memory layout

Hotpads treats pointers as abstract data types [49] whose contents may not be accessed, enabling the microarchitecture to manipulate them. Hotpads introduces new instructions to support three pointer operations: dereference, comparison, and object allocation, summarized in Table 5.1.

Addressing discipline: Hotpads uses a single addressing mode, base+offset, where the base register is always an object pointer. As shown in Table 5.1, the offset can be an immediate (base+ displacement) or a register (base+index). The standard load and store instructions can be used to access non-pointer data.

In Hotpads, the base register rb used in memory accesses must be an object pointer, i.e., it must contain the object’s starting address. Pointers to arbitrary locations within an object are not allowed. This restriction is not unique to Hotpads: several JVMs enforce it to facilitate pointer manipulations.

Pointer load and store: Hotpads provides load and store variants to access pointers: ldptr and stptr (Table 5.1). These instructions have the same semantics as ld and st, but they let the system know that the data accessed is a pointer.

Pointer dereference: Hotpads includes a dereference instruction to facilitate pointer rewriting: derefptr (Table 5.1). Like ldptr, derefptr loads the pointer at address disp(rb). Unlike ldptr, derefptr denotes that the program immediately intends to access the pointed-to object.
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Load</td>
<td>ld rd, disp(rb)</td>
<td>rd &lt;- Mem[EffAddr]</td>
</tr>
<tr>
<td>Data Store</td>
<td>st rd, disp(rb)</td>
<td>Mem[EffAddr] &lt;- rd</td>
</tr>
<tr>
<td>Pointer Load</td>
<td>ldptr rp, disp(rb)</td>
<td>rp &lt;- Mem[EffAddr]</td>
</tr>
<tr>
<td>Pointer Store</td>
<td>stptr rp, disp(rb)</td>
<td>Mem[EffAddr] &lt;- rp</td>
</tr>
<tr>
<td>Pointer Dereference</td>
<td>derefptr rp, disp(rb)</td>
<td>rp &lt;- Mem[EffAddr]; brings object in L1</td>
</tr>
<tr>
<td>Pointer Equality</td>
<td>seqptr rd, rp1, rp2</td>
<td>rd &lt;- (rp1==rp2)? 1:0</td>
</tr>
<tr>
<td>Allocation</td>
<td>alloc rp, rs1, rs2</td>
<td>NewAddr &lt;- Alloc(rs1); Mem[NewAddr] &lt;- rs2; rp &lt;- NewAddr;</td>
</tr>
</tbody>
</table>

Table 5.1: Hotpads ISA. rd/rs denote registers that hold data; rp/rb hold a pointer. All memory accesses use base+offset addressing. The table shows the base+displacement format disp(rb), where EffAddr = rb + disp and disp is an immediate. Instructions also have base+index variants, e.g., ld rd, (rb,rs), where EffAddr = rb + rs.

*derefptr* enables efficient pointer rewriting on L1 data. If the pointed-to object is not in the L1 pad, the system brings it in and rewrites the dereferenced pointer in the L1 (e.g., A’s pointer to B in Fig. 4-3). If a pointer was first accessed using *ldptr* and then dereferenced with *ld*, it would be hard for the system to, at *ld* time, rewrite the pointer’s original location that *ldptr* accessed. Conversely, *ldptr* is also needed because programs sometimes need to compare pointers, and bringing in their pointed-to objects would be wasteful.

**Pointer comparison:** *seqptr rd, rp1, rp2* (set-if-equal-pointers) sets register *rd* to 1 if source registers *rp1* and *rp2* point to the same object, and to 0 otherwise (Table 5.1). This instruction is needed because registers may point to different copies of the same object, and thus have different bit patterns.

**Object layout:** The Hotpads ISA imposes some rules about the layout of objects within the pads:

1. Objects must be word-aligned (our implementation uses 64-bit words) and be at least two words long.
2. The first word of the object contains an immutable *type id*, set at object creation.

This type id lets the program identify the object’s type (e.g., it can be a vtable...
pointer). The top 16 bits of the type id must be 0, as Hotpads uses them to store some per-object metadata.

The type id is opaque to Hotpads. Hotpads does not rely on type identifiers to determine which words of an object are pointers. Instead, it relies on \texttt{ldptr}, \texttt{stptr}, and \texttt{derefptr} to identify them. Reading the first word of an object (i.e., \texttt{ld rd, 0(rb)}) returns its type id; writing the first word of an object (i.e., \texttt{st rd, 0(rb)}) is an illegal operation.

**Object allocation:** Finally, Hotpads provides an instruction to allocate a new object: \texttt{alloc rp, rs1, rs2} allocates a new object with size (in words) given in \texttt{rs1}, type id given in \texttt{rs2}, and writes the new pointer to \texttt{rp}.

**Code addresses:** For simplicity, Hotpads treats the code segment as a single object and does not hide code addresses (Section 6.8). Indirect jumps use normal addresses.

**Pointer integrity:** Hotpads tracks enough metadata to guarantee the integrity of pad pointers. A program cannot transform non-pointer data into a pointer to a pad. But Hotpads does not store any metadata in main memory, and relies on language-level memory safety to guarantee the integrity of main memory pointers. This means that compiler or JIT engine bugs may cause programs to fetch the wrong data from main memory, but they will never corrupt or illegally access pad state. These bugs are avoidable with a small, automatically verified trusted code base [101]. Alternatively, Hotpads could track main-memory object metadata to prevent them in hardware (Section 6.9).
Chapter 6

Hotpads Microarchitecture

We now present the microarchitecture of Hotpads. We first explain the operation of Hotpads under several simplifications: we assume all objects are of limited size (e.g., up to 64 bytes); we only consider data accesses, not instruction accesses; and we consider a single-core system running a single process. We will remove these limitations from Section 6.6 onwards.

6.1 Pointer format

Fig. 6-1 shows the format of Hotpads pointers. The lower 48 bits contain the object’s address, and the upper 16 bits contain several pieces of metadata, including the object’s size (in words) and two bits whose roles we will introduce later. Embedding metadata in pointers simplifies several operations.

![Figure 6-1: Hotpads pointer format.](image)

All object addresses in Hotpads are word addresses. Hotpads maps the data arrays of all pads and main memory to different addresses. We use power-of-2-aligned mappings for simplicity. For example, a 64 KB L1 pad, a 1 MB L2 pad, and a 2 GB main memory would use mappings 64–128 KB, 1–2 MB, and 2–4 GB. These mappings make it trivial to determine an address’s level, and the per-level address is the full address’s lower bits. Finally, an empty (null) pointer is the all-zeros string.
6.2 Canonical levels and invariants

Objects start their life in the L1 pad and move up the hierarchy as they are evicted by successive CEs. We define an object’s canonical level as the largest level it has reached since it was created. For example, in Fig. 4-3, C’s canonical level is L1, A’s is L2, and B’s is main memory. This is the case even though the L1 pad has copies of A and B.

