Hardware-Software Co-optimization of Memory Management in Dynamic Languages

Mohamed Ismail and G. Edward Suh
Cornell University
Ithaca, NY, USA
{mii5,gs272}@cornell.edu

Abstract

Dynamic programming languages are becoming increasingly popular, yet often show a significant performance slowdown compared to static languages. In this paper, we study the performance overhead of automatic memory management in dynamic languages. We propose to improve the performance and memory bandwidth usage of dynamic languages by co-optimizing garbage collection overhead and cache performance for newly-initialized and dead objects. Our study shows that less frequent garbage collection results in a large number of cache misses for initial stores to new objects. We solve this problem by directly placing uninitialized objects into on-chip caches without off-chip memory accesses. We further optimize the garbage collection by reducing unnecessary cache pollution and write-backs through partial tracing that invalidates dead objects between full garbage collections. Experimental results on PyPy and V8 show that less frequent garbage collection along with our optimizations can significantly improve the performance of dynamic languages.

CCS Concepts • **Software and its engineering → Garbage collection**; Scripting languages; • **Computer systems organization** → *Processors and memory architectures*;

Keywords memory management, caches, Python, Javascript

ACM Reference Format:
Mohamed Ismail and G. Edward Suh. 2018. Hardware-Software Co-optimization of Memory Management in Dynamic Languages . In *Proceedings of 2018 ACM SIGPLAN International Symposium on Memory Management (ISMM'18).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3210563.3210566

ISMM'18, June 18, 2018, Philadelphia, PA, USA

ACM ISBN 978-1-4503-5801-9/18/06…$15.00
https://doi.org/10.1145/3210563.3210566

1 Introduction

As software becomes more complex and the costs of developing and maintaining code increase, dynamic programming languages are becoming more desirable alternatives to traditional static languages. Dynamic languages allow programmers to express functionality with less code. In addition, run-time checks and memory management are built-in, limiting the possibility of low-level program bugs such as buffer overflow. Dynamic languages such as Javascript, Python, PHP, and Ruby consistently rank in the top ten most popular languages across multiple metrics [3, 7, 15]. These dynamic languages are increasingly utilized in production environments as well in order to bring new features quickly. For example, Twitter initially used Ruby on Rails to build their infrastructure and Dropbox used Python.

Automatic memory management is a key feature of dynamic languages and other static-but-managed languages (e.g. Java) that contributes significantly to performance overhead. It allows the programmer to easily write code without having to worry about allocation and de-allocation of dynamically created variables. In dynamic languages, particularly, all variables are allocated dynamically which results in frequent allocation and deallocation of small objects. To optimize the cost of frequent allocations, simple sequential allocators are often used. To amortize the cost of the de-allocation, garbage collection needs to run infrequently.

However, there is a fundamental trade-off between garbage collection overhead and cache performance of dynamically allocated objects in today's dynamic languages. On one hand, frequent garbage collection operations lead to significant performance overhead. On the other hand, less frequent garbage collection requires more memory space to keep dynamically-allocated objects over a longer garbage collection period. Such memory allocation using a large memory region increases the working set size and can significantly degrade the cache performance through loading of newly allocated objects from memory and increased cache pollution and write-backs. The impact on cache performance is particularly significant if the memory space for frequent allocations does not fit into on-chip caches. Today's generational garbage collectors generally allocate half of the last level cache size for young objects in order to balance cache pressure and garbage collection overhead.

In this paper, we propose hardware support and software optimizations for reducing modern memory management overhead in dynamic languages. First, we propose to optimize cache performance for newly-allocated objects by directly placing them in on-chip caches without reading the corresponding locations from off-chip memory. Because newly-allocated memory locations need to be initialized anyways, there is no need to read their previous values from memory. Next, we must deal with cache pollution and additional write-backs related to newly-allocated objects. A partial tracing strategy is proposed to determine dead cache lines that do not need to be kept or written back.

These optimizations remove the main obstacles in using a large memory region for new memory allocations and enable running garbage collection far less frequently than what is considered to be optimal today. In this way, our approach simultaneously reduces overhead for both garbage collection and frequent memory allocations. Our experimental results show that this co-optimization of garbage collection and memory allocation can achieve significant performance improvements; 22% performance on average and up to 68% performance for PyPy [1], a popular implementation for Python, and 17% performance on average and up to 63% performance for V8 [4] running Javascript.

The high-level idea of directly placing newly-allocated memory locations into on-chip caches without off-chip accesses is known as *cache installation* and has been studied previously in the context of C and C++ [5, 9, 23]. However, the cache installation itself only leads to small performance improvements for C and C++ because they only optimize relatively infrequent memory allocations. This paper shows that optimizing initialization of newly-allocated locations are far more important for dynamic languages with frequent memory allocations, and more importantly can be used to enable less frequent garbage collection to significantly reduce overhead of managed memory. This co-optimization of garbage collection and memory allocation is essential in obtaining the performance improvements reported in this paper.

Moreover, we found that previous cache installation mechanisms for C and C++ are not well-suited for dynamic languages. The previous mechanisms are designed for memory allocations for objects larger than a cache line (64 bytes). Yet, dynamic languages often allocate objects smaller than a cache line. In addition, previous designs are not built to simultaneously reduce unnecessary cache pollution and write-backs. In this paper, we present a new invalid memory tracking mechanism that is designed to enable cache installation even for small objects by leveraging the sequential memory allocators widely used in today's generational garbage collectors. The same tracking mechanism can simultaneously reduce unnecessary cache pollution and write-backs with software assistance.

The major contributions of this paper include:

1. This paper provides a detailed study of the automatic memory management in dynamic languages, and identifies the trade-off between garbage collection overhead and cache performance of dynamically-allocated objects.
2. This paper introduces a new invalid memory region tracking mechanism that allows cache installation even for small objects commonly used in dynamic languages as well as write-back reduction and pollution control.
3. This paper presents a partial tracing algorithm that can be run to identify invalid cache lines with lower overhead compared to full garbage collection overhead.
4. This paper demonstrates that the proposed optimizations can lead to significant performance improvements in the state-of-the-art implementations of two widely used dynamic languages, Python and Javascript, using a wide range of applications.

This rest of the paper is organized as follows. Section 2 discusses related work. Section 3 describes the memory management features in modern dynamic languages and studies the trade-off between garbage collection overhead and cache performance. Section 4 describes our invalid memory tracking mechanism and Section 5 describes our partial tracing algorithm. Section 6 evaluates the proposed memory management optimizations, and Section 7 concludes the paper.

2 Related Work

The idea of directly installing a cache line without going to memory has been studied previously, but existing proposals are not suitable for dynamic languages. PowerPC has an dcbz instruction that that can install a cache line directly [6]. However, to use this instruction, software needs to be aware of the cache line size of the underlying architecture and explicitly install cache lines one at a time. This limits the portability and applicability.

