2018 51st Annual IEEE/ACM International Symposium on Microarchitecture

Rethinking the Memory Hierarchy for Modern Languages

Po-An Tsai, Yee Ling Gan, Daniel Sanchez
Massachusetts Institute of Technology
{poantsai, elainegn, sanchez}@csail.mit.edu

***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 and enables automatic memory management.**

Hotpads extends the same insight to the memory hierarchy: it hides the memory layout from software and takes control over it, dispensing with the conventional flat address space abstraction. This avoids the need for associative caches. 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.

***Index Terms*—Memory hierarchy, cache, scratchpad, memory-safe languages, managed languages, garbage collection.**

I. Introduction

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.

Fortunately, languages have changed. All 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.

Prior work has sought to *bridge the semantic gap* between memory-safe languages and architectures by accelerating common operations, such as type checks or object-based addressing [20, 96] and protection [32, 102] (Sec. II). But they do so within a conventional memory hierarchy. By contrast, in this work we *redesign the memory hierarchy to cater to memory-safe languages*. This avoids many of the overheads of conventional hierarchies and unlocks new optimizations.

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 [54]) that may only be dereferenced or compared. In software, hiding the memory layout enables automatic memory management, i.e., garbage collection (GC).

We present Hotpads, a novel memory hierarchy that extends this insight to hardware (Sec. III). Hotpads hides the memory layout and takes control over it, dispensing with the flat address space abstraction. The Hotpads ISA (Sec. IV) prevents programs from reading or manipulating raw pointers, enabling Hotpads hardware to 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 GC (Sec. V). 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 have pointers to objects beyond the L1 pad. When the program dereferences each of these pointers, *hardware automatically rewrites the pointer* with the 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, and evicts live (i.e., referenced) but non-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 [95]. 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 (Sec. VI). Hotpads substantially outperforms cache hierarchies for three key reasons (Sec. VII). 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 2.3× more efficient than L1 caches. Third, CEs dramatically reduce GC overheads, by 8× on average. Overall, Hotpads improves performance by 34% and reduces memory hierarchy energy by 2.6×. 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 (Sec. IX). We leave these and other techniques to future work.

II. Background on Memory-Safe Languages

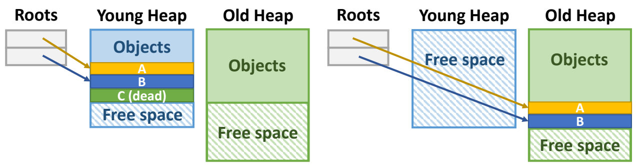
It is the right time to redesign memory systems for memory-safe (a.k.a. managed) languages. These languages rely on a combination of type systems, static analysis, and runtime checks to prevent programs from manipulating memory directly. They prevent large classes of bugs, such as buffer overflows, and enable garbage collection. Nearly all languages introduced in the last 30 years are memory-safe, such as Java, Go, and Rust. Although many applications and most operating systems are still written in C/C++, several projects like the Singularity OS [34] Verve [104], and Redox [75] 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 (Sec. V-I).

Much prior work has focused on *bridging the semantic gap* between memory-safe languages and architectures. Object-oriented systems [19, 20, 62, 68, 96] reduce virtual call overheads and accelerate object references. Capability-based systems [21, 32, 48, 102] provide object-based memory protection and isolation. Typed architectures [2, 16, 43, 44, 88] 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*.

Garbage collection (GC), also known as automatic memory management, frees programmers from manually freeing memory. Instead, the system automatically reclaims memory occupied by dead (i.e., unreferenced) objects. There are two main types of GC: *tracing* [57] and *reference counting* [18]. Though both styles have pros and cons [7, 56], 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 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.

Tracing algorithms can be moving or non-moving [100]. 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. Hotpads performs a process similar to moving GC to achieve a compact layout with variable-sized objects.



(a) Before young heap GC. (b) After young heap GC.
Fig. 1: Generational GC example.

Prior work has proposed many techniques to reduce GC overheads [4, 6, 9, 12, 71]. We focus on two dimensions: generational and concurrent GC.

Generational GC algorithms exploit the *generational hypothesis*, the empirical observation that most objects die young [95]. They use separate heaps for objects of different ages. Fig. 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 [33, 60, 72, 90].

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, 22, 51, 92]. 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 [51] 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*, Sec. VI-A).

Hardware techniques to accelerate GC date back to Lisp machines [61]. Recent work includes HAMM [38], which accelerates reference counting to reduce young heap GC overheads. Cooperative cache scrubbing [82] 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, 58, 76, 92].

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.

III. Hotpads Overview

Fig. 2 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. 2 shows a hierarchy with three levels of pads, but Hotpads supports an arbitrary number of levels.

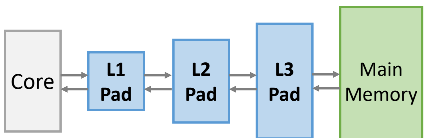


Fig. 2: Hotpads is a hierarchical memory system with multiple levels of *pads*.

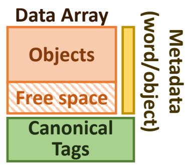


Fig. 3: Pad organization.

Pads: Fig. 3 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. 3 also shows that pads have some *metadata* (e.g., to record whether each word holds a pointer) and a *canonical tags* array, which is a decoupled tag store like the V-way cache [74]. We will later see how these auxiliary structures are used.

Key features: As explained in Sec. I, 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 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

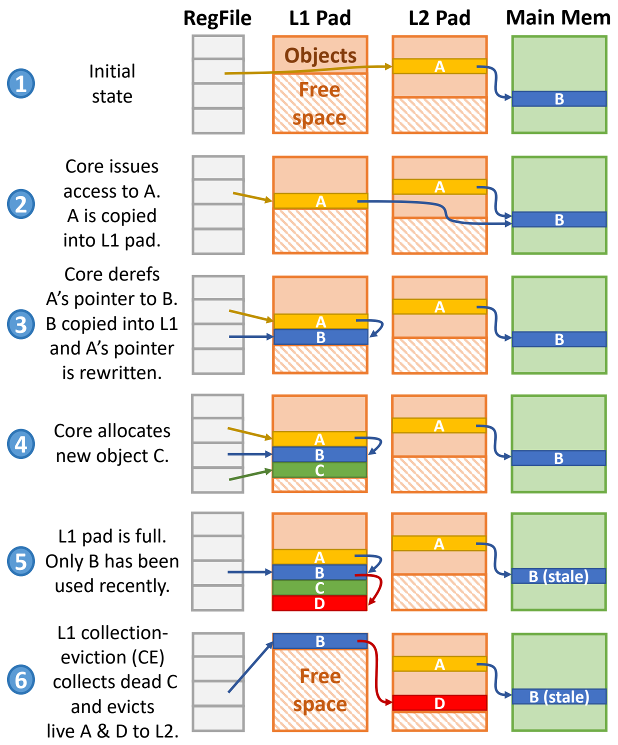

RegFile
L1 Pad
L2 Pad
Main Mem
1
Initial state
Objects
Free space
A
B
2
Core issues access to A. A is copied into L1 pad.
3
Core derefs A's pointer to B. B copied into L1 and A's pointer is rewritten.
4
Core allocates new object C.
C
5
L1 pad is full. Only B has been used recently.
D
B (stale)
6
L1 collection-eviction (CE) collects dead C and evicts live A & D to L2.