An object’s canonical address is the object’s address at its canonical level. A pointer’s canonical bit (Fig. 6-1) stores whether the address it holds is canonical.

For simplicity, Hotpads enforces four key invariants:

**Invariant 1**: An object always exists at its canonical level.

An object may have copies in smaller, lower levels than its canonical, but the object’s canonical level is its backing store. In other words, in Hotpads the canonical level acts like main memory does in a cache hierarchy—it is the object’s “final resting place”. Unlike in cache hierarchies, any level can be an object’s canonical. This level only grows over time.

**Invariant 2**: Pointers in the L1 pad can hold either L1 or canonical addresses, while pointers in other pads and memory always hold canonical addresses.

This invariant simplifies pointer rewriting (Section 6.4). While it limits rewriting to the L1, this is where it is most valuable.

**Invariant 3**: Objects at a particular level may only point to objects at the same or higher levels.

**Invariant 4**: When an object dies, it may only be garbage-collected at its canonical level.

These two invariants enable hierarchical CEs (Section 6.5).

6.3 Pad organization

As we saw in Chapter 4, each pad consists of a data array, a canonical tags (c-tags) array, and some per-object and per-word metadata (Fig. 4-2). We now describe each of these components.
Fig. 6-2 shows the format of objects stored in the data array, detailing the format of the first word. Hotpads manages the first word’s upper 16 bits, which contain rarely-accessed metadata for coherence and CEs. In addition, if this object is non-canonical (i.e., a copy), the object’s canonical pointer is stored directly above the object. This allows translating from a non-canonical to a canonical pointer with a single data array access.

The c-tags array is a conventional set-associative structure that allows mapping the canonical address of a resident object into its per-level address. Fig. 6-3 shows the format of each c-tag entry. Only non-canonical objects need a c-tag entry. For example, the L2 pad needs to hold the canonical-to-L2 address translation for every object whose canonical level is L3 or main memory, but not for ones with canonical level L2.

Finally, Fig. 6-4 details the per-word and per-object metadata. Each word has an associated pointer bit that tracks whether the word holds a pointer. Each object has six associated metadata bits: valid and dirty bits, and recency bits used for evictions.

This metadata is not in the data array because it is used on nearly every access. Instead, it is kept in separate, narrow arrays. Because each object can be at least two words long, the per-object metadata array has one entry for every two words of the data array (this way it can be indexed directly). Overall, this metadata takes 4 bits/word, a 6.25% overhead.
6.4 Steady-state operation

We first explain Hotpads’s operation in steady state, i.e., between executions of the CE process. Section 6.5 explains CEs.

6.4.1 Performing memory accesses

L1 pad accesses: A request from the core includes both base (object pointer) and offset (word within the object). Fig. 6-5 shows the flow of accesses. If the pointer is L1 and canonical, the access proceeds with no checks (by Invariant 1, the data must be there). If the pointer is L1 and non-canonical (i.e., a copy), the pad checks the object’s valid bit. If the valid bit is set, the access is performed by directly indexing the data array. If the valid bit is unset, the L1 pad reads the object’s canonical pointer and restarts the access with it. Loads check the valid bit in parallel with the access. Stores set the dirty bit.

Accesses beyond the L1 pad: Pad misses are sent to the next-level pad. These misses always use a canonical pointer (Invariant 2), but there may be copies of the object at levels before its canonical. Therefore, an access traverses all levels until it finds a copy or reaches the object’s canonical level.

Each access proceeds as in the L1, except that, on a hit, the entire object is copied to the L1. For example, in Fig. 4-3, the core is dereferencing A’s pointer.
to B, which is a main memory address. B’s access misses on L1’s c-tags, then on L2’s c-tags, then is served from main memory.

**Invalidations:** The c-tags array may fill up or suffer from conflicts, so inserting a new canonical→level address translation may require removing another translation. In this case, the pad picks the least-recently-accessed object copy in the set and marks it as invalid. If the copy is dirty, it is written back to the next level. If the copy is clean, it is simply dropped.

In a single-core system, invalidations are an optimization to support removing c-tag entries cheaply. Pointers to invalidated objects need not be found and rewritten. Alternatively, we could size the c-tags array conservatively and trigger a CE upon overflow. However, coherence needs invalidations (Section 6.7), so using them for c-tag overflows too is the sensible choice.

**Pointer rewrites:** Pointer rewriting is performed whenever the L1 pad is accessed with a non-L1 pointer. If the access came from a conventional load or store instruction, then the object’s L1 address is sent to the core so that the relevant register can be rewritten. If the access came from a derefptr instruction, the L1 object’s field that contains the non-L1 pointer is rewritten.

Pointer rewrites of objects in the L1 pad are not writes, as they do not change the pointer’s semantics—it still points to the same object. Rewrites do not set the object’s dirty bit and, in multicores, are performed even if the pad has a read-only copy of the object (Section 6.7). Pointer stores do set the dirty bit.

### 6.4.2 Performing other pointer operations

**Pointer comparison:** In general, seqptr may try to compare (1) two non-canonical (L1) pointers, (2) two canonical pointers, or (3) a canonical and a non-canonical (L1) pointer. Cases (1) and (2) are simple equality checks, done within the core. For case (3), the core first obtains the L1 pointer’s canonical pointer from the L1 pad, then compares canonical pointers.

**Object allocation:** The alloc instruction allocates small objects in the L1 pad, and large objects in higher levels. In our implementation, an object of size S is allocated as follows:
Large objects are accessed in \textit{subobjects} (6.6). The object’s type id is written to its first word, and other words are zeroed.

6.4.3 Maintaining CE metadata

\textbf{Pointer bits} let CEs work without software intervention. A word’s pointer bit is set if it holds a \textit{pad} pointer—we need not identify main-memory pointers, as CEs do not manipulate them. Pointer bits are set on \texttt{ldptr}, \texttt{stptr}, and \texttt{derefptr}, and are propagated through the hierarchy. They are not stored in main memory, so objects copied from main memory start with pointer bits cleared. This is safe because, by Invariants 2 and 3, they may only be main memory pointers. Pointer bits also ensure integrity for pad pointers (Chapter 5).

\textbf{Recency bits} let CEs select which objects to evict. We use 4-bit coarse-grain LRU timestamps as in [78]. On an access, a \textit{current timestamp} value is written to the object’s recency bits. When \(\frac{1}{8}\)th of the pad’s capacity has been tagged with the current timestamp, the timestamp is increased. We find that this works nearly as well as perfect LRU.

Prior work has proposed higher-performing policies than LRU [21,31,37,71,89,100], but adapting them to Hotpads is not trivial because CEs perform evictions in bulk rather than one line at a time. Adapting the insights behind these policies to Hotpads is interesting future work.