Other studies have built on the idea to reduce cache misses from stores to newly-allocated memory regions in the context of C and C++. Lewis et al. propose a hardware table to explicitly track mallocs and install newly-allocated cache lines [9]. This works well for C and C++ because dynamically allocated objects are often larger than one cache line. In contrast, dynamic languages frequently allocate small objects, which are smaller than a cache line. The malloc table cannot be used to install a cache line since no assumption can be made on neighboring words in the same cache line. Our tracking table installs cache lines for small objects by assuming later words are unallocated and can additionally be used to reduce cache pollution and eliminate unnecessary write-backs.

Sartor et al. [25] describe using cache installation and scrubbing instructions in the context of reducing DRAM traffic and energy. They use cache installation instructions to

eliminate useless read traffic for nursery allocation and scrubbing instructions to invalidate or deprioritize dead cache lines to reduce dead write traffic. Their solution relies on ISA instructions similar to the PowerPC dcbz instruction and requires software to be aware of the cache line size. They overcome the limitations of using cache install instructions by installing 32kB regions at a time. This can result in unnecessary cache pollution. We consider nursery sizes that are many times larger than the size of the last level cache. Scrubbing such nursery ranges after each nursery collection would be ineffective. Instead, we use partial tracing to identify invalid cache lines at more frequent intervals than nursery collection, and we use hardware to efficiently install and "scrub" cache lines on-demand.

Hu and John [5] and Rui et al. [23] proposed store fill buffer designs where they direct store misses to a buffer and only retrieve the cache line from memory if either the cache line is evicted from the buffer before being fully written to or a word is read before it is written. If a full cache line is written in the buffer, the cache line is directly installed into the cache. These designs can work without any information about memory allocation. The store fill buffer can reduce unnecessary memory reads when the initialization of a full cache line happens within a short period. However, the buffer is shared among all stores and an entry may be evicted before its fully initialized when objects are small. In our design, we use software to precisely tell the hardware the areas of memory that are newly allocated. Our tracker can also be used for write-back reduction and pollution control, while the store fill buffer can be used only for cache installation.

Yang et al. [33] were the first to identify and present a detailed study on the performance impact of zeroing in modern managed languages on recent Intel processors. They show that existing options of zeroing, whether zeroing in bulk or during object allocation, have different trade-offs but similar performance impacts. By using existing cache-bypassing store instructions in the x86 architecture to perform bulk zeroing, they are able to improve the overall performance of the program. We similarly find that object initialization can have high impact on performance if the nursery does not fit in the cache. Our solution is to use cache installation to load nursery cache lines on-demand as objects are initialized. Their bulk zeroing technique is complementary and can be used during garbage collection to further reduce the overhead of garbage collection.

Zhang et al. [34] identified a strong correlation between object allocation rate and memory bus write traffic in partially scalable programs written for Java. They conclude that scalability and performance are limited by object allocation rate on multi-processor platforms resulting in an "allocation wall." In our work, we find that performance of dynamic languages is similarly limited by frequent object allocation when the nursery is larger than the last level cache size. By using cache installation, we can remove this allocation wall and can use larger nursery sizes to achieve better performance. While we focus on dynamic languages, we believe that our technique is applicable to other garbage collected languages. The prior work suggests that the cache optimizations are also important for static-but-managed languages such as Java.

Our tracking hardware for invalid memory regions is similar to the Range Cache [29], which is designed to store security tags for a range of addresses. For our application though, we only need to indicate whether a range of addresses is invalid, which allows us to greatly simplify the hardware. In addition, we do not keep overlapping ranges, which allows us to ensure fast (single-cycle) hits for lookups.

Many previous studies optimized garbage collection for caches. Reddy et al. [20] proposes to pin the nursery, which contains the most recently allocated objects, in the last level cache. This reduces the average memory access latency for new object allocations. However, this limits the nursery size to be less than the cache size which may result in frequent garbage collection. In addition, program performance may suffer from poor cache performance as the workable cache space for non-nursery memory is effectively reduced. Other studies [17, 27, 28] try to eliminate unnecessary write-back operations by using garbage bits to track garbage data in the cache. Garbage cache lines are not written back and can be replaced before other valid cache lines. We propose a novel partial tracing technique that decouples tracing a small part of a nursery from full garbage collection.

Wilson et al. [31] explored how different cache designs affect the performance of generational garbage collection. They concluded that careful attention to memory hierarchy issues can significantly decrease the performance impact of garbage collection. They point out that as caches get larger, the best way to achieve high performance is to make the young generation (nursery) fit within the last level cache. In our work, we show that we can beat the performance of small nurseries that fit in the cache by co-optimizing both hardware and software.

Previous studies have also proposed hardware support for garbage collection. Some of them aim to accelerate the computational overhead of reference counting [8, 32]. Others use a hardware co-processor to achieve more predictable garbage collection in a real-time setting [10, 11, 14, 26].

3 Memory Management in Dynamic Languages

3.1 Generational Garbage Collection

For a language with automatic memory management, garbage collection is used to free memory from objects that are no longer in use. The process of determining which objects are live and which are not incurs non-trivial performance overhead. In order to amortize this overhead, garbage collection should be run at infrequent intervals.

Full garbage collection is expensive, especially when all objects and the full memory space needs to be scanned. In order to limit the inefficiency, the memory space can be separated into multiple subspaces based on the age of objects. In the simplest implementation of generational garbage collection, there is one subspace for young objects, sometimes called a *nursery*, and another subspace for old objects. Objects are allocated in the nursery and are moved to the old subspace if they survive long enough.

Efficient generational garbage collection relies on the assumption that most objects in a program die young. Therefore, a copying garbage collector can efficiently move a small number of surviving objects from the nursery to the old subspace. Once the object is in the old subspace, a slower mark-sweep garbage collector can run less frequently. This can be extended to any number of subspaces based on age.

Generational garbage collection is used in many high performance implementations of modern languages as shown in Table 1. Real implementations of generational garbage collection add variations to this general scheme. For example, the PyPy collector runs the mark-sweep collector incrementally in the old subspace [18]. V8 adds an additional semi-space in the nursery. During garbage collection, young objects are copied from one semi-space to the other and only move to the old generation if they have already been copied once [16].

3.2 Sequential Allocation

In order to make allocation fast in generational garbage collection, a sequential allocator is typically used. The young subspace (i.e. nursery) is always guaranteed to be empty following a garbage collection. The sequential allocator simply uses a pointer to maintain the invariant that anything before is allocated and anything after is unallocated. On allocation, the pointer is incremented to maintain the invariant. A check is also performed to ensure the allocation does not exceed the subspace region. If it does, garbage collection will be run to empty the subspace and reset the pointer to the beginning of the subspace.

When a live object is moved from the young subspace to the old subspace, the old subspace can use a more complicated allocator as the move to the old space happens infrequently. In some cases, a variation of a sequential allocator may still be used in the old subspace.

3.3 Trade-off between Garbage Collection and Cache Performance

While necessary, garbage collection can lead to significant performance overhead. As discussed in Appendix A, garbage collection can be run less frequently by increasing the nursery size. This spreads some of the overhead over more of the program execution and reduces the number of live objects that will be traced and moved.

Table 1. Summary of popular language implementations that use generational garbage collection.