Fig. 4: Example showing Hotpads's key features.

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 non-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. As in moving GC, live objects are compacted into a contiguous region to simplify free 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* (Sec. V-B). For example, an L2 object may point to objects in the L2 or main memory, but not to L1 objects.

IV. Hotpads ISA: Hiding the memory layout

Hotpads treats pointers as abstract data types [54] 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 I.

Addressing discipline: Hotpads uses a single addressing mode, base+offset, where *the base register is always an object pointer*. As shown in Table I, 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 I). 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 I). 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.

`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 I). 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 and be at least two words long (our implementation uses 64-bit words).
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 `ldptr`, `stptr`, and `derefptr` to identify them. Reading the first word of an object (i.e., `ld rd, 0(rb)`) returns its type id; writing the first word of an object (i.e., `st rd, 0(rb)`) is illegal and causes exception.

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

TABLE I: Hotpads ISA.

Instruction	Format	Operation
Data Load	`ld rd, disp(rb)`	`rd <- Mem[EffAddr]`
Data Store	`st rd, disp(rb)`	`Mem[EffAddr] <- rd`
Pointer Load	`ldptr rp, disp(rb)`	`rp <- Mem[EffAddr]`
Pointer Store	`stptr rp, disp(rb)`	`Mem[EffAddr] <- rp`
Pointer Dereference	`derefptr rp, disp(rb)`	`rp <- Mem[EffAddr];` `brings object in L1`
Pointer Equality	`seqptr rd, rp1, rp2`	`rd <- (rp1==`$_p$`rp2)? 1:0`
Allocation	`alloc rp, rs1, rs2` (`rs1` = size) (`rs2` = type id)	`NewAddr <- Alloc(rs1);` `Mem[NewAddr] <- rs2;` `rp <- NewAddr;`

`rd`/`rs` denote registers that hold data; `rp`/`rb` hold a pointer. All memory accesses use base+offset addressing. The base is always an object pointer. 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`.

Code addresses: For simplicity, Hotpads treats the code segment as a single object and does not hide code addresses (Sec. V-H). 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 [104]. Alternatively, Hotpads could track main-memory object metadata to prevent these bugs in hardware (Sec. V-I).

V. 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 Sec. V-F onwards.

A. *Pointer format*

Fig. 5 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.

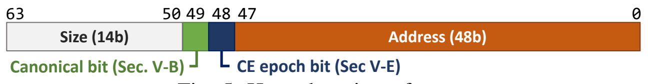


Fig. 5: Hotpads pointer format.

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.

B. 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 ④ , **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. 5) 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 (Sec. V-D). 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 (Sec. V-E).

C. Pad organization

As we saw in Sec. III, each pad consists of a *data array*, a canonical tags (*c-tags*) array, and some per-object and per-word *metadata* (Fig. 3). We now describe each of these components.

Fig. 6 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.

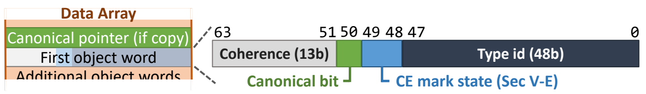


Fig. 6: In-pad object format.

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. 7 shows the format of each c-tag entry. The c-tags array is similar to the V-Way cache's decoupled tag array [74], but it maps objects rather than cache lines to data array addresses.

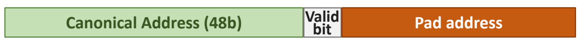


Fig. 7: Canonical tag entry format.

Only non-canonical objects (i.e., copies) 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. 8 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.

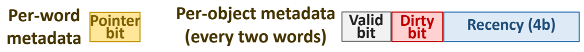


Fig. 8: In-pad metadata format.

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.

D. Steady-state operation

We first explain Hotpads's operation in steady state, i.e., between executions of the CE process. Sec. V-E explains CEs.

D.1. Performing memory accesses

L1 pad accesses: A request from the core includes both base (object pointer) and offset (word within the object). Fig. 9 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.

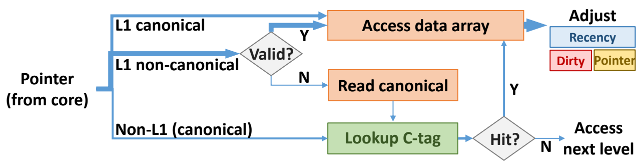


Fig. 9: Steps in L1 pad access. Wider arrows denote more frequent events.

If the pointer is not L1, which implies it must be canonical (by Invariant 2), the L1 pad first checks the c-tags to see if it has a copy of the object. On a c-tag hit, the L1 pad obtains the copy's L1 address and proceeds as above. On a c-tag miss, the L1 pad requests the object from the L2 pad. When it receives the object, it copies it into the L1 data array (using some free space), adds a c-tag entry, and performs the access.

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 ② → 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 only happen when a c-tag array entry needs to be removed. Multicore systems extend invalidations to support coherence (Sec. V-G).

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 (Sec. V-G). Pointer stores do set the dirty bit.

Although rewriting only happens in the L1 pad, Hotpads saves significant energy because workloads generally enjoy high L1 hit rates, and associativity overheads are higher in L1 pads because each access fetches a smaller amount of data (a single 64-bit word vs. an entire object in higher-level pads).

D.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:

$S \leq 512\,\text{B}$	$512\,\text{B} < S \leq 4\,\text{KB}$	$4\,\text{KB} < S < 128\,\text{KB}$	$S \geq 128\,\text{KB}$
↪ L1 pad	↪ L2 pad	↪ L3 pad	↪ Main mem

Large objects are accessed in *subobjects* (Sec. V-F). The object's type id is written to its first word, and subsequent words are zeroed.

D.3. Maintaining CE metadata

Pointer bits let CEs work without software intervention. A word's pointer bit is set if it holds a *pad* pointer—we need not identify main-memory pointers, as CEs do not manipulate them. Pointer bits are set on `ldptr`, `stptr`, and `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 (Sec. IV).

Recency bits let CEs select which objects to evict. We use 4-bit coarse-grain LRU timestamps as in [79]. On an access, a *current timestamp* value is written to the object's recency bits. When $1/8^{th}$ 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 [23, 36, 42, 74, 93, 103], 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.

E. 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 non-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 (Sec. V-B) 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 CEs 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 (0.1–1 ms [17]), and biases generational GC to use large young heaps (typically as large as the LLC [7, 82]). But since Hotpads provides fast allocation, *we allocate stack frames in the heap*. Because the stack is part of the heap, CEs need not to traverse it separately like software GCs do.

2. Mark live objects: We use a standard tricolor mark pass [22] to find which objects are live and referenced from the L1.