6.5 The collection-eviction (CE) process

When a pad’s free space reaches a low threshold, a collection-eviction (CE) is triggered to free up space. Similar to GCs, CEs traverse the data array to find dead objects. In addition, a CE evicts live but not-recently-accessed objects to the next-level pad. Each CE seeks to free about 75\% of the pad’s capacity (this threshold works consistently well in our experiments).

Invariants 3 and 4 (Section 6.2) enable hierarchical CEs: a pad can perform a CE
without involving larger, higher-level pads. However, pads need to involve lower-level pads in their own CE (e.g., an L2 CE needs help from the L1). We first explain how L1 pad CEAs work, then discuss CEs on higher-level pads.

An engine within the pad performs the CE, which involves similar steps to moving GC: finding roots, marking live objects, compacting or evicting live objects, and updating pointers:

1. **Find roots:** Roots are the pointers outside the pad that point to objects in the pad. The L1 pad’s roots are the L1 pointers currently in the core’s registers, which the core provides.

   Root-finding has negligible cost in Hotpads, a key difference with software GC. Software GCs interrupt each thread and unwind its stack to find roots. This process can take significant time (0.1–1ms [15]), and biases generational GC to use large young heaps (typically as large as the LLC [7,81]). But since Hotpads provides fast allocation, we allocate all data in the heap, including stack frames. Thus, the objects the stack points to are not roots. Instead, they are found in the mark pass.

2. **Mark live objects:** We use a standard tricolor mark pass [20] to find which objects are live and referenced from the L1.

   We use two bits in each object’s first word (**CE mark state** in Fig. 6-2) to mark objects as unscanned, to-scan, or scanned. Canonical objects start unscanned, while roots and copies (Invariant 4) are marked to-scan. The pad iteratively inspects the array and scans each to-scan object: it marks it scanned and promotes all the unscanned L1 objects it points to to to-scan. The process finishes when there are no to-scan objects left: scanned objects are live, and those still unscanned are dead.

   To accelerate this phase, we use a small FIFO of to-scan pointers (16 in our implementation). If the FIFO is not full, objects promoted to to-scan are inserted to it. If the FIFO is not empty, the next object to scan is dequeued from it. If the FIFO is empty, the data array is traversed for to-scan objects.

3. **Compact or evict live objects:** The engine now scans the array, processing every live object. The object is evicted if its recency field shows it’s not in the most-recently-accessed 25% of capacity. Otherwise, the object is moved to the free space. This way, moved objects stay in one compact chunk. Moving an object frees its space, so this process can be bootstrapped with very little free capacity—enough
to fit one object.

During this phase, the controller builds a rename table that, given the old address of a live object, returns the object’s new pointer. This table is kept in the data array. We later describe how to build the rename table without space overheads.

Finally, evictions must preserve Invariant 3: an object may only point to objects in the same or higher levels. Therefore, if an evicted object \( E \) has pointers to other L1 objects, they must be rewritten. For each pointed-to object \( P \), if \( P \)'s canonical level is beyond the L1, \( E \)'s pointer is rewritten to \( P \)'s canonical. However, if \( P \)'s canonical level is L1, then \( P \) is made an L2 canonical object, so that \( E \) can reference \( P \) from the L2, as shown in Fig. 6-6. \( P \) is not evicted from the L1 unless it also is not recently-used.

4. **Update pointers:** Finally, the CE engine traverses all the pointers in the array, querying the rename table to update each old pointer to its new location. The core’s pointers are also updated. Then, the rename table is discarded.

**Concurrent operation:** We use a simple alternating-bit protocol [87] to let CEs and program execution happen concurrently. At the start of a CE, the pad’s controller flips an epoch bit. This epoch bit is embedded in all pointers (Fig. 6-1), allowing to distinguish old vs. new pointers. Mark and pointer updates only apply to old pointers. Finally, if the core accesses an old pointer during the compaction phase, the L1 needs to check that its object has not been moved yet. This check is cheap because the data array is compacted in sequence. If the object has moved, the rename table is accessed to find its new location. This slow path has a negligible performance impact because it happens only on one phase of the CE process.

**Dual-ended compaction:** Enabling large rename tables without space over-
heads. To make lookups cheap, we use a directly-addressed rename table with one pointer per two words of the data array (since objects are at least two words long). This table takes 50% of the pad’s capacity. Because the CE frees about 75% of capacity, we use this free space to hold the rename table, then release it when pointer updates finish.

For this to be efficient, it is crucial that the rename table grows incrementally, as we perform the compaction pass and free space for it (if we had to allocate the full rename table in advance, we could not use more than 50% of the pad’s capacity for objects!). Fig. 6-7 shows how we accomplish this. The key idea is to place the rename table slightly after the end of the old region, then alternate processing objects from the start and end of the old region. Compacting from the start frees space for the new region, and compacting from the end frees space for the rename table. The table is immediately above the new region, and is freed after pointer updates.

![Figure 6-7: Dual-ended compaction example. All objects are 2 words long, except E, which takes 4 words.](image)

CEs at higher-level pads follow the same four steps as the L1, with two key differences. First, root-finding involves traversing all lower-level pads in addition to reading the core’s pointers. Each pad looks for pointers to the level performing the CE, and sends only those as roots. Second, the final update pointers phase also requires lower-level pads in addition to the core to scan and update their pointers. Each pad looks for old pointers to the level performing the CE and requests updated pointers from the level’s rename table.
6.6 Supporting arbitrarily large objects

We have so far assumed that objects have a bounded size, but supporting arbitrarily large objects is useful, e.g., for large arrays. We accomplish this by caching **subobjects** accesses to objects larger than a threshold $SS$ fetch a small subobject of size at most $SS$ into the L1 ($SS = 64$ bytes in our implementation).

Subobjects use pads like caches. One can see each object as a distinct address space, and subobjects as the way to cache it.

To ease code generation, loads and stores to large objects implicitly fetch the right subobject. However, this lowers efficiency because pointers to the full object are not rewritten.

We observe that accessing the same subobject repeatedly through the same register is a common pattern. To avoid the associative lookup costs in this case, we introduce **shadow subobject registers**.

Each register is associated with a shadow subobject register. The shadow subobject register stores a pointer to the subobject that was last accessed using its associated register. On an access, if the offset falls within the same subobject that is stored in the shadow subobject register, Hotpads performs a direct access to the L1 pad instead of an associative lookup. Otherwise, the shadow subobject register is overwritten with a new subobject pointer.

6.7 Object-level coherence

Hotpads is orthogonal to coherence. We implement MESI coherence with three simple changes over the single-core design:

First, we make the first shared pad level inclusive. For example, we use core-private L1 and L2 pads and a fully-shared L3 pad. We make the L3 pad inclusive, so all main-memory objects fetched into L1s are also allocated in the L3.