Language	Implementation	Garbage Collector Description
Javascript	V8[16]	Two generation collector with two young semi-spaces and and an old space. Young objects are copied from one young semi-space to the other and then to the old space if they survive.
Python	PyPy[18]	Two generation collector with a nursery and an old region. Young objects surviving in the nursery are copied and moved to the old region, where incremental mark-sweep garbage collection is used.
Ruby	Rubinius[22]	Concurrent generational collection.
Java	Hotspot[12]	Three generation collector with a young generation with three subspaces, an old space, and a permanent space. An object starts in the eden subspace of the young generation and is copied to one of two survivor spaces. If it survives, it is copied to the old space.
C#	.NET CLR[30]	Three generation collector with a young generation for short-lived objects, a buffer generation for semi-short-lived objects, and an long-lived generation.

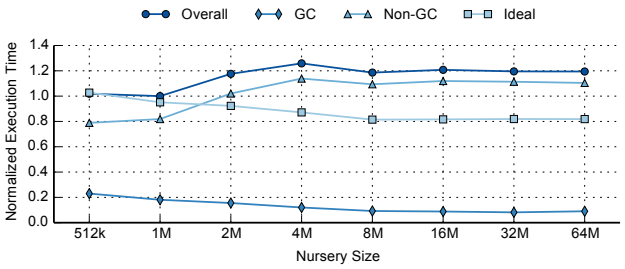


Figure 1. Execution time breakdown as a function of nursery size.

Unfortunately, simply increasing nursery size negatively affects cache performance and can significantly degrade overall performance. For a large nursery, initial accesses to newly-allocated objects are likely to require a load from off-chip memory. Nursery accesses will also have a larger memory footprint and evict more cache lines.

Figure 1 illustrates the impact of the nursery size on the average execution time of 64 benchmarks running on PyPy, a popular Python implementation, normalized to the standard 1MB nursery size used by PyPy. The results are broken down to show execution time of garbage collection vs. the remainder of the execution. As the nursery size increases, garbage collection overhead decreases, yet the overall execution time often increases.

Figure 2, which shows the last-level cache (LLC) miss-rate breakdown as a function of the nursery size, illustrates why

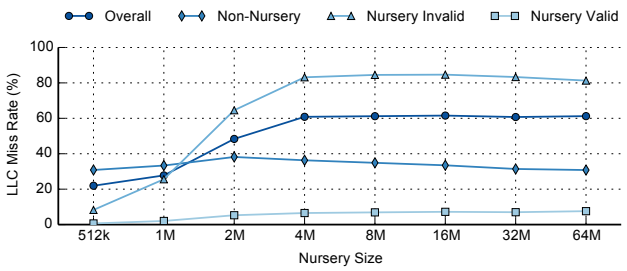


Figure 2. LLC miss rate as a function of nursery size.

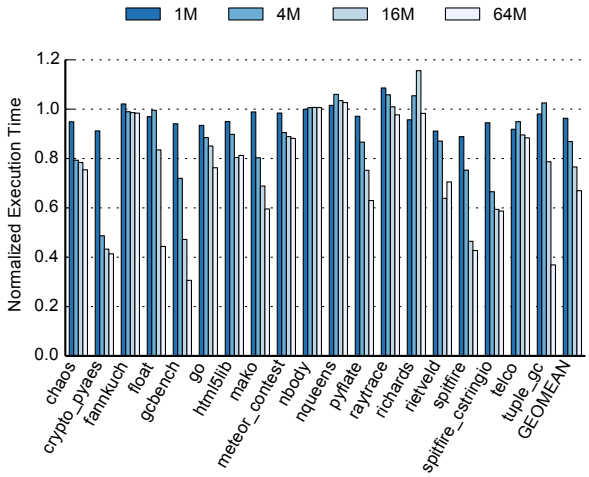


Figure 3. Execution time as a function of nursery size when new objects are directly installed in caches.

the overall execution time increases. As the nursery size increases, the overall LLC miss-rate increases significantly, mainly because initial accesses to newly-allocated nursery locations miss in the cache (shown as Nursery Invalid). The results suggest that it is important to reduce cache misses for initial nursery accesses in order to enable using a larger nursery with low garbage collection overhead.

A larger nursery also puts more pressure on caches and increases cache misses for non-nursery accesses (Non-Nursery) or nursery accesses after the initialization (Nursery Valid). While not shown in the figure, the number of write-backs can also increase significantly for a large nursery.

3.4 Cache Installation of Invalid Memory

The initial accesses to newly-allocated objects represent a series of stores to initialize the objects. These accesses retrieve a cache line from memory and simply overwrite it with a new value. Therefore, there is no need to read these invalid (uninitialized or unallocated) memory locations. Instead, a store miss to an invalid memory region can be serviced by directly placing an arbitrary value (such as zero) into the cache block without reading memory if all memory locations mapped to the cache block are invalid. This technique is often called *cache installation*.

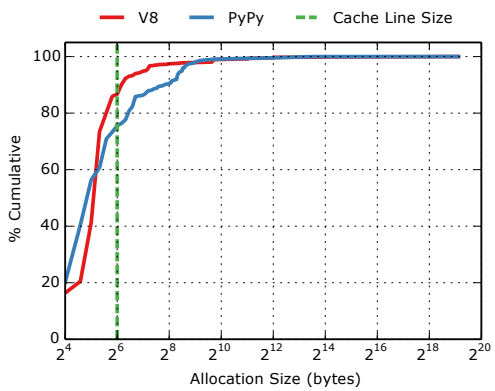


Figure 4. The cumulative distribution of the memory allocation size for PyPy and V8.

The cache installation of invalid memory regions not only reduces unnecessary memory accesses, but also enables us to reduce garbage collection overhead using large nursery sizes. Figure 3 shows the normalized execution time as a function of the nursery size when all initial accesses to invalid (uninitialized or unallocated) nursery locations are somehow identified and directly placed into an on-chip cache without off-chip accesses. Unlike the baseline PyPy, large nursery sizes combined with cache installation can significantly improve the performance. On average, the 64MB nursery with cache installation outperforms the baseline (1MB nursery) by 28.7%. Gcbench, which has high garbage collection overhead runs 69.4% faster with a 64MB nursery. Nqueens, which shows a 2.7x slowdown for the 64MB nursery in the baseline, only shows a 2.7% slowdown with the ideal cache installation. The results suggest that the co-optimization of the nursery size and the initial cache misses for the nursery has a potential for significant performance improvements.

While cache installation helps reduce read memory traffic, it does nothing to reduce write-backs or cache pollution. The newly installed cache lines cause existing cache lines to be evicted. On average, we found that a 64MB nursery results in 3.74x more write-backs compared to a 1MB nursery in PyPy. Additional write-backs usually do not directly affect performance as they happen in the background. Yet, it can be a significant concern for bandwidth-limited systems.

3.5 Memory Allocation Size

In order to use the cache installation, a full cache line must be guaranteed to be uninitialized. In static languages such as C and C++, memory allocations are often larger than a cache line, and existing cache installation mechanisms either explicitly capture memory allocations larger than a cache line using a table [9] or capture a series of stores that overwrite an entire cache line over a short period [5, 23].