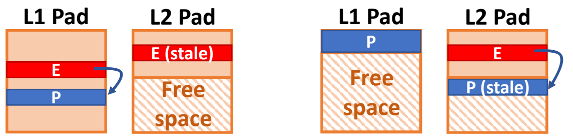
We use two bits in each object's first word (**CE mark state** in Fig. 6) to mark objects as *unscanned*, *to-scan*, or *scanned*. Canonical objects start *unscanned*, while roots and copies (Invariant 4) are marked *to-scan*. The pad iteratively inspects the array and scans each *to-scan* object: it 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 is 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. Thus, 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 not 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 point to P from the L2, as shown in Fig. 10. P is not evicted from the L1 unless it is not recently-used.



(a) Before E's eviction. (b) After E's eviction.

Fig. 10: Example of an eviction that requires changing the canonical level of a non-evicted object (P).

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* [91] 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. 5), 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 overheads. 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. 11 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.

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

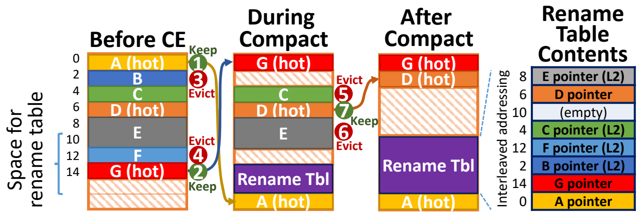


Fig. 11: Dual-ended compaction example. All objects are 2 words long, except E, which takes 4 words.

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.

F. 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.

Fig. 12 shows an example how Hotpads accesses subobjects. **A** is a 224-byte (28-word) object at L2 word address 0xA00. Fig. 12 shows the state of the system after the core issues a load to **A** at a 20-word offset. This fetches a subobject **A.2** with words 16–23 of **A** into the L1. The L1 c-tag entry maps **A.2**'s canonical address 0xA10 to its L1 address, 0xD0.

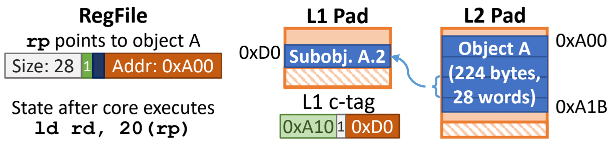


Fig. 12: Example access to an L2-canonical object using subobjects.

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

Each register has an associated shadow subobject register. The shadow subobject register stores a pointer to the subobject that was last accessed using its associated register. If an access 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.

Finally, objects≥128 KB cannot encode their size in the pointer's 14 size bits (Fig. 5). These objects are allocated directly in main memory (Sec.V-D2), their pointers' size bits are set to 0, and only their subobjects are fetched into the pads.

G. Object-level coherence

Hotpads is orthogonal to coherence. We implement MESI coherence with four 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). 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.

Fourth, the private, intermediate levels (the L2 pad in our case) store whether lower levels may have the object, so that they can filter invalidations. For example, when the L1 evicts an object, the L2 marks it as not present in the L1. If refetched, it is marked as present. This acts as a one-entry sharer bit-vector.

A key advantage of this design is that it avoids false sharing as long as contended data is in a separate object. For example, two cores may contend on a small object, and only this contended object is transferred on coherence actions.

Fig. 13 shows such an example on a two-core system with private L1 pads and a shared L2 pad. Initially, core 0's L1 has a copy of B in E state (1). Core 1 then issues a store to B (2). This invalidates B's copy in core 0's L1. Note A's pointer to B's now-invalid L1 copy is not updated. Finally, core 0 traverses A's pointer to B (3), finds B's copy is invalid, reads its canonical address above (Fig. 6), and fetches it to the L1 (using the same space). This fetch downgrades core 1 L1's B copy to S and causes a dirty writeback to the shared pad. Both cores' L1s end up with read-only shared copies of B.

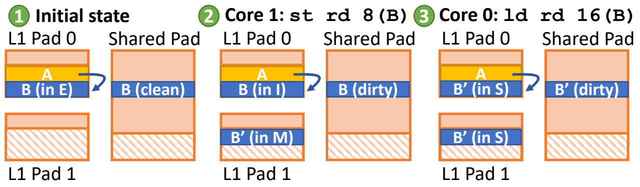


Fig. 13: Example of object-level coherence in Hotpads.

Coherence requires some changes to the invalidation machinery and object format because now canonical objects may need to be invalidated. For example, suppose B in Fig. 13 1 pointed to a newly created, L1 canonical object C. To preserve Invariant 3 (an object may only point to objects in the same or higher levels), when B is invalidated C must be made an L2 canonical object. Hotpads does so following the same procedure as in Fig. 10. For this reason, accesses to canonical objects must do a valid check, and canonical objects also include a canonical pointer above as in Fig. 6. This pointer is used to find the new canonical address if the object is invalidated.

This design handles subobjects in the same way. For large objects, coherence is maintained at subobject granularity. The shared pad uses extra bits (4 bits per subobject) to track subobject sharers of large canonical objects.

H. *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 [78] 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.

I. *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 64 bytes, small objects suffer from overfetching. We add a small cache to the memory controller (8 KB 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. All addressing modes are supported, and hardware treats addresses as offsets to the object. In this mode, Hotpads is somewhat slower than caches due to serial c-tag lookups (there is no pointer rewriting) and bulk evictions.

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 [34] and Verve [104] 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.

TABLE II: Configuration of the simulated 4-core system.

Cores		4 cores, x86-64 ISA, 3.6 GHz, Westmere-like OOO [81]: 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 LQ, 32-entry SQ
Caches	**L1**	64 KB, 8-way set-associative, split D/I caches, 64 B lines
	L2	512 KB private per-core, 8-way set-associative
	L3	4 banks, 2 MB/bank, 16-way set-associative, LRU replacement
Hotpads	**L1D**	64 KB data array, 1K ctag entries (+4KB metadata)
	L1I	64 KB cache, 8-way set-associative, 128 B lines
	L2	512 KB data array, 8K ctag entries (+32KB metadata)
	L3	4×2 MB data array, 4×32K ctag entries (+4×128KB metadata)
Main mem		2 DDR3-1600 channels, 20 nJ per 64B access [59]

VI. Experimental Methodology

We prototype Hotpads using MaxSim [77], a simulation platform that combines ZSim [81], a Pin-based [55] simulator, and Maxine [101], a 64-bit metacircular research JVM.

A. *Hardware*

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

Core modifications: We use out-of-order cores modeled and validated after Westmere [81]. We encode the Hotpads ISA using x86 opcodes that the JVM does not emit.

We believe that a Hotpads ISA designed from the ground up should have separate architectural registers for data and pointers (similar to the index registers of early computers). This separation would enable multiple optimizations, e.g., placing the pointer register file close to the L1 instead of early in the pipeline. However, x86 has general-purpose registers, so for ease of prototyping we allow any register to hold pointers.

Each physical register includes a pointer bit with the same semantics as those in the pads: `ldptr` and `derefptr` set their destination register's pointer bit, and other instructions reset their destination register's pointer bit. An exception is triggered if a non-pointer instruction attempts to use a register that holds a pointer as a source operand (and vice versa).