Second, this shared pad level uses some per-object space to hold the object’s sharer set. We simulate systems of up to 4 cores, so a 4-bit sharer bit-vector suffices. This bit-vector fits in the unused bits of the first word of each object (Fig. 6-2). Systems with more cores could use extra words above each object to store larger bit-vectors.
We leave this to future work.

Third, we repurpose the valid and dirty bits to encode the four coherence states (Modified, Exclusive, Shared, Invalid). Accesses manipulate these bits as in conventional MESI and trigger the same actions. For example, a store to an object in S triggers an upgrade request to the shared pad, which invalidates copies in other pads before granting exclusive permission. We use the same invalidation machinery described in Section 6.4.

This design handles subobjects in the same way (for large objects, coherence is maintained at subobject granularity). Its key advantages are that it does not require a separate coherence directory, and that it avoids false sharing as long as contended data is kept in a separate object. For example, multiple cores may contend on a single-word Lock object, and only a single word will be transferred on coherence actions.

6.8 Instruction pads vs. instruction caches

We have so far ignored instruction fetches. In principle, we could use Hotpads to improve instruction fetch efficiency. For example, if each basic block was its own object, branches could be rewritten to use L1 addresses, making the L1 instruction pad work like a trace cache [77] without associative lookups.

In practice, this approach would require drastic ISA and JIT engine changes, so we leave it to future work. Instead, we treat the code region as a single large object, and fetch it in subobjects. Each core has a conventional L1 instruction cache that accesses subobjects from the L2 pad.

6.9 Crosscutting issues

Banked pads: In multicores, shared pads should be banked to achieve high throughput. We stripe the L3 pad’s address space across banks, then manage each bank as a separate pad. Each bank holds full objects—they are not split across banks.

To keep load balance, each L2 pad evicts L2-canonical objects across L3 banks in a round-robin fashion. We empirically observe that this suffices to keep bank capacity and bandwidth balanced. For objects whose canonical level is the L3 or main memory,
their address directly determines their L3 bank.

Finally, all L3 pad banks perform CEs together, as each bank may hold objects with pointers to other banks.

**Interfacing with main memory:** In our implementation, each L3 bank caches and evicts to a separate region of main memory. Each bank thus holds its own main-memory bump pointer.

Since DDR3/4 impose a minimum burst length of 64B, small objects suffer from overfetching. We add a small cache to the memory controller (8KB in our implementation) to retain overfetched data. Thanks to spatial locality, a small cache avoids a large fraction of overfetch overheads.

In our implementation, we garbage-collect main memory in software, using the same stop-the-world implementation as our baseline JVM. We simply flush the pads and treat main memory as a single large object, accessed in subobjects by the GC thread. Concurrent main-memory GC is also possible, but we leave it to future work. Finally, Hotpads could be generalized to allow managing main memory with other techniques, like reference-counting GC. We also leave this to future work.

**Supporting legacy code:** To ease adoption, Hotpads supports memory-unsafe programs by treating all their memory as a single large object. In this legacy mode, pads are somewhat slower than caches due to the lack of pointer rewriting.

**Virtual memory (VM) and multiple processes:** We only evaluate single-process setups and leave a detailed VM study to future work. However, Hotpads should greatly reduce VM overheads. At the extreme, OSes like Singularity [29] and Verve [101] eliminate the need for VM and rely on verified type and memory safety for process isolation. With a more conventional OS, Hotpads could support partitioning each pad’s capacity into process-private regions, and perform either demand-paging or segmentation of main memory only, with address translation on L3 pad misses.
Chapter 7

Experimental Methodology

We prototype Hotpads using MaxSim [76], a simulation platform that combines ZSim [80], a Pin-based [50] simulator, and Maxine [98], a 64-bit metacircular research JVM.

<table>
<thead>
<tr>
<th>Cores</th>
<th>4 cores, x86-64 ISA, 3.6 GHz, Westmere-like OOO [80]: 16B-wide ifetch; 2-level bpred with 2048×10-bit BHSRs + 4096×2-bit PHT, 4-wide issue, 36-entry IQ, 128-entry ROB, 32-entry IQ, 32-entry SQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>64 KB, 8-way set-associative, split D/I caches, 64 B lines</td>
</tr>
<tr>
<td>L2</td>
<td>512 KB private per-core, 8-way set-associative</td>
</tr>
<tr>
<td>L3</td>
<td>4 banks, 2 MB/bank, 16-way set-associative, LRU replacement</td>
</tr>
<tr>
<td>L1D</td>
<td>64 KB data array, 1K ctag entries (+4KB metadata)</td>
</tr>
<tr>
<td>L1I</td>
<td>64 KB cache, 8-way set-associative, 128 B lines</td>
</tr>
<tr>
<td>L2</td>
<td>512 KB data array, 8K ctag entries (+32KB metadata)</td>
</tr>
<tr>
<td>L3</td>
<td>4×2 MB data array, 4×32K ctag entries (+4×128KB metadata)</td>
</tr>
<tr>
<td>Main mem</td>
<td>2 DDR3-1600 channels, 20 nJ per 64B access [54]</td>
</tr>
</tbody>
</table>

Table 7.1: Configuration of the simulated 4-core system.
7.1 Hardware

We simulate a 4-core processor with a three-level cache or pad hierarchy, using parameters given in Table 7.1.

7.1.1 Core modifications

We use out-of-order cores modeled and validated after Westmere [80]. We encode the Hotpads ISA using x86 opcodes that the JVM does not emit.

Hotpads requires some modifications to the core pipeline. The commit stage includes one pointer bit per register with the same semantics as those in the pads: ldptr, derefptr, and subptr set their destination register’s pointer bit, and other instructions reset their destination register’s pointer bit. To perform CE root-finding, the core flushes and quiesces the pipeline, streams out its pointer registers to the pad starting the CE, and resumes execution. This takes tens of cycles.

Pointer rewriting is performed lazily (at commit time), by updating the physical register directly. Like in the pads, pointer rewrites are not treated like writes: an inflight instruction does not list its pointer register as a destination, and the issue logic can dispatch multiple instructions that use the same pointer register. This may cause the core to issue a few back-to-back loads with a canonical address before the first of such loads rewrites the pointer. This is perfectly safe.

7.1.2 Speculative execution

The L1 pad fetches and allocates objects speculatively, before loads and alloc instructions commit. When mispeculation is detected, the L1 pad simply rolls back its bump pointer, freeing mis-allocated objects. Like pointers, L1 c-tags are updated lazily, at commit time.

7.1.3 Cache scrubbing

We implement cooperative cache scrubbing [81], which adds instructions to zero and scrub cache lines and uses them in the JVM to reduce memory traffic due to object allocation and useless writebacks. In particular, we chose to use the combination of
clzero2 and clinvalidate instructions as our optimized baseline because Sartor et al. find that a combination of zeroing and scrubbing provide the largest reduction in memory traffic. We tested clinvalidate, clzero2, and a combination of both, and find that the combination of clinvalidate + clzero2 was the most effective in reducing overall traffic (both read and write), as we will see in Section 8.8.