For dynamic languages, however, we found that memory allocations are often small. Figure 4 shows the cumulative distribution of the allocation size (in bytes) for PyPy and

V8. The distribution is averaged across all benchmarks (49 for Python and 37 for V8). The vertical line represents the typical cache line size of 64 bytes. For PyPy more than 73.2% of allocated objects are smaller than a cache line. For V8, 85.9% of allocations are smaller than a cache line. The results suggest that the cache installation for dynamic languages must be able to effectively handle small object allocations.

4 Invalid Memory Regions Tracking (IMRT)

We define invalid memory regions as memory locations that are either unallocated or uninitialized. When an object is created, memory is allocated but remains uninitialized. Upon calling the constructor, data is written to the object for initialization. The initialized object can thereafter be used.

For cache installation without loading from memory, we need to guarantee that a whole cache line is from an invalid memory region. Invalid memory regions do not contain useful data and can hold any value without affecting the functionality of the program. Most processor caches associate multiple words with a tag in a single cache line (e.g. 64-bytes). In a typical write-allocate cache design, cache lines which are written to must be first loaded from memory because neighboring words in the same cache line may be later read. If the whole cache line is from an invalid memory region, reading from memory unnecessarily uses memory bandwidth and increases latency.

Explicitly tracking memory allocations is not enough for cache installation in dynamic languages because most allocations are smaller than a cache line. Even if we identify a memory allocation, a cache line should still be read from off-chip memory because some words in the cache line could still be valid.

Instead, we use two features of the typical memory management with generational garbage collection to enable allocating of full cache lines:

1. The young subspace (nursery) is fully unallocated after each garbage collection.
2. The allocation is done in a sequential fashion.

These features ensure that installation does not affect functional correctness by guaranteeing that memory locations above a newly-allocated object are always invalid. If an object is smaller than a cache line, a full cache line is installed into the cache. In the program, however, the object is still allocated in memory at a byte granularity.

To limit the cache pollution, objects should be installed in the cache when they are needed. Premature cache installation may unnecessarily evict useful cache lines. In our approach, we propose to use a small hardware table to track invalid memory regions in a subspace at the cache-line granularity (typically, 64 bytes). Then, invalid memory locations are directly installed in the cache on the first write to the corresponding cache line.

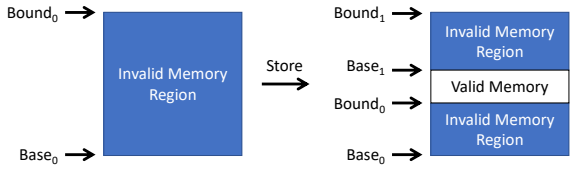


Figure 5. An example of splitting an invalid memory region into two after a memory allocation.

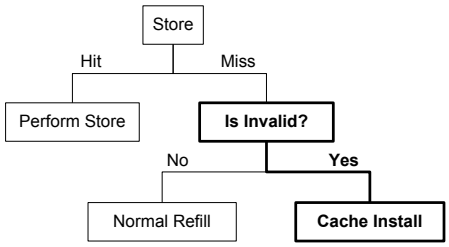


Figure 6. Cache installation decision on a store.

4.1 Tracking Table for Address Ranges

In order to track invalid regions, the software must first tell hardware where the initial invalid memory region such as a young subspace is. For the young subspace, this only needs to be done every garbage collection cycle and when the subspace is first allocated. The software provides the base and bound as full addresses. Hardware keeps cache-line aligned base and bound addresses using a table, and only installs cache lines that are fully covered by the invalid memory region.

Each entry of the tracking table stores a memory range (cache-line aligned base and bound addresses) for one invalid memory region. When software initially provides the base and bound, an entry is added to the table and other entries that fall within the base and bound of the added entry are cleared. The hardware table monitors stores within the invalid region, and updates its entries to maintain the invariant that every range in the table is guaranteed to represent an invalid memory region. Figure 5 shows how the invariant is maintained by splitting one entry into two when there is a store in the middle of an invalid region.

4.2 Handling Tracking Table Evictions

Because a hardware table has a limited size, it may eventually run out of space. In that case, one of the memory ranges need to be evicted. To limit loss of information, an eviction policy that evicts the smallest range should be used.

Even if an entry is evicted, the tracking will still be correct in the sense that there is no false detection of invalid regions. Some stores to invalid memory regions may not be detected and handled normally without cache installation, but the program execution will still be correct.

4.3 Cache Installation

Figure 6 shows how the cache installation using invalid memory region tracking can be done at the LLC (last-level cache)

Table 2. Software interfaces for the tracking hardware.

Invoked By	ISA Instruction	Operation
OS	store <base>[<track_hw_addr>+ set_inv_base_offset]	Sets the base address for an invalid region in a temporary register.
	store <bound>[<track_hw_addr>+ set_inv_bound_offset]	Copies the base (held in a temporary register) and bound for an invalid region to the tracking table, evicting an entry if necessary.
	store <base>[<track_hw_addr>+ set_val_base_offset]	Set the base address for a valid region in a temporary register.
	store <bound>[<track_hw_addr>+ set_val_bound_offset]	Uses the base (held in a temporary register) and bound for a valid region to update the tracking table, splitting an entry if necessary.
Program	syscall <inv><ptr_to_base_bound_array>	The OS will iterate through the array, translate the virtual addresses to physical addresses, and will set the invalid regions in the track table.
	syscall <val><ptr_to_base_bound_array>	The OS will iterate through the array, translate the virtual addresses to physical addresses, and will validate the regions in the track table.

level. On a store miss, the IMRT will be checked. If the requested block is from an invalid memory region, then the cache line will be installed instead of being requested from memory. Placing the tracking table at the LLC removes the need for changing the cache coherence protocol and also enables using the table to identify invalid cache lines for reducing unnecessary cache evictions and write-backs.

Yet, placing the tracking at the LLC-level level introduces a challenge; tracking needs to be performed using physical addresses instead of virtual addresses. This requires OS support for setting the initial invalid region. When the user-level software performs a system call with an invalid memory region in virtual addresses, the OS translates them to physical address regions and sets the tracking table. As a contiguous virtual address range may not map to a contiguous physical address range, the OS may need to set multiple physical address ranges in the tracker.

Since the tracker is managed by the OS and uses physical addresses, multiple processes and threads can share the hardware by invoking the relevant system calls to set the entries. The hardware can additionally be used by the OS when mapping and unmapping pages before, during, and after program execution, and eliminate the need to read in new pages to cache or write-back unused pages from cache.

4.4 Implementation Details

The tracking hardware needs interfaces for both an OS and a memory allocator. These interfaces can be implemented in many ways. Here, we show a design using memory-mapped interfaces. For the OS interface, a unique hardwired physical address is assigned to the tracking table so that the OS can configure tracking hardware.

Table 2 summarizes the software interfaces necessary for the tracking hardware. When user-level software wants to set an invalid memory range, it calls a system call to set the invalid range. Since the tracking will be done automatically in the IMRT, the user-level software only needs to make system calls when setting ranges. The invalid memory range can be cleared by setting a valid range using another system call.