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 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 accesses with a canonical address before the first of such loads rewrites the pointer. This does not impact correctness.

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.

Cache scrubbing: Finally, we implement cooperative cache scrubbing [82], which adds instructions to zero and scrub (i.e., undirty) cache lines and uses them in the JVM to reduce memory traffic due to object allocation and recycling.

TABLE III: Java workloads and inputs used.

Suite	Benchmark and input
DaCapo MR-9.12	batik, fop, h2, jython, pmd, luindex, lusearch, lusearch-fix, sunflow, xalan (all use default inputs)
SpecJBB	1 warehouse per thread, 50K transactions
JgraphT	pagerank, coloring with amazon-2008 graph

TABLE IV: Latency, energy, and area of caches and pads.

	Type	Latency (cycles)			Energy (pJ)			Area	Leakage
		Direct	Hit	Miss	Direct	Hit	Miss	(mm^2)	(mW)
L1	Cache	–	2	1	–	74	250	0.26	32
	Pad	2	3	1	16	27	194	0.28	32
L2	Cache	–	9	2	–	371	402	1.53	77
	Pad	7	9	2	348	378	30	1.55	81
L3	Cache	–	14	5	–	742	795	21.46	791
	Pad	9	14	5	655	771	828	21.88	793

We model 8-byte L1 accesses and 64-byte objects for pad misses and fills.

B. *Software*

JVM: Our cache-based systems use the Maxine JVM with the C1X JIT compiler. For our Hotpads experiments, we modify the JIT compiler to follow the Hotpads ISA. We fast-forward JVM initialization and warm up the JIT compiler like prior work [8] before starting simulation.

Our cache-based systems use a tuned, stop-the-world generational GC. We set the young heap size to 16 MB, twice the LLC size, which provides the best average performance across all benchmarks [24] (this matches prior work [7, 82]). The old heap is tuned per application: for each workload, we first find the smallest heap size that does not crash, and use 2× that size. This is standard methodology [9, 82].

Hotpads performs concurrent CEs in hardware, and non-generational, stop-the-world GC in software when the main-memory heap fills up. Hotpads uses the same heap sizes as the cache-based systems. Comparing stop-the-world young GCs in software vs. concurrent CEs in Hotpads is fair, because concurrent GCs have higher overheads (Sec. II) and we compare throughput and efficiency, not pause times.

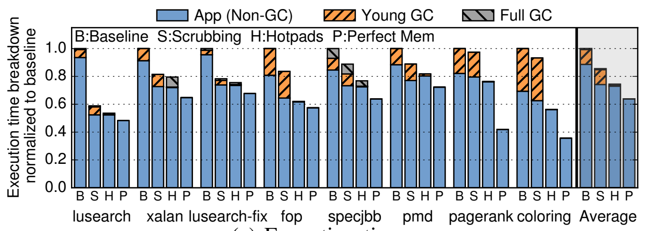
Workloads: We study 13 Java workloads: 10 from the DaCapo [8] suite, `SPECjbb2005` [87], and the `PageRank` and `Coloring` graph processing workloads from `JgraphT` [67], a popular Java graph library. Table III describes their input sets.

VII. Evaluation

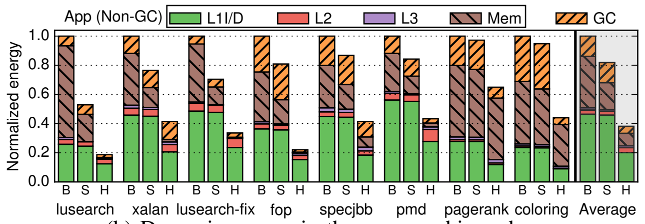
A. *Latency, energy, and area of caches vs. pads*

Table IV reports the latency, dynamic energy, leakage, and area for both caches and pads. We use CACTI 6.5 [64] 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 [46, 79, 106], and cache figures agree with prior work [31, 41, 46].

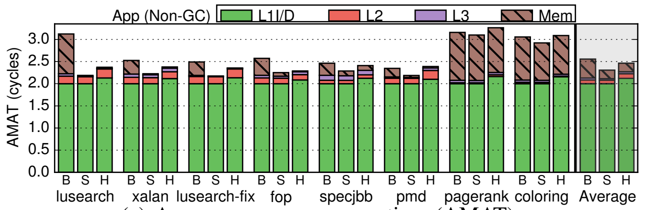
Table IV shows that pads are slightly larger than caches (2% overall area overhead), and have slightly higher leakage, owing to their extra state. However, *direct* accesses to pads, i.e., those that do not check the c-tags, are faster and substantially



(a) Execution time.



(b) Dynamic energy in the memory hierarchy.



(c) Average memory access time (AMAT).

Fig. 14: Simulation results for single-threaded workloads.

more efficient. This difference is largest in the L1: a direct L1 pad access consumes 16 pJ, 4.3× less than an L1 cache hit. As we will see, direct accesses are the common case in the L1. Table IV shows smaller differences for higher levels, but Table IV assumes 64-byte objects. In practice, L2 and L3 pads transfer fewer words than caches, which improves efficiency.

B. *Hotpads outperforms traditional hierarchies*

We first analyze single-threaded workloads. Due to space constraints, we show results for 8 representative apps out of the 13 benchmarks. *All averages include all 13 apps.* We first discuss general trends, then differences across apps.

Performance: Fig. 14a 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 34% on average and by up to 86% (on `lusearch`). These gains stem from both reducing GC overheads, which take 11% of time on average on the baseline vs. 1.5% on Hotpads, and reducing application runtime by 21% due to better memory performance.

By contrast, Scrubbing outperforms the baseline by only 17%. 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 15%.

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 8×. Although young-GC overheads are larger than full-GC overheads, our software schemes use the best-performing young heap size, 16 MB (about the L3 size). 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 57% over the baseline. Hotpads thus bridges 61% of the performance gap between the baseline and a perfect memory system, whereas Scrubbing bridges 30% of the gap.

Memory hierarchy energy: Fig. 14b shows the breakdown of dynamic energy in the memory hierarchy.

Hotpads reduces the memory hierarchy's dynamic energy by 2.6× over the baseline, due to three major factors. First, L1 (instruction and data) dynamic energy is 2.3× smaller because L1 data pads receive mostly direct accesses (Sec. VII-E), making them much more efficient than L1 data caches. Second, Hotpads reduces main memory energy by 4.1×. Third, Hotpads CEs take 2.9× less energy than software GCs.

By contrast, Scrubbing reduces dynamic energy by 22% over the baseline, chiefly by reducing main memory traffic. Hotpads consumes 2.1× less energy than Scrubbing.

Memory hierarchy latency: Fig. 14c shows the breakdown of average memory access time (AMAT) by hierarchy level for *application work, i.e., excluding GC*.

Fig. 14c 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 IV). Thanks to pointer rewriting, most L1 pad accesses do not incur this penalty. As a result, Hotpads's AMAT is 4% lower than the baseline's and is only 7% higher than Scrubbing's AMAT.