7.2 Software

7.2.1 JVM

Our cache-based baseline uses the Maxine JVM with a tuned, stop-the-world generational GC. We set the nursery’s size to 8 MB, which is a reasonable performance choice (this matches prior work [7, 81]).

To model Hotpads correctly, we extend MaxSim in a number of ways. Within Maxine, we instrument several runtime events to use the ABI in Chapter 6. This includes object-level operations, such as object allocations (i.e. the new keyword) and pointer updates. To be able to perform collection-eviction at any point in time, we also record the roots in the static region (JVM, native library) and stack regions by instrumenting modifications to the static region and stack regions.

To ensure our simulations are not distorted (i.e., having more instructions/memory accesses than the uninstrumented version), we extend the intermediate representations (IR) in Maxine’s JIT compiler to generate instrumentation code as special instructions, which are then treated as magic operations in ZSim and have no effect on microarchitectures other than Hotpads. This methodology is similar to the oracular memory management framework used by Hertz et al. [26].

For our Hotpads experiments, we modify Maxine’s JIT compiler to follow the Hotpads ISA. Hotpads performs concurrent CEs in hardware, and non-generational, stop-the-world GC in software when the main-memory heap fills up. We also perform an explicit full garbage collection in the JVM right before entering the region of interest after the application’s warm-up phase.

Comparing stop-the-world young GCs against concurrent CEs is fair. Concurrent GCs seek to reduce pause times by performing GC while the program is running.
They trade off shorter pauses for lower throughput due to synchronization and the additional complexity in coordinating application and GC threads [3, 20, 46, 88]. Since the length of the extra execution time caused by GC activities is greater for concurrent GC due to its lower throughput, comparing the execution time between concurrent CE and stop-the-world generational GC algorithm is fair.

### 7.2.2 Workloads

We study 13 Java workloads: 10 from the Dacapo [8] suite, SPECjbb2005 [85], and the PageRank and Coloring graph processing workloads from JgraphT [62], a popular Java graph library. Table 7.2 describes their input sets.

For each workload, we first find the smallest heap size that does not crash (to the closest MB), and use twice that size. This is standard practice and it reflects a moderate heap pressure on the application [9, 81]. Table 7.2 also details the heap sizes we use for each benchmark. We fast-forward JVM initialization and warm-up the JIT compiler similar to prior work [8] before starting simulation.

<table>
<thead>
<tr>
<th>Apps</th>
<th>Suite</th>
<th>Input</th>
<th>Configured heap size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>lusearch-fix</td>
<td>Dacapo</td>
<td>default</td>
<td>54</td>
</tr>
<tr>
<td>lusearch</td>
<td>Dacapo</td>
<td>default</td>
<td>56</td>
</tr>
<tr>
<td>hiindex</td>
<td>Dacapo</td>
<td>default</td>
<td>84</td>
</tr>
<tr>
<td>sunflow</td>
<td>Dacapo</td>
<td>default</td>
<td>90</td>
</tr>
<tr>
<td>xalan</td>
<td>Dacapo</td>
<td>default</td>
<td>194</td>
</tr>
<tr>
<td>fop</td>
<td>Dacapo</td>
<td>default</td>
<td>232</td>
</tr>
<tr>
<td>batik</td>
<td>Dacapo</td>
<td>default</td>
<td>344</td>
</tr>
<tr>
<td>jython</td>
<td>Dacapo</td>
<td>default</td>
<td>572</td>
</tr>
<tr>
<td>pmd</td>
<td>Dacapo</td>
<td>default</td>
<td>764</td>
</tr>
<tr>
<td>h2</td>
<td>Dacapo</td>
<td>default</td>
<td>1402</td>
</tr>
<tr>
<td>coloring</td>
<td>JgraphT</td>
<td>amazon-2008 graph</td>
<td>9624</td>
</tr>
<tr>
<td>pagerank</td>
<td>JgraphT</td>
<td>amazon-2008 graph</td>
<td>9784</td>
</tr>
<tr>
<td>specjbb</td>
<td>SpecJBB</td>
<td>1 warehouse per thread,50K transactions</td>
<td>264</td>
</tr>
</tbody>
</table>

Table 7.2: Java workloads, inputs used and configured heap size. The minimum heap size (to the closest MB) for each application was found using a generational GC scheme with 8MB young generation. Applications were configured with twice their minimum heap size to reflect moderate heap pressure.

---

1We use the most recent release (9.12-MR1-bach), excluding the benchmarks with significant simulation errors as reported by MaxSim’s authors [76].
Chapter 8

Evaluation

8.1 Latency, energy, and area of caches vs. pads

Table 8.1 reports the latency, dynamic energy, leakage, and area for both caches and pads. We use CACTI 6.5 [59] to derive these figures. We extend CACTI to model the pads in detail. We optimize the L1s for delay, using parallel tag and data accesses for L1 caches. L2s and L3s are optimized to minimize energy-delay-area product. Their SRAM cells use low-leakage transistors, and L2 and L3 caches perform serial tag and data accesses. This is a commonly-used methodology [41,78,103], and cache figures agree with prior work [27,36,41].

<table>
<thead>
<tr>
<th>Type</th>
<th>Latency (cycle)</th>
<th>Energy (pJ)</th>
<th>Area (mm²)</th>
<th>Leakage (mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Direct Hit Miss</td>
<td>Direct Hit Miss</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>L1</strong></td>
<td>Cache</td>
<td>– 2 1</td>
<td>– 74 250</td>
<td>0.26</td>
</tr>
<tr>
<td></td>
<td>Pad</td>
<td>2 3 1</td>
<td>16 27 194</td>
<td>0.28</td>
</tr>
<tr>
<td><strong>L2</strong></td>
<td>Cache</td>
<td>– 9 2</td>
<td>– 371 402</td>
<td>1.53</td>
</tr>
<tr>
<td></td>
<td>Pad</td>
<td>7 9 2</td>
<td>348 378 30</td>
<td>1.55</td>
</tr>
<tr>
<td><strong>L3</strong></td>
<td>Cache</td>
<td>– 14 5</td>
<td>– 742 795</td>
<td>21.46</td>
</tr>
<tr>
<td></td>
<td>Pad</td>
<td>9 14 5</td>
<td>655 771 828</td>
<td>21.88</td>
</tr>
</tbody>
</table>

Table 8.1: Latency, energy, area, and leakage of different cache/pad organizations. We model 8-byte (1-word) L1 accesses, and 64-byte objects for pad misses and fills.