Figure 7 shows the data path of the tracking hardware with four entries. The input to the datapath are two addresses, `addr` and `addr+1`, at the cache-line granularity; for 64-byte

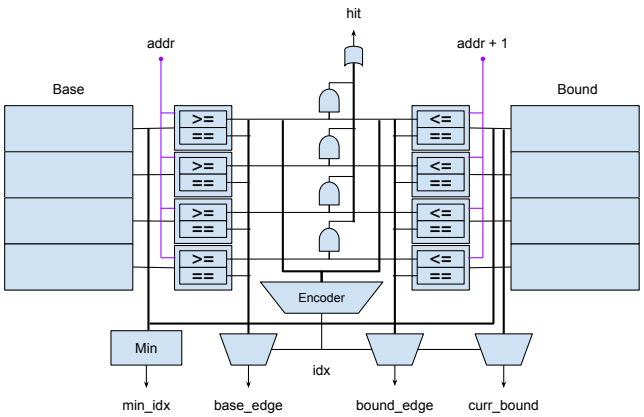


Figure 7. Tracking hardware data path.

Table 3. Pseudo-code for updating base and bound fields.

Base Update	Bound Update
`if (hit && base_edge):` `  base[idx] = addr + 1`	`if (hit && !base_edge):` `  bound[idx] = addr`

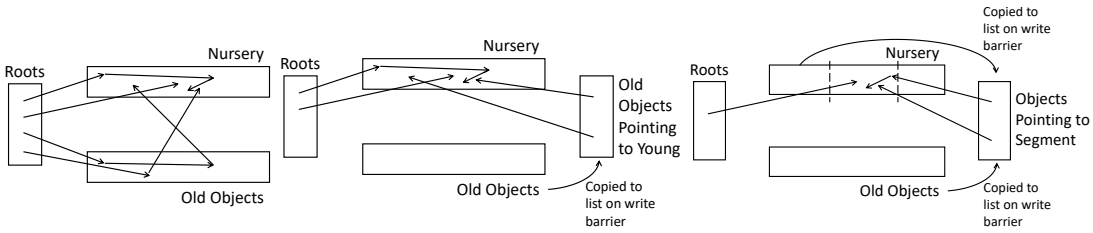
`if (hit && !(base_edge`
`  || bound_edge)):`
`  base[min_idx] = addr + 1`
	`if (hit && !(base_edge`
	`  || bound_edge)):`
	`  bound[min_idx] = curr_bound`

cache lines, `addr` is obtained by removing the 6 LSBs of the memory address for a store. The output is a bit indicating a hit. Note that the bound value is exclusive, so the `addr` needs to be less than the bound for a hit.

The update hardware is not drawn, but follows the logic shown in Table 3. If the `addr` is at the `base_edge`, we only need to update the base at the current index by incrementing `addr` by 1. If the `addr` is at the `bound_edge`, we only need to set the bound to be `addr`, since the bound is exclusive. Finally, if the `addr` is not at an edge, then we must split the entry by setting the existing entry bound to be the `addr` and the new entry to have the range of `addr+1` to the current bound. The hardware is more complex compared to traditional caches because each entry needs to handle an arbitrary memory range. Yet, the number of table entries is quite small compared to caches.

5 Partial Tracing

Although cache installation reduces memory reads, it does not address cache pollution and increased memory writes from a large nursery. Installed cache blocks will still evict



(a) Full Tracing **(b)** Optimized Tracing **(c)** Partial Tracing

Figure 8. Graphical depictions of how tracing works. Arrows represents pointers that are traversed.

other cache lines to make room and need to be written back to memory when evicted. The problem is particularly severe if the nursery is larger than the last-level cache when most cache lines in the nursery will be evicted and written-back to memory before they can be used again after garbage collection.

However, many of these cache evictions and write-backs may be unnecessary. Following the generational hypothesis that most objects die young, there is a strong likelihood that many objects are dead before they are evicted from the cache. If the dead objects can be identified, we can avoid unnecessary eviction of valid cache lines and unnecessary write-backs of dead objects.

For this purpose, we introduce a technique named partial tracing, which identifies dead objects in a subset of a nursery with a goal to optimize cache replacements and write-backs.

5.1 Partial Tracing Algorithm

We propose to adapt the tracing algorithm used during the nursery garbage collection to determine live objects for a segment of the nursery that is most likely to be in the cache when this partial tracing is performed. We divide the nursery into segments that would fit in the cache. During execution, we sequentially allocate space for objects as needed. Once the full segment has been allocated, we run tracing to determine which objects are live only for that segment. This information is communicated to the hardware and used for cache replacement and removing unnecessary write-backs.

One challenge in performing tracing to identify live objects for only a segment of the nursery is that we need to follow pointers for all live objects as shown in Figure 8(a). Not only is the computation unnecessary but we may also pollute the cache by accessing all of the live objects. As shown in Figure 8(b), generational garbage collectors solve a similar problem using a write barrier to limit tracing of old objects to only those old objects that contain pointers to young objects in the nursery. The write barrier works by issuing a callback function when a write to any pointer in the old object occurs. The callback function adds the old objects to a list which is then used as a starting point for tracing instead of having to perform complete tracing through the program roots.

As shown in Figure 8(c), we extend the write barriers to young objects in previous segments within a nursery. When we perform tracing on a segment and find the live objects in that segment, we add a write barrier to the pointers of those live objects. If a pointer in any young object in a previous segment is written to, the write barrier invokes a callback to add those objects to a trace list. We also modify the callback for old objects to only add objects that point to one segment rather than the whole nursery. The list containing old objects and young objects from previous segments can be used to perform tracing for only the segment in an efficient manner.

Decoupling tracing from the full garbage collection also opens the door for running a part of the garbage collection, namely tracing, concurrently with the main application. As discussed before, the copying garbage collectors need to stall the main application in order to copy objects and update pointers. However, partial tracing, which only identifies live objects, does not need to pause the application and can be performed concurrently without incurring performance overhead if an extra core is available.

5.2 Identifying Dead Cache Lines

After performing partial tracing, we have a list of live objects in the recently-allocated segment of a nursery that is likely to be in the on-chip cache. Using information about their start addresses and object sizes, we can determine which memory ranges are valid. We can additionally use the base and the bound of the segment to determine which memory ranges in the segment are invalid. We start by assuming the whole segment is invalid. Using the ranges of valid memory ranges, we can split the initial invalid segment into smaller invalid ranges. Once we iterate through all of the valid memory ranges, we are left with an accurate list of invalid memory regions.

5.3 Integration with IMRT

The IMRT design that we described in the previous section can be used to track invalid regions for both unallocated/uninitialized nursery regions and dead objects after partial tracing. As shown in Table 2, the IMRT interface allows software to add valid and invalid regions to the table. To add the invalid regions from the partial tracing, software first sets the traced segment as an invalid region in the IMRT. Then, each live object after the tracing is added as a valid region to break the segment into multiple invalid regions.

Table 4. Microarchitectural parameters for simulations.