Differences across apps: Fig. 14 sorts apps by the mean lifetime of their objects. Hotpads's benefits vary across them.

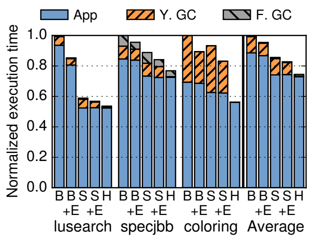
The first three apps (`lusearch`, `xalan`, `lusearch-fix`) allocate many short-lived objects that fit in on-chip pads. Hotpads collects them before they reach main memory, and thus nearly eliminates main memory traffic and enjoys minimal GC energy. By contrast, the baseline and Scrubbing still incur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

The next three apps (`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.

Finally, the last two apps (`pagerank`, `coloring`) have large, long-lived data structures that reside in main memory. Hotpads's benefits for them mostly come from reducing GC overheads, but main memory traffic is only slightly lower than the cache-based schemes.

C. *Cache hierarchy enhancements vs. Hotpads*

Fig. 15 shows results for cache hierarchies enhanced with state-of-the-art techniques: using DRRIP [36] in the LLC


(a) Execution time.

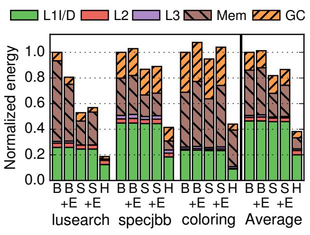

(b) Energy breakdowns.

Fig. 15: Simulation results for baselines w/ DRRIP & prefetchers.

and stream prefetchers modeled after Nehalem's between the L1s and L2 (B+E for enhanced baseline; S+E for enhanced Scrubbing). These features improve performance by 8% over the baseline and 5% over Scrubbing, but Hotpads still outperforms them by 23% and 12%. Moreover, prefetchers *degrade* memory energy over Scrubbing due to mispredicted prefetches, increasing Hotpads's advantage.

We observe that the enhanced cache hierarchies help GCs and bump-pointer object allocation, which have regular, scanning access patterns. However, they barely help application accesses to objects, which are irregular. Because Scrubbing already accelerates allocation, these enhancements do not reduce application runtime over Scrubbing, and improve GCs only.

These results show that Hotpads's features are more effective than conventional cache optimizations. Moreover, Hotpads could adapt similar optimizations e.g., by prefetching related objects or consecutive subobjects (Sec. IX).

D. *Hotpads reduces data movement across the hierarchy*

Fig. 16 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 6.6× in main memory, while scrubbing only saves 66%. 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. 17 shows the number of canonical object bytes that are allocated or evicted into each level (*In* bar), and the number of object bytes that are evicted out of or collected at each level (*Out* bar). Most of the data is allocated *and collected* 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. 16's drastic reduction in write traffic beyond the L1.

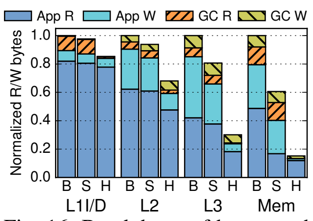


Fig. 16: Breakdown of bytes read and written per level.

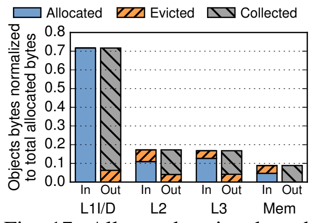


Fig. 17: Allocated, evicted, and collected bytes per pad level.

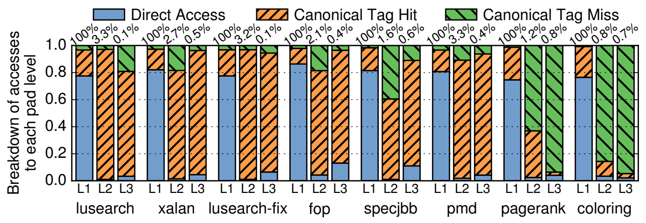


Fig. 18: Breakdown of access types across pad levels.

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 29% and 33% utilization. L2s and L3s show similar differences.

E. *Pointer rewriting avoids most associative lookups*

Fig. 18 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. This explains why L1 pads consume far less energy than L1 caches (Fig. 14b), and why they only incur a small AMAT penalty (Fig. 14c).

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.

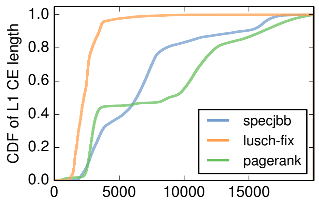
F. *CEs are fast and infrequent*

Table V shows the duration and frequency of CEs on all pads, averaged across all apps. 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 are about 1000× more frequent and cheaper than L3 CEs).

Fig. 19 shows distributions (CDFs) of L1 pad CE length and interval between consecutive CEs for three representative apps. Two main factors determine the length of each CE. First, apps 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

TABLE V: CE DURATION AND FREQUENCY ACROSS PAD LEVELS.

Pad level	L1	L2	L3
Avg. CE length (cycles)	5.1K	251K	5.7M
Avg. interval between CEs (cycles)	128K	3.9M	237M
Fraction of time running CEs	3.99%	6.29%	2.41%


(a) CE length (cycles).

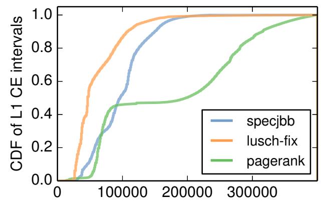
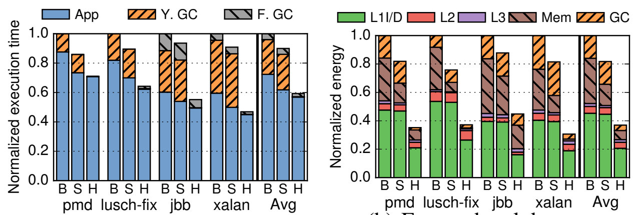

(b) Interval between CEs (cycles).

Fig. 19: CDFs of lengths and intervals for L1 CEs.



(a) Execution time. (b) Energy breakdowns.

Fig. 20: Simulation results for multithreaded workloads.

pad, while the other two apps have longer CEs. Second, the pad gives priority to demand accesses over CE accesses, so CEs take longer on apps 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 apps. 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.

G. *Hotpads performs well on multithreaded workloads*

Fig. 20 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 68%). Hotpads reduces non-GC runtime similarly to single-thread results, and also achieves a similar energy reduction over the baseline (2.7×).

H. *Case study: Hotpads benefits on compiled code*

We have so far focused on Java workloads, where JIT overheads and JITted code quality limit performance compared to compiled applications (e.g., those written in C/C++, Go, or Rust). Nonetheless, compiled programs can also benefit from Hotpads as long as they use the Hotpads ISA.