Table 8.1 shows that pads are slightly larger than caches (2% overall area overhead), and have slightly higher leakage, owing to their extra state. However, direct accesses to pads, i.e., those that do not check the c-tags, are faster and substantially more
efficient. This difference is largest in the L1: a direct L1 pad access consumes 16pJ, 4.3× less than an L1 cache hit. As we will see, direct accesses are the common case in the L1. Table 8.1 shows smaller differences for higher levels, but Table 8.1 assumes 64-byte objects. In practice, L2 and L3 pads transfer fewer words than caches, which improves efficiency.

8.2 Hotpads outperforms traditional hierarchies

We first analyze single-threaded workloads. Multi-threaded results, which are similar, are discussed later (Section 8.6).

**Performance:** Fig. 8-1a compares the end-to-end runtime of different schemes, and includes the contributions of application (i.e., non-GC) work and GC overheads (lower is better). In addition to Baseline, Scrubbing, and Hotpads, we also evaluate a Perfect memory system, where all memory accesses take one cycle and no GCs are ever triggered. This unimplementable memory system serves as an upper bound.

Overall, Hotpads outperforms the baseline by 41% on average and by up to 2.2× (on **coloring**). These gains come from both reducing GC overheads, which take 17% of time on average on the baseline vs. 1.2% on Hotpads, and reducing application runtime by 16% due to higher memory performance.

By contrast, Scrubbing outperforms the baseline by only 13%. Scrubbing reduces application runtime because it (i) allocates new objects directly in caches instead of fetching their unused space from main memory, and (ii) avoids writing back the cache lines of dead objects. However, Scrubbing does not accelerate GC, so Hotpads outperforms Scrubbing by 25%.

Focusing on GC overheads, most time is taken by young GCs, as full GCs happen rarely. Hotpads’s CEs eliminate young-GC overheads, reducing overall GC cost by 14.3×. Although young-GC overheads are larger than full-GC overheads, our software schemes use a resonable-performing young heap size, 8 MB (the L3 size), which matches prior work [7,81]. Larger young heaps increase young GC costs due to higher main memory traffic, and smaller young heaps make full GCs more frequent and expensive.

Finally, the perfect memory system improves performance by 65% over the baseline. Hotpads thus bridges 63% of the performance gap between the baseline and a perfect
Figure 8-1: Simulation results for single-threaded workloads.
memory system, whereas Scrubbing bridges 20% of the gap.

**Memory energy and AMAT:** To give more insight into these results, Fig. 8-1b shows the breakdown of dynamic energy in the memory hierarchy, and Fig. 8-1c shows the average memory access time (AMAT) for application work, i.e., excluding GC overheads.

Hotpads reduces the memory hierarchy’s dynamic energy by $2.8\times$ over the baseline, due to three major factors. First, L1 (instruction and data) dynamic energy is $2.3\times$ smaller because L1 pads receive mostly direct accesses (Section 8.4), making them much more efficient than L1 data caches. Second, Hotpads reduces main memory energy by $3.7\times$. Third, Hotpads CEs take $4.5\times$ less energy than software GCs.

By contrast, Scrubbing reduces dynamic energy by 21% over the baseline, chiefly by reducing main memory traffic. Hotpads consumes $2.3\times$ less energy than Scrubbing.

Fig. 8-1c shows that Hotpads’s L1 pad efficiency comes at a slight cost in AMAT: L1 latency is 6% higher due to the longer latency of L1 accesses that require a c-tag lookup (which take an extra cycle over caches, see Table 8.1). As a result, Hotpads’s AMAT is 3% lower than the baseline’s and is only 6% higher than Scrubbing’s AMAT.

**Differences across applications:** Fig. 8-1 shows the execution time for each application. Hotpads’s benefits vary across them.

Applications such as `lusearch`, `xalan`, `lusearch-fix`, allocate many short-lived objects that fit in on-chip pads. Hotpads collects them before they reach main memory, and thus nearly eliminates main memory traffic and enjoys minimal GC energy. By contrast, the baseline and Scrubbing still incur main memory traffic because contention from code and non-heap data evict part of the young heap to main memory.

Applications such as `fop`, `specjbb`, `pmd` have a mix of short- and long-lived objects. Hotpads’s CEs evict long-lived objects to main memory, incurring some main memory traffic, although much less traffic than the baseline and Scrubbing, as short-lived objects are collected on-chip.

The graph applications (`pagerank`, `coloring`) have large, long-lived data structures that reside in main memory. Hotpads’s speedups mostly come from reducing GC overheads, but main memory traffic is only slightly lower than for the cache-based schemes.
8.3 Hotpads reduces data movement across the memory hierarchy

Fig. 8-2 shows the read and write traffic in bytes for each level, averaged across all apps and normalized to the baseline’s. L1 reads and writes are due to loads and stores; L2+ reads are due to object or line fetches, and writes are due to evictions of dirty data from lower levels. Hotpads saves significant traffic beyond the L1, up to $6.7 \times$ in main memory, while scrubbing only saves 68%. These savings stem from two Hotpads features. First, Hotpads moves objects rather than cache lines. Smaller objects improve pad utilization, leading to fewer misses, and reduce the amount of data transferred per miss. Second, CEs collect dead objects quickly, which reduces write traffic.

Object lifetime analysis: Fig. 8-3 shows the number of object bytes that are allocated or evicted into each level, and the number of object bytes that die at each level. Most of the data is allocated and dies in the L1 pad. Therefore, Hotpads only needs to evict a small portion of allocated bytes to larger levels, and only 10% reaches main memory. This explains Fig. 8-2’s drastic reduction in write traffic beyond the L1.

Data array utilization: Hotpads uses on-chip capacity more efficiently than caches. We define utilization as the ratio between the number of accessed words and the number of total words brought or allocated in the cache or pad. In the L1, Hotpads achieves 35% utilization across all benchmarks, while the baseline and Scrubbing achieve 31% and 33% utilization. L2s and L3s show similar differences.

8.4 Pointer rewriting avoids most associative lookups

Fig. 8-4 shows the fraction of direct accesses, c-tag hits, and c-tag misses for all Hotpads levels. The number above each bar is the fraction of total accesses that reach this level (L1=100%, as all accesses start at the L1).

Pointer rewriting is highly effective, turning 80% of the L1 pad accesses into direct accesses, which require no c-tag lookup and consume the least energy. This explains why L1 pads consume far less energy than L1 caches (Fig. 8-1b), and why they only
incurs a small AMAT penalty (Fig. 8-1c).

Pointer rewriting only works at the L1, so larger pads have a lower fraction of direct accesses (only objects whose canonical level is that pad see direct accesses). However, because the L1 filters most accesses, the fraction of L2 and L3 direct accesses has a small impact on overall energy consumption.