Core	4-way OOO, 2.66GHz
L1I	32kB, 4-way, 4-cycle latency
L1D	32kB, 8-way, 4-cycle latency
L2	256kB, 8-way, 10-cycle latency
L3	2MB, 16-way, 40-cycle latency
Memory	Micron DDR3-1333

Table 5. Summary of the designs used for evaluation.

Design	Description
Base	Baseline with no optimization.
IMRT	Cache installation with a 256-entry tracking table.
IMRT+PTO	Cache installation with a 256-entry tracking table along with partial tracing optimization.
IMRT+CPTO	Cache installation with a 256-entry tracking table along with concurrent partial tracing optimization

5.4 Cache Eviction and Write-backs

To reduce unnecessary cache pollution and write-backs, the IMRT is referenced for cache replacements and write-backs in addition to memory reads (for cache installation). The cache replacement policy prioritizes eviction of cache lines that belong to an invalid memory region so that live objects can stay longer in the cache for reuse. The memory addresses from replacement candidates are looked up in the IMRT on a cache miss, and invalidated if found in the IMRT. The invalid cache lines are replaced first before evicting valid cache lines. Note that the IMRT look-ups can be performed in the background over a long LLC miss latency without affecting performance. The IMRT table is also referenced on a write-back. If the address of the evicted (dirty) cache line is found in IMRT, the write-back will be eliminated.

6 Evaluation

6.1 Methodology

For the evaluation, we use a simulation infrastructure that is based on ZSim [24]. ZSim is used to model cycle-level microarchitecture behaviors of an out-of-order core with memory hierarchy comparable to modern processors. The processor core is configured to mimic an Intel Westmere processor. For the DRAM memory system, we use DRAMSim2 [21] integrated to ZSim to model DDR3-1333 memory.Table 4 summarizes the architecture parameters.

For the implementations of dynamic languages, we use PyPy [1] for Python and V8 [4] for Javascript. Both Python and Javascript are widely used in practice, with Python being used in a range of applications from web servers to scientific computing and Javascript being primarily used for web applications. PyPy and V8 represent the state-of-art implementations for Python and Javascript. Both use just-in-time compilation to achieve good performance and use a variation of generational garbage collection.

For benchmarks, we use a wide array of applications to get a representative sample of real-world applications. For Python, we use benchmarks from the official Python performance benchmark suite [19] and benchmarks from the PyPy benchmark suite. The designers say that the Python benchmark suite focuses on real-world benchmarks, using whole applications when possible, rather than synthetic benchmarks [19]. For Javascript, we use JetStream [2], which "combines a variety of JavaScript benchmarks, covering a variety of advanced workloads and programming techniques" including SunSpider, Octane, and LLVM. In total, we use 49 benchmarks for Python and 37 benchmarks for Javascript. For Python, we warm up the benchmark by running it 2 times followed by running it 3 times for evaluation. For Javascript, we run the benchmark 3 times for evaluation.

Table 6. The coverage of a limited-size IMRT table compared to the infinite size table.

	4	8	16	32	64	128	256
PyPy	77.6%	84.7%	89.4%	92.9%	95.2%	96.1%	97.1%
V8	79.5%	79.7%	88.5%	94.7%	96.5%	97.9%	98.8%

We evaluate the design points shown in Table 5. The baseline (Base) represents the case without any optimization. IMRT represents the invalid memory region tracking mechanism in this paper where only cache installation is enabled. IMRT+PTO represents the case where both cache installation and partial tracing are enabled. Because the partial tracing optimization requires a significant re-write of a garbage collector, we currently have a partial tracing implemented only for PyPy, but not for V8. IMRT+CPTO represents the case where partial tracing is run concurrently with the normal program with cache installation. Note that we do not have an actual implementation of concurrent tracing but model it by subtracting the computation overhead of running tracing from the execution. The cache pollution from tracing is still included. For the baseline, we use the default 1MB nursery for PyPy and V8. For our optimizations, we use a 64MB nursery for PyPy and 128MB nursery for V8. For our average speedup calculations, we use geometric mean.

6.2 Tracking Table Size

For this study, we evaluate how many invalid memory addresses our tracking table can capture with a limited storage size compared to the ideal case with an unlimited storage. For this purpose, we compute the percentage of memory reads to invalid memory regions that are captured by a given IMRT table size for cache installation compared to the total number of reads to invalid memory regions.

The results shown in Table 6 suggest that a small tracking table can capture nearly all of the invalid memory ranges. Since the memory allocation of objects happens sequentially and initialization also happens mostly sequentially, most memory allocations only need to update the base address of an existing entry rather than creating an entry of a new memory range. An update at the boundary does not split an entry into multiple ranges, so no additional space is required in the tracking table.

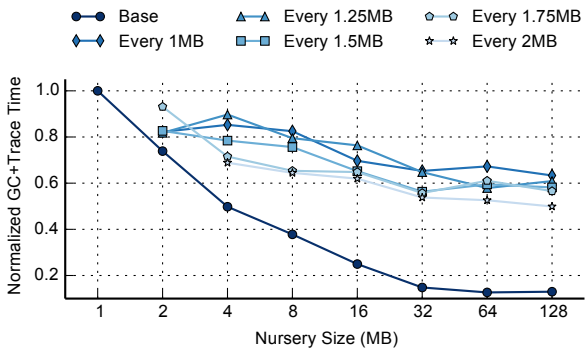


Figure 9. Normalized execution time for garbage collection and partial tracing.

A first-order evaluation shows that a 256-entry table would have minimal area and power overhead. The main overhead of the table comes from the memory required to store the base and bound addresses (i.e. a pair of 64-bit addresses). This would equate to 4kB of memory and could be reduced by compression of the addresses. For our first-order evaluation, we use CACTI [13], an integrated model for cache and memory access time, cycle time, area, leakage, and dynamic power. If we model our IMRT as a fully associative 4kB cache, it uses $0.038mm^2$ area and has $6mW$ leakage power on a $22nm$ node. For comparison, the 32kB L1 data cache would use $0.408mm^2$ area and would have $15mW$ leakage power on a $22nm$ node.

6.3 Partial Tracing Period

Here, we study the overhead of partial tracing as we vary nursery sizes and partial tracing frequencies. We normalize the execution time of the garbage collection with and without partial tracing at each nursery size to the baseline garbage collection (no partial tracing) with a 1MB nursery. For reference, garbage collection with a 1MB nursery is on average 16.0% of the total program execution time. Figure 9 shows how the baseline garbage collection overhead (Base) decreases as the nursery size increases. With a 64MB nursery, garbage collection overhead is reduced by an average of 87.4%.

Adding partial tracing on top of garbage collection enables cache optimizations for replacements and write-backs, but reduces the savings from a large nursery. In the figure, 'Every x-MB' indicates the case where partial tracing is performed once every x-MB in addition to the full garbage collection when the entire nursery gets full. More frequent partial tracing has a potential to more quickly identify and remove dead objects in the cache, but also has higher overhead. However, the results suggest that the performance differences among different tracing frequencies are rather small. In the following studies, we use the partial tracing period of every 1.25MB for IMRT+PTO. This configuration has a potential to remove 42.2% of the baseline garbage collection overhead.