To demonstrate this, we study `GCBench` [11], a C benchmark for garbage collection. `GCBench` creates and traverses binary trees of different sizes. We compare three variants of `GCBench`:

- *Manual* allocates objects (trees and tree nodes) using malloc, and frees each object manually when it goes out of scope. This is the standard C implementation. We report results under the standard glibc malloc and Google's tcmalloc [27].
- *Automatic* uses garbage collection to avoid freeing objects explicitly. We report results under the Boehm GC, a non-moving GC for C/C++ [10], and Hotpads.
- *Custom* uses a fully customized memory management strategy: it allocates an arena [30] for each tree. Tree nodes are allocated compactly in the arena using very simple bump-pointer allocation. When the tree goes out of scope, the arena is deallocated, freeing all tree nodes in bulk. This is currently the fastest implementation of `GCBench` in the computer language benchmarks game [28].

Fig. 21 shows simulation results, normalized to glibc's malloc. TCMalloc improves over the default in all metrics. TCMalloc's allocation routines are faster, reducing execution time and instructions. TCMalloc also stores small object more compactly, reducing memory energy and footprint.

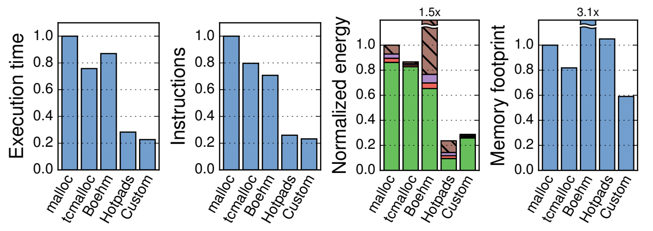


Fig. 21: Simulation results for `GCbench`.

Boehm's GC provides automatic memory management at the cost of memory energy (1.5× worse than the baseline) and footprint (3.1× worse). It is also slightly slower than TCMalloc.

Custom achieves the best performance due to its specialized allocation strategy. It reduces execution time and instructions by 4.3×, memory energy by 3.5×, and footprint by 67%.

Finally, Hotpads retains the simplicity of GC and delivers performance close to Custom. Hotpads reduces execution time by 3.6×. It achieves the lowest memory energy, a 4.2× reduction. And its memory footprint is only 5% worse than the baseline and much lower than Boehm's GC.

This result shows that Hotpads can also substantially benefit compiled applications. Compiled languages are moving away from explicit memory management. For example, modern C++ advocates using smart pointers [89], and Go uses tracing GC. Workloads in these languages could be compiled to use Hotpads automatically. We leave this to future work.

I. *Legacy mode incurs small performance overheads*

Finally, Hotpads's legacy mode incurs modest overheads on programs that use conventional loads and stores. We run unmodified SPEC CPU2006 applications in legacy mode (Sec. V-I). Hotpads is 4% slower than a cache hierarchy on average (up to 14% on `xalancbmk`). This slowdown stems from two factors. First, in this mode Hotpads caches subobjects from a single main-memory object and does not rewrite pointers. This causes serial tag-data lookups that hurt access latency. Second, Hotpads performs bulk evictions. This keeps the data array partly unused, whereas caches stay nearly full with lines.

VIII. Additional Related Work

Though we have focused on GC-based languages, prior work has proposed software [1, 5, 66] and hardware [13, 65, 97] 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 [39, 40], which Hotpads would avoid.

Adaptive-granularity designs like sector caches [53, 86] and Amoeba [47] improve utilization and reduce traffic, but they have more involved tag lookups and require predictors to fetch data at the right granularity [37, 47]. Hotpads avoids these overheads while fetching data at fine granularity.

The V-Way cache [74] decouples tag and data arrays similarly to how c-tag and data arrays are decoupled in pads. V-Way improves associativity by oversizing the tag array, whereas pads need this organization to manage the data array independently.

GPUs and many accelerators [14, 15, 26, 29, 98] 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 [50].

Stash [46] 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.

Virtual memory and caches conventionally use two separate associative lookups, on TLBs and cache tags. TLC [85], D2D [83], D2M [84], and cTLB [49] fold cache tag information into the TLB to reduce or eliminate cache tag lookups. However, 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.

IX. 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 the speculation-related cache side channels [70] that underpin the recent Spectre [45] and Meltdown [52] attacks. A secure Hotpads implementation might need to close other side channels, e.g., randomizing CEs.
- **Isolation:** Hotpads may reduce or eliminate VM overheads (Sec. V-I), 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 [73, 80, 99].
- **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 (Sec. V-D)? Could we perform locality-aware level selection and bypass [42, 94] 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 [25, 63] or accelerate NVM logging [35, 69, 105].

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.

Acknowledgments

We sincerely thank Maleen Abeydeera, Joel Emer, Mark Jeffrey, Christos Kozyrakis, Hyun Ryong Lee, Xiaosong Ma, Anurag Mukkara, Suvinay Subramanian, Victor Ying, Guowei Zhang, and the anonymous reviewers for their feedback. This work was supported in part by NSF grant CAREER-1452994 and by a grant from the Qatar Computing Research Institute.