### 8.5 CEs are fast and infrequent

Table 8.2 shows the duration and frequency of CEs on all pads, averaged across all applications. CEs are short and are active for a small fraction of cycles at all levels. While smaller pads have more frequent CEs, each CE is also very cheap (e.g., L1 CEs...
<table>
<thead>
<tr>
<th>Pad level</th>
<th>L1</th>
<th>L2</th>
<th>L3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg. CE length (cycle)</td>
<td>5.1K</td>
<td>251K</td>
<td>5.7M</td>
</tr>
<tr>
<td>Avg. CE interval (cycle)</td>
<td>128K</td>
<td>3.9M</td>
<td>237M</td>
</tr>
<tr>
<td>Active ratio</td>
<td>3.99%</td>
<td>6.29%</td>
<td>2.41%</td>
</tr>
</tbody>
</table>

Table 8.2: CE duration and frequency across pads.

Fig. 8-5 shows distributions (CDFs) of L1 pad CE length and interval between consecutive CEs for three representative applications. Two main factors determine the length of each CE. First, applications with more live pad capacity have longer CEs, as evictions dominate CE time. For example, lusearch-fix has short L1 CEs (<5Kcycles) because a lot of its data dies in the L1 pad, while the other two applications have longer CEs. Second, the pad gives priority to demand accesses over CE accesses, so CEs take longer on applications with more frequent pad accesses. We also observe that the interval between CEs depends on the allocation rate. For example, lusearch-fix has shorter inter-CE intervals (<100Kcycles) than the other applications. Finally, these CDFs show that, although there is variability among CEs, even in the worst case (longest CE and shortest inter-CE interval), CEs are only active for a small fraction of the time.

are about 1000× more frequent and cheaper than L3 CEs).
8.6 Hotpads performs well on multithreaded workloads

Fig. 8-6 shows the runtime and energy breakdowns for multithreaded workloads. The key difference is that GC overheads are larger because Maxine’s GC is not parallel. This is a hard-to-address limitation of Maxine. We expect that a parallel GC would achieve similar GC overheads as the single-thread results. Due to serial young GCs, Hotpads improves performance further (by 67%). Hotpads reduces non-GC runtime similarly to single-thread results, and also achieves a similar energy reduction over the baseline (2.7×).

8.7 Legacy mode incurs small performance overheads

Hotpads runs legacy code at modest overheads. We run SPEC CPU2006 applications using the legacy mode (Section 6.9) in Hotpads. Hotpads is 4% slower than a cache hierarchy on average (up to 14% on xalancbmk). This slowdown stems from the fact that in this mode Hotpads caches subobjects from a single main-memory object and does not rewrite pointers.

8.8 Cache scrubbing validation

Cache scrubbing seeks to reduce the memory traffic and improve the overall performance of memory-safe languages running on a cache hierarchy (Section 3.1). We
implemented the cache scrubbing instructions as explained in Sec. 7.1.3 and tested our implementation for clzero2, clinvalidate, and clzero2 + clinvalidate.

The general trend that we observed matches that in [81]. As seen in Fig. 8-8, clinvalidate reduces memory write traffic. The savings from clinvalidate are proportional to the number of young generation GCs that the application incurs.

Fig. 8-7 shows that clzero2 drastically reduces memory read traffic. This reduction is proportional to the amount of data that the application allocates. clinvalidate
increases the read traffic slightly because invalidated cache lines have to be fetched in from main memory.

The reduction in read and write traffic is the greatest for clzero2 + clinvalidate. These memory traffic savings not only translate into energy savings, but also translate into significant performance speedup as seen in Fig. 8-9.

### 8.9 Effects of young generation size

We examine the effects of changing the young generation capacity, while holding the total heap size per application constant for both the cache-based baseline and also the cache scrubbing baseline. Each application is given a fixed heap size (twice its minimum heap size), as shown in Table 7.2. We sweep the size of the young generation, which also changes the size of the old generation.

Fig. 8-10 shows how the execution time, main memory traffic, and energy change as the young generation size grows from 4MB to 32MB across all 13 applications. Fig. 8-11 to Fig. 8-13 show the same metrics for lusearch, xalan, and coloring.

Increasing the young generation size leads to a decrease in execution time for many applications, such as h2, xalan (Fig. 8-12), and coloring (Fig. 8-13). This matches the trend in Fig. 8-10 as the young generation size is increased from 4MB to 16MB. Increasing the young generation size reduces the number of young generation GCs.
Figure 8-11: Young generation size sweep for lusearch. lusearch experiences a spike in the 32MB datapoint because the pressure on the small old generation leads to increased number of old generation GCs.

Figure 8-12: Young generation size sweep for xalan. xalan benefits from having a larger young generation size because the benefits it gets from reduced young generation GCs outweights the increased cost due to loss of cache locality.

Figure 8-13: Young generation size sweep for coloring. coloring benefits from having a large young generation size.
Hence the time spent doing GCs for the baseline (the orange portion of the execution time graphs) decreases.

However, the increase in young generation capacity may reduce cache locality. Since new objects are allocated in the young generation, we reap cache locality benefits when more of the young generation address range can be kept in the cache. When the size of the young generation increases, it is more likely that the young generation footprint cannot be kept in the cache, so an access or allocation would lead to a miss to fetch from main memory. As a result, we see in Fig. 8-10b that increasing young generation size on average increases the application memory read traffic (the blue portion of the memory traffic graph) and the application write traffic (the orange portion of the memory traffic graph).

Overall, increasing the young generation size reduces the energy spent on GC since the number of young GCs decreases with larger young generation sizes (the orange portion of the bar in the energy graphs). However, the loss of cache locality with larger young generation sizes also results in increased energy spent in main memory (the brown portion of the bar in the energy graphs). The overall effect on energy depends on which effect is greater. For example, xalan (Fig. 8-12) benefits more from the reduced number of GCs than it is harmed by the loss of cache locality, so it favors a larger young generation size.

The exceptions to these trend are notably lusearch (Fig. 8-11) and lusearch-fix. They have relatively small heap sizes (56MB and 54MB, respectively). When given 32MB young generation size, that leaves these applications with small old generations. The old generation fills up at a faster rate with each young generation GC, which results in more old generation semispace GCs. These old generation GCs are more costly and significantly increase the execution time for these applications. lusearch has very few old generation GCs with smaller young generation size, but the number of old generation GCs increases drastically with 32MB young generation. These effects cause the spike on the 32MB datapoint in Fig. 8-10.

Fig. 8-14 shows the gmean of the ratio of cache scrubbing to the baseline for each of the performance metrics, summarizing the improvements of cache scrubbing over the baseline for a given young generation size. With a young generation size of 8MB, cache scrubbing provides approximately 10% improvement in execution time,
Table 8.3: The average improvement that Hotpads has over the various baseline schemes.