6.4 Overall Performance

In this study, we evaluate the overall performance improvements of the proposed optimizations. The performance is presented as the execution time normalized to the baseline. For the IMRT schemes, we use the table size of 256 entries to support both cache installation and write-back elimination. For PyPy, we show results for both 1MB and 64MB nursery sizes in order to separately evaluate the improvements from cache installation and less-frequent garbage collection.

Figure 10 shows the normalized execution time of the various designs for each Python benchmark. The results show that simply increasing the nursery (Base-64M) leads to a significant slowdown in many applications. On the other hand, cache installation with a 1MB nursery (IMRT-1M) only reduces the execution time by 4.7% on average. This shows that cache installation by itself does not lead to significant performance improvements. Using a large (64MB) nursery with cache installation (IMRT-64M) reduces the execution time by 22% on average compared to the baseline with only a single benchmark showing a noticeable slowdown. Moreover, the execution time is reduced by over 50% for multiple benchmarks. The average improvement is within 2.0% of the possible improvement with an ideal tracker. The results show the importance of co-optimizing garbage collection period with cache installation.

Cache installation with partial tracing (IMRT+PTO) shows lower performance improvements on average compared to IMRT due to the overhead of additional tracing. However, IMRT+PTO can still significantly reduce the execution for many applications. Moreover, for applications where the impact on cache performance of a large nursery is particularly significant, IMRT+PTO outperforms IMRT, reducing the execution time by an additional 5.9% for `spitfire_cstringio` or 1.4% for `meteor_contest`.

If partial tracing is performed concurrently with the application (IMRT+CPTO), then the execution time is always better than IMRT, since the partial tracing improves cache performance and there is little sequential execution overhead for running it. As a result, there is an additional 1.3% average reduction in the execution time over IMRT. By running partial tracing concurrently, we can achieve significant performance improvement while simultaneously reducing memory accesses.

Figure 11 shows the normalized execution time for V8 when the IMRT hardware is used for cache installation with the 128MB nursery size. Cache installation alone reduces the average execution time by 8.2%. When combined with the larger nursery size, the average execution time is reduced by 16.8% with some benchmarks showing reductions more than 40%. The results show that the proposed hardware tracker is general enough to be applied to multiple languages and implementations beyond PyPy.

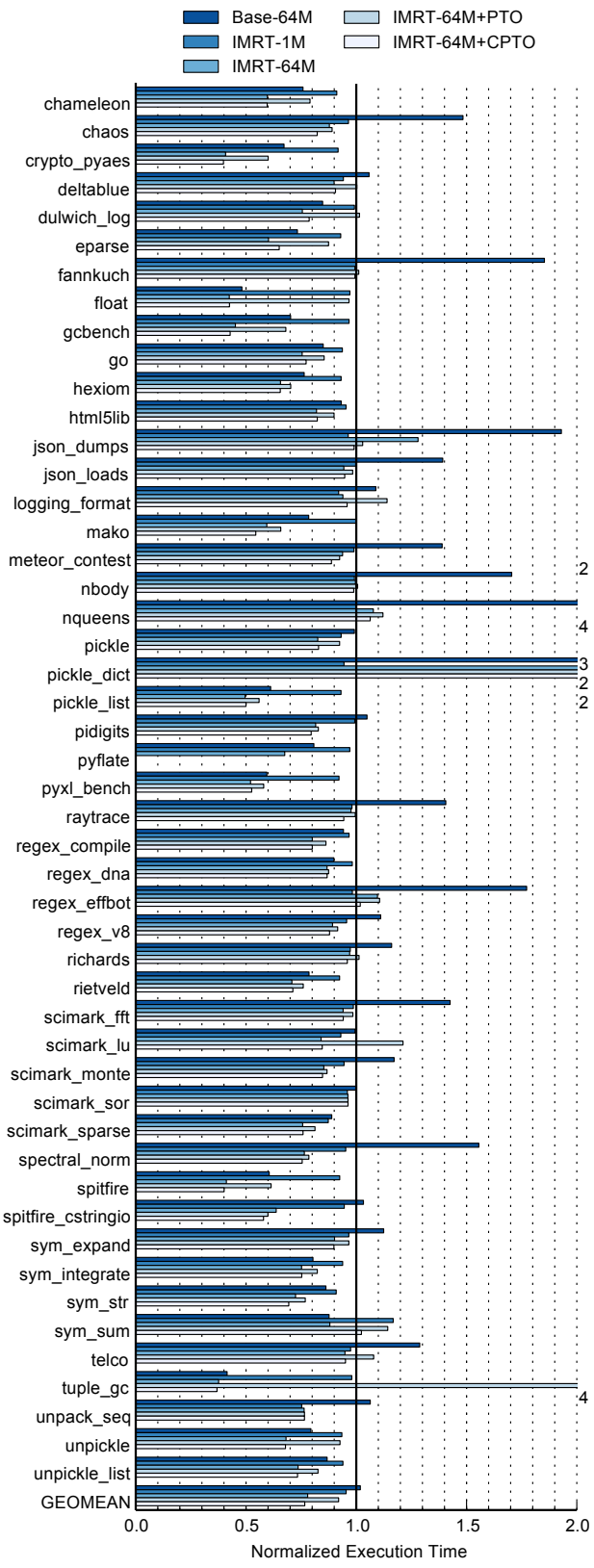


Figure 10. Normalized execution time for PyPy. The execution time is normalized to the baseline with a 1MB nursery.

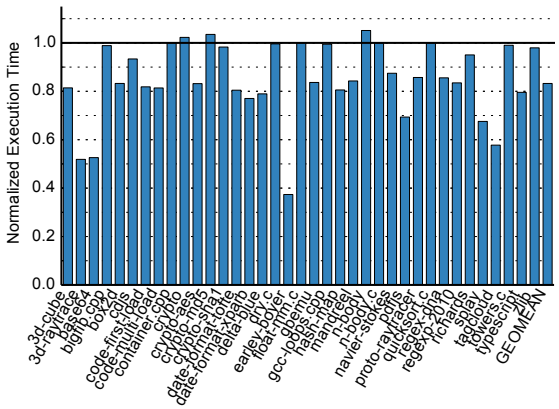


Figure 11. Normalized execution time for V8. The execution time is normalized to the baseline with a 1MB nursery.

Table 7. Breakdown of cache miss rates at the LLC.

	PyPy			V8	
	Base-64M	**IMRT-64M**	**IMRT-64M+PTO**	**Base-128M**	**IMRT-128M**
Overall	60.2%	17.6%	17.4%	48.2%	21.2%
Non-Nursery	29.5%	29.5%	26.1%	34.2%	26.5%
Nursery	89.7%	9.5%	9.2%	70.6%	13.8%

6.5 Cache Miss-Rate Breakdown

Here, we study the impact of our optimizations on the last-level cache (LLC) miss-rates for different memory regions. Table 7 shows a breakdown of the cache miss-rates for PyPy and V8 for the baseline and the proposed tracker with the same nursery size. The miss-rates are shown for all accesses (overall), nursery accesses, and non-nursery accesses. The results show that the proposed cache installation is indeed quite effective in reducing the cache misses to the nursery. For PyPy, the results suggest that the partial tracing optimization can reduce cache pollution and slightly improve the cache miss rate for both nursery and non-nursery accesses. For V8, we used the IMRT table for both nursery and some of non-nursery allocations as reflected in the reduced miss-rate for the non-nursery accesses.