References

[1] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors." in *Proc. PLDI*, 2009.
[2] O. Anderson, E. Fortuna, L. Ceze, and S. Eggers, "Checked load: Architectural support for javascript type-checking on mobile processors," in *Proc. HPCA-17*, 2011.
[3] T. A. Anderson, M. ONeill, and J. Sarracino, "Chihuahua: A concurrent, moving, garbage collector using transactional memory," *TRANSACT*, 2015.
[4] A. W. Appel, J. R. Ellis, and K. Li, "Real-time concurrent collection on stock multiprocessors," in *Proc. PLDI*, 1988.
[5] T. M. Austin, S. E. Breach, and G. S. Sohi, "Efficient detection of all pointer and array access errors," in *Proc. PLDI*, 1994.
[6] D. F. Bacon, P. Cheng, and V. Rajan, "A real-time garbage collector with low overhead and consistent utilization," in *Proc. PLDI*, 2003.
[7] S. M. Blackburn, P. Cheng, and K. S. McKinley, "Myths and realities: The performance impact of garbage collection," in *SIGMETRICS*, 2004.
[8] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg *et al.*, "The DaCapo benchmarks: Java benchmarking development and analysis," in *Proc. OOPSLA*, 2006.
[9] S. M. Blackburn and K. S. McKinley, "Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance," in *Proc. PLDI*, 2008.
[10] H.-J. Boehm, "A garbage collector for C and C++," http://www.hboehm.info/gc/, archived at https://perma.cc/L5WY-Y28N, 2002.
[11] H.-J. Boehm, "An artificial garbage collection benchmark," http://hboehm.info/gc/gc_bench.html, archived at https://perma.cc/Y4BY-7RN4, 2002.
[12] H.-J. Boehm, A. J. Demers, and S. Shenker, "Mostly parallel garbage collection," in *Proc. PLDI*, 1991.
[13] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase *et al.*, "Flexible hardware acceleration for instruction-grain program monitoring," in *Proc. ISCA-35*, 2008.
[14] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proc. ASPLOS-XIX*, 2014.
[15] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Proc. ISCA-43*, 2016.
[16] J. Choi, T. Shull, M. J. Garzaran, and J. Torrellas, "Shortcut: Architectural support for fast object access in scripting languages," in *Proc. ISCA-44*, 2017.
[17] A. Clements and R. Hudson, "Go proposal: Eliminate STW stack re-scanning," http://github.com/golang/proposal/blob/master/design/17503-eliminate-rescan.md, archived at https://perma.cc/7FHN-MDDH, 2016.
[18] G. E. Collins, "A method for overlapping and erasure of lists," *Communications of the ACM*, vol. 3, no. 12, 1960.
[19] A. R. Cunha, C. N. Ribeiro, and J. A. Marques, "The architecture of a memory management unit for object-oriented systems," in *Proc. ISCA-18*, 1991.
[20] W. J. Dally and J. T. Kajiya, "An object oriented architecture," in *Proc. ISCA-12*, 1985.
[21] J. Devietti, C. Blundell, M. M. Martin, and S. Zdancewic, "HardBound: Architectural support for spatial safety of the C programming language," in *Proc. ASPLOS-XIII*, 2008.
[22] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. Steffens, "On-the-fly garbage collection: An exercise in cooperation," *Communications of the ACM*, vol. 21, no. 11, 1978.
[23] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum, "Improving cache management policies using dynamic reuse distances," in *Proc. MICRO-45*, 2012.
[24] Y. L. Gan, "Redesigning the memory hierarchy for memory-safe programming languages," Master's thesis, MIT, 2018.
[25] M. J. Garzarán, M. Prvulovic, J. M. Llabería, V. Viñals, L. Rauchwerger, and J. Torrellas, "Tradeoffs in buffering memory state for thread-level speculation in multiprocessors," in *Proc. HPCA-9*, 2003.
[26] M. Gebhart, S. W. Keckler, B. Khailany, R. Krashinsky, and W. J. Dally, "Unifying primary cache, scratch, and register file memories in a throughput processor," in *Proc. MICRO-45*, 2012.
[27] S. Ghemawat and P. Menage, "TCMalloc: Thread-Caching Malloc http://goog-perftools.sourceforge.net/doc/tcmalloc.html," 2007.
[28] I. Gouy, "The computer language benchmarks game," https://benchmarksgame-team.pages.debian.net/benchmarksgame, archived at https://perma.cc/7AW2-6NPN, 2002.
[29] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *Proc. MICRO-49*, 2016.
[30] D. R. Hanson, "Fast allocation and deallocation of memory based on object lifetimes," *Software: Practice and Experience*, vol. 20, 1990.
[31] M. Horowitz, "Computing's energy problem (and what we can do about it)," in *Proc. ISSCC*, 2014.
[32] M. E. Houdek, F. G. Soltis, and R. L. Hoffman, "Ibm system/38 support for capability-based addressing," in *Proc. ISCA-8*, 1981.
[33] R. Hudson and A. Clements, "Request oriented collector (ROC) algorithm," http://golang.org/s/gctoc, archived at https://perma.cc/S2SV-SVHX, 2016.