<table>
<thead>
<tr>
<th></th>
<th>8MB baseline</th>
<th>8MB scrubbing</th>
<th>16MB baseline</th>
<th>16MB scrubbing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution time</td>
<td>41%</td>
<td>25%</td>
<td>35%</td>
<td>16%</td>
</tr>
<tr>
<td>Dynamic energy</td>
<td>2.8×</td>
<td>2.3×</td>
<td>2.7×</td>
<td>2.2×</td>
</tr>
<tr>
<td>Memory traffic</td>
<td>6.7×</td>
<td>4.0×</td>
<td>6.4×</td>
<td>3.9×</td>
</tr>
</tbody>
</table>

45% improvement in main memory traffic, and 20% improvement in energy. Cache scrubbing provides the most benefits with either 8MB or 16MB. Table 8.3 summarizes the improvements that Hotpads achieves over the baseline and cache scrubbing schemes with 8MB young generation size and 16MB young generation size.

8.10 Sensitivity studies

We conducted several sensitivity studies to find the best configuration for testing the baseline and the best parameters for Hotpads. These studies are conducted using small inputs. Since the behavior of the small inputs and default inputs are similar, the results of the sensitivity studies are still applicable to default inputs.

8.10.1 Allocation threshold for Hotpads’s L1 pad

The allocation threshold affects the distribution of where objects are initially allocated. If the size of an object is greater than the threshold for that pad level, it is allocated on the next pad level. When the threshold is lower, fewer objects are allocated in the L1 pad and more objects have to be copied into L1 pad on an access, increasing data movement. Conversely, if the threshold is high, too many objects are allocated in the
L1 pad, increasing the frequency of the L1 CE and potentially prolonging objects’ lifetimes by evicting them before they die.

From Fig. 8-15, we see that with a higher threshold, more objects are allocated in and die within the L1 pad, consistent with the weak generational hypothesis. The number of CEs for each application does not change that much with the allocation threshold as seen in Fig. 8-16. However, the number of bytes transferred from the L1 (the blue portion of the bars in Fig. 8-17) changes significantly with the
Figure 8-17: How the number of bytes that are moved from a level during application runtime (non-GC) changes with the L1 pad allocation threshold.

Figure 8-18: How the average execution time, memory traffic, and number of CEs change as either cache line size or subobject threshold increases.

allocation threshold. This figure represents the bytes evicted from a pad level during the application’s run outside of the CE process. These results show that, for these applications, 512B is the overall best choice for the L1 allocation threshold. As the L1 pad is used the most in Hotpads, we focus on finding the right allocation choice for the L1 pad and not other pad levels.
8.10.2 Baseline’s cache line size and Hotpads’s subobject threshold

We hold the cache configuration constant as in Table 7.1 and sweep the cache line size to study the effects of cache line size on different performance metrics. We also conduct a similar set of experiments for Hotpads’s subobject threshold. The subobject threshold determines the size at which we switch to copying a subobject instead of the entire object between levels. For example, if the subobject threshold is 64B, and an access was issued to a field in a 1MB object in main memory, only 64B of that object would be copied to the L1 pad and accessed. Fig. 8-18 shows, on average, how the execution time, main memory traffic, and number of CEs (only for Hotpads) change as either the cache line size or subobject threshold is increased across the 13 applications.

**Execution time:** Execution time initially decreases as the cache line size or subobject threshold increases but eventually increases. The initial decrease is because the application is benefitting from fetching in more bytes per miss, reducing the number of misses. However, as cache line increases further, each fetch causes the cache capacity to fill up even faster, leading to more evictions. The minimal execution time for each application occurs at different cache line sizes. For example, sunflow and lusearch-fix favor the 512B cache line size, whereas pagerank’s minimum execution time occurs with 256B cache line size.

Similar to the baseline cache line size sweep, for Hotpads’s subobject threshold sweep the execution time initially decreases, hits a minimum point, and then increases again. The minimum average execution time is around the 256B subobject threshold. However, the improvement in execution time (Fig. 8-18a) over 32B subobject threshold is not as large as the baseline’s improvement over a 32B cache line. For Hotpads, the subobject threshold affects a smaller percentage of accesses to objects whereas the cache line size affects all accesses. Objects that start out and die in the L1 pad are not affected by the subobject threshold since they never need to be copied in from a larger level. Furthermore, objects that fall below the subobject threshold are always entirely copied into the L1 pad, so further increases in subobject threshold affect even smaller percentage of accesses to objects.

**Main memory traffic:** The other metric to consider is the amount of memory traffic
in the applications as cache line size changes. The larger the cache line size, the
greater the amount of memory traffic. This is because each miss now fetches in more
bytes than is required.

Hotpads’s main memory traffic shows a similar trend to that of the cache line size
sweep, increasing with the subobject threshold. As Hotpads allocates new objects in
cache, fewer objects reach main memory, therefore there is a smaller increase in main
memory traffic compared to that of the baseline.

**Effect on CEs:** Increasing the subobject threshold directly increases the rate at
which each of the pad levels fill up, which in turn increases the frequency of CEs for
Hotpads (Fig. 8-18c). L1 pad CEs experience the largest increase in magnitude, but
generally L2 pad CEs have the largest percentage increase. With the increased GC
frequency, sometimes we may evict an object that is just about to die to the L2 pad,
which in turn, also increases the number of L2 pad CEs. The number of GCs that the
baseline incurs does not change with the cache line size.
Chapter 9

Future Work and Conclusion

Beyond our specific implementation, Hotpads opens up exciting new avenues in many aspects of memory systems that we leave to future work. These include:

- **Security**: Since Hotpads has no caches and hides addresses, it should effectively avoid cache timing side channels [66] that underpin the recent Spectre [40] and Meltdown [47] attacks. A secure Hotpads implementation might need to close other side channels, e.g., randomizing when CEs happen.

- **Isolation**: Hotpads may reduce or eliminate VM overheads (Section 6.9), e.g., by segmenting shared pad capacity among processes. Beyond functional isolation, this would provide performance isolation much more cheaply than cache partitioning, which has considerable overheads [70, 79, 96].

- **Hierarchy management**: How should Hotpads leverage the insights that prior work has developed to manage caches? For example, how to adapt recent replacement policies to bulk evictions (Section 6.4)? Could we perform locality-aware level selection and bypass [37, 91] for new and fetched objects? Could we rearrange objects in pads to facilitate prefetching?

- **Concurrency and non-volatility**: Hotpads need not overwrite old copies of objects on an eviction or invalidation, making it possible to have pads act as log-like multiversioned stores, which could be used to implement transactional memory [22, 58] or accelerate NVM logging [30, 65, 102].

In conclusion, we have shown that the key insight behind memory-safe languages, hiding the memory layout, can be applied to design efficient memory hierarchies.
Hotpads outperforms cache hierarchies because it moves objects rather than lines, avoids most associative lookups, and greatly reduces GC overheads. Hotpads lights the path to future memory systems that support the needs of modern programs.
Bibliography


[64] Oracle, “Memory Management in the Java HotSpot Virtual Machine.”