[34] G. C. Hunt and J. R. Larus, "Singularity: Rethinking the software stack," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 2, 2007.
[35] J. Izraelevitz, T. Kelly, and A. Kolli, "Failure-atomic persistent memory updates via JUSTDO logging," in *Proc. ASPLOS-XXI*, 2016.
[36] A. Jaleel, K. Theobald, S. C. Steely, and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in *Proc. ISCA-37*, 2010.
[37] D. Jevdjic, S. Volos, and B. Falsafi, "Die-stacked DRAM caches for servers: Hit ratio, latency, or bandwidth? Have it all with footprint cache," in *Proc. ISCA-40*, 2013.
[38] J. A. Joao, O. Mutlu, and Y. N. Patt, "Flexible reference-counting-based hardware acceleration for garbage collection," in *Proc. ISCA-36*, 2009.
[39] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *Proc. ISCA-42*, 2015.
[40] S. Kanev, S. L. Xi, G.-Y. Wei, and D. Brooks, "Mallacc: Accelerating Memory Allocation," in *Proc. ASPLOS-XXII*, 2017.
[41] S. Keckler, W. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the future of parallel computing," *IEEE Micro*, vol. 31, no. 5, 2011.
[42] S. M. Khan, Y. Tian, and D. A. Jimenez, "Sampling dead block prediction for last-level caches," in *Proc. MICRO-43*, 2010.
[43] C. Kim, J. Kim, S. Kim, D. Kim, N. Kim, G. Na, Y. H. Oh, H. G. Cho *et al.*, "Typed architectures: Architectural support for lightweight scripting," in *Proc. ASPLOS-XXII*, 2017.
[44] C. Kim, S. Kim, H. G. Cho, D. Kim, J. Kim, Y. H. Oh, H. Jang, and J. W. Lee, "Short-circuit dispatch: Accelerating virtual machine interpreters on embedded processors," in *Proc. ISCA-43*, 2016.
[45] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," *arXiv preprint arXiv:1801.01203*, 2018.
[46] R. Komuravelli, M. D. Sinclair, J. Alsop, M. Huzaifa, M. Kotsifakou, P. Srivastava, S. V. Adve, and V. S. Adve, "Stash: Have your scratchpad and cache it too," in *Proc. ISCA-42*, 2015.
[47] S. Kumar, H. Zhao, A. Shriraman, E. Matthews, S. Dwarkadas, and L. Shannon, "Amoeba-cache: Adaptive blocks for eliminating waste in the memory hierarchy," in *Proc. MICRO-45*, 2012.
[48] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight Jr, and A. DeHon, "Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security," in *Proc. CCS*, 2013.
[49] Y. Lee, J. Kim, H. Jang, H. Yang, J. Kim, J. Jeong, and J. W. Lee, "A fully associative, tagless DRAM cache," in *Proc. ISCA-42*, 2015.
[50] J. Leverich, H. Arakida, A. Solomatnikov, A. Firoozshahian, M. Horowitz, and C. Kozyrakis, "Comparing memory systems for chip multiprocessors," in *Proc. ISCA-34*, 2007.
[51] P. Liden and S. Karlsson, "The Z garbage collector: An introduction," in *FOSDEM*, 2018.
[52] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin *et al.*, "Meltdown," *arXiv preprint arXiv:1801.01207*, 2018.
[53] J. S. Liptay, "Structural aspects of the System/360 Model 85, II: The cache," *IBM Systems Journal*, vol. 7, no. 1, 1968.
[54] B. Liskov and S. Zilles, "Programming with abstract data types," in *ACM Sigplan Notices*, vol. 9, no. 4. ACM, 1974.
[55] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi *et al.*, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proc. PLDI*, 2005.
[56] F. Mao, E. Z. Zhang, and X. Shen, "Influence of program inputs on the selection of garbage collectors," in *VEE*, 2009.
[57] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, Part I," *Comm. ACM*, vol. 3, no. 4, 1960.
[58] P. McGachey, A.-R. Adl-Tabatabai, R. L. Hudson, V. Menon, B. Saha, and T. Shpeisman, "Concurrent GC leveraging transactional memory," in *Proc. PPoPP*, 2008.
[59] Micron, "1.35V DDR3L power calculator (4Gb x16 chips)," 2013.
[60] Microsoft, "Memory management and garbage collection in .NET," http://docs.microsoft.com/en-us/dotnet/standard/garbage-collection, archived at https://perma.cc/XG3P-5CDF, 2017.
[61] D. A. Moon, "Garbage collection in a large lisp system," in *Proc. LISP and functional programming*. ACM, 1984.
[62] D. A. Moon, "Architecture of the symbolics 3600," in *Proc. ISCA-12*, 1985.
[63] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, "LogTM: Log-based transactional memory." in *Proc. HPCA-12*, 2006.
[64] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," HP laboratories, Tech. Rep., 2009.
[65] S. Nagarakatte, M. M. Martin, and S. Zdancewic, "Watchdog: Hardware for safe and secure manual memory management and full memory safety," in *Proc. ISCA-39*, 2012.
[66] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "SoftBound: Highly compatible and complete spatial memory safety for C," in *Proc. PLDI*, 2009.
[67] B. Naveh, "JGraphT," http://jgrapht.org, 2018.
[68] T. Nojiri, S. Kawasaki, and K. Sakoda, "Microprogrammable processor for object-oriented architecture," in *Proc. ISCA-13*, 1986.
[69] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *Proc. ISCA-41*, 2014.
[70] C. Percival, "Cache missing for fun and profit," *BSDCan*, 2005.
[71] F. Pizlo, L. Ziarek, P. Maj, A. L. Hosking, E. Blanton, and J. Vitek, "Schism: Fragmentation-tolerant real-time garbage collection," in *Proc. PLDI*, 2010.
[72] Python Software Foundation, "Garbage collector interface, the Python standard library," https://docs.python.org/3/library/gc.html, 2018.
[73] M. Qureshi and Y. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proc. MICRO-39*, 2006.
[74] M. Qureshi, D. Thompson, and Y. Patt, "The V-way cache: Demand based associativity via global replacement," in *Proc. ISCA-32*, 2005.
[75] Redox Developers, "Redox OS," https://www.redox-os.org/, 2018.
[76] C. G. Ritson, T. Ugawa, and R. E. Jones, "Exploring garbage collection with Haswell Hardware Transactional Memory," *ACM SIGPLAN Notices*, vol. 49, no. 11, 2015.
[77] A. Rodchenko, C. Kotselidis, A. Nisbet, A. Pop, and M. Luján, "Maxsim: A simulation platform for managed applications," in *Proc. ISPASS*, 2017.
[78] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace cache: A low latency approach to high bandwidth instruction fetching," in *Proc. MICRO-29*, 1996.
[79] D. Sanchez and C. Kozyrakis, "The ZCache: Decoupling ways and associativity," in *Proc. MICRO-43*, 2010.
[80] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and efficient fine-grain cache partitioning," in *Proc. ISCA-38*, 2011.
[81] D. Sanchez and C. Kozyrakis, "ZSim: Fast and accurate microarchitectural simulation of thousand-core systems," in *Proc. ISCA-40*, 2013.
[82] J. B. Sartor, W. Heirman, S. M. Blackburn, L. Eeckhout, and K. S. McKinley, "Cooperative cache scrubbing," in *Proc. PACT-23*, 2014.
[83] A. Sembrant, E. Hagersten, and D. Black-Schaffer, "The direct-to-data (D2D) cache: Navigating the cache hierarchy with a single lookup," in *Proc. ISCA-41*, 2014.
[84] A. Sembrant, E. Hagersten, and D. Black-Schaffer, "A split cache hierarchy for enabling data-oriented optimizations," in *Proc. HPCA-23*, 2017.
[85] A. Sembrant, E. Hagersten, and D. Black-Shaffer, "TLC: A tag-less cache for reducing dynamic first level cache energy," in *Proc. MICRO-46*, 2013.
[86] A. Seznec, "Decoupled sectored caches: Conciliating low tag implementation cost," in *Proc. ISCA-21*, 1994.
[87] Standard Performance Evaluation Corporation, "SPECjbb2005 (Java server benchmark)," https://www.spec.org/jbb2005/, 2006.
[88] P. Steenkiste and J. Hennessy, "Tags and type checking in lisp: Hardware and software approaches," in *Proc. ASPLOS-II*, 1987.
[89] B. Stroustrup and H. Sutter, "Core C++ guidelines," https://isocpp.github.io/CppCoreGuidelines/, archived at https://perma.cc/ZRW6-TS8N, 2018.
[90] Sun Microsystems, "Memory management in the Java HotSpot virtual machine," http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf, archived at https://perma.cc/Z97A-27AB, 2006.
[91] A. S. Tanenbaum and D. J. Wetherall, *Computer networks*, 5th ed., P. Hall, Ed., 2010.
[92] G. Tene, B. Iyengar, and M. Wolf, "C4: The continuously concurrent compacting collector," in *Proc. ISMM*, 2011.
[93] E. Teran, Z. Wang, and D. A. Jiménez, "Perceptron learning for reuse prediction," in *Proc. MICRO-49*, 2016.
[94] P.-A. Tsai, N. Beckmann, and D. Sanchez, "Jenga: Software-defined cache hierarchies," in *Proc. ISCA-44*, 2017.
[95] D. Ungar, "Generation scavenging: A non-disruptive high performance storage reclamation algorithm," in *ACM Sigplan notices*, vol. 19, no. 5. ACM, 1984.
[96] D. Ungar, R. Blau, P. Foley, D. Samples, and D. Patterson, "Architecture of SOAR: Smalltalk on a RISC," in *Proc. ISCA-11*, 1984.
[97] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic, "Memtracker: Efficient and programmable support for memory access monitoring and debugging," in *Proc. HPCA-13*, 2007.
[98] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank *et al.*, "Baring it all to software: Raw machines," *IEEE Computer*, vol. 30, no. 9, 1997.
[99] X. Wang, S. Chen, J. Setter, and J. F. Martínez, "SWAP: Effective fine-grain management of shared last-level caches with minimum hardware support," in *Proc. HPCA-23*, 2017.
[100] P. R. Wilson, "Uniprocessor garbage collection techniques," in *Memory Management*. Springer, 1992.
[101] C. Wimmer, M. Haupt, M. L. Van De Vanter, M. Jordan, L. Daynès, and D. Simon, "Maxine: An approachable virtual machine for, and in, java," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, 2013.
[102] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann *et al.*, "The CHERI capability model: Revisiting RISC in an age of risk," in *Proc. ISCA-41*, 2014.
[103] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely Jr, and J. Emer, "SHiP: Signature-based hit predictor for high performance caching," in *Proc. MICRO-44*, 2011.
[104] J. Yang and C. Hawblitzel, "Safe to the last instruction: Automated verification of a type-safe operating system," in *Proc. PLDI*, 2010.
[105] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: Closing the performance gap between systems with and without persistence support," in *Proc. MICRO-46*, 2013.
[106] T. Zheng, H. Zhu, and M. Erez, "SIPT: Speculatively indexed, physically tagged caches," in *Proc. HPCA-24*, 2018.