6.6 Off-Chip Memory Operations

In addition to performance improvements, the proposed optimizations also reduce off-chip memory accesses. Less memory accesses reduce energy consumption in memory. Lower memory bandwidth usage is also important for bandwidth-limited applications or systems where many applications share a memory channel. Figure 12 shows the number of memory operations (reads and writes) for various PyPy benchmarks. Here, only benchmarks where the memory bandwidth utilization is more than 5% of the the maximum memory bandwidth are shown. While the IMRT scheme can reduce memory reads through cache installation, the partial tracing optimization can further reduce the memory writes, especially when there are many dead cache lines

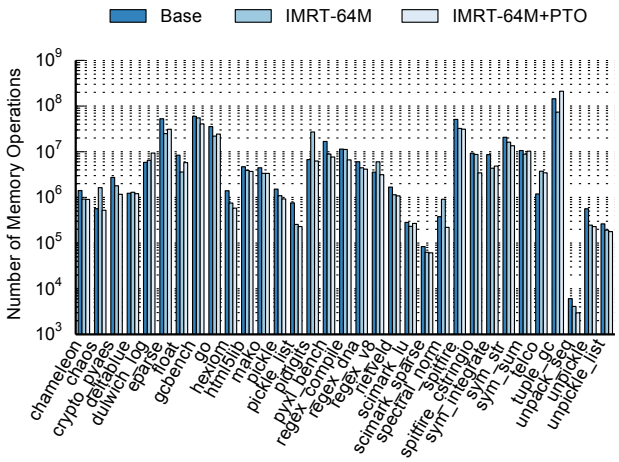


Figure 12. Total off-chip memory operations for PyPy.

Table 8. Configuration of the low-end processor.

Core	4-way OOO, 2.17GHz
L1I	24kB, 6-way, 3-cycle latency
L1D	32kB, 8-way, 3-cycle latency
L2	512kB, 16-way, 14-cycle latency
Memory	6400 MB/s mem with 210-cycle latency

being written back and evicted. On average, partial tracing reduces memory operations by an additional 13.7%. For some benchmarks, the reduction is far more significant (over 4x for `pidigits`). There are some cases where the total number of memory operations increase with partial tracing due to the memory accesses from tracing itself.

6.7 Microarchitectural Sweeps

Here, we study how the performance is affected by a different processor configuration and last-level cache sizes. In this study, we use a configuration for a low-end processor similar to the Intel Silvermont (see Table 8). Low-end processors often have smaller caches and less tolerant to memory latency due to smaller buffers and queues. As a result, these processors are more sensitive to memory performance and we expect our optimizations to be more effective.

Figure 13 shows the average normalized execution time results for PyPy. In the baseline, the nursery size is adjusted to be half of the cache size. With a small cache, garbage collection is run more frequently. As the cache size increases, garbage collection becomes less frequent and a proportionately larger space is available for the non-nursery accesses. From the results, we find that the proposed cache installation scheme with a large nursery size (IMRT-64M with 512kB LLC) can improve the performance as much as increasing the cache size by 8x (Base with 4MB LLC). Furthermore, we can reduce the average execution time by 40% if we have a 512kB LLC, significantly more than in a high-performance processor. Alternatively, our scheme can perform just as well with half of the nursery size as the base case (IMRT-1M with 4MB LLC vs. Base with 4MB LLC).

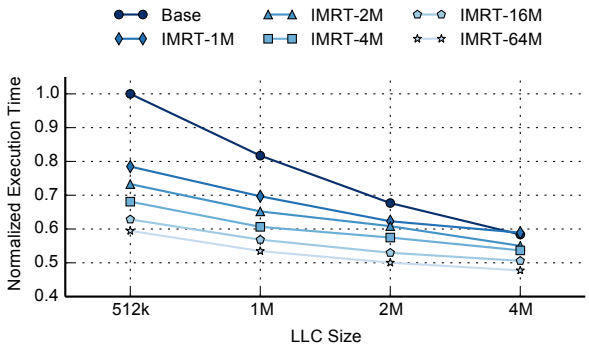


Figure 13. Normalized execution time of a low-end processor with different LLC and nursery sizes for PyPy.

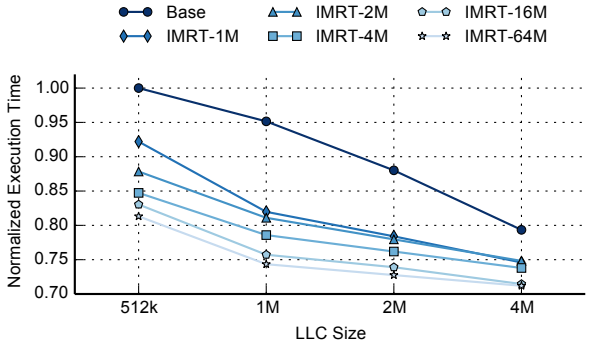


Figure 14. Normalized execution time of a low-end processor with different LLC and nursery sizes for v8.

Figure 14 shows the average normalized execution time results for V8. For the baseline, we fix the nursery size to be 1MB (the minimum V8 allows). As the cache size increases, the cache performance improves and the execution time decreases. Similar to PyPy, we can achieve the same performance with a larger nursery (IMRT-64M with 512kB LLC) as 8x the cache size (Base with 4MB LLC). In addition, we can reduce the average execution time by 19% if we have a 512kB LLC, slightly more than in a high-performance processor.

The results indicate that the cache performance is more important on low-end processors than high-end processors, and the proposed optimizations lead to more significant performance improvements for processors with smaller caches.

7 Conclusion

In this paper, we present how memory management in dynamic languages can be optimized using both hardware and software together. In particular, we show that a large nursery size can be used to significantly reduce garbage collection overhead if its impact on cache performance can be controlled. We introduce a new invalid memory region tracker and partial tracing to address this cache performance issue. Overall, our study shows that this hardware-software co-optimization is general enough to be applicable to multiple dynamic languages (PyPy and V8) and can significantly improve performance.

A Model of Garbage Collection Execution Time

We build a first order model of the total garbage collection time, denoted T_{GC}. For this model, we are only concerned with nursery collection of the young space and not the full garbage collection of the old space. The total garbage collection time can be broken down into a series of garbage collection invocations:

$$T_{GC} = N * t_{GC} \tag{1}$$

where N is the number of times garbage collection is run and t_{GC} is the average time per invocation.

Garbage collection is triggered when the memory allocated by the mutator matches the nursery size:

$$N = \frac{S_{PA}}{S_N} \tag{2}$$

where S_{PA} is the total number of bytes allocated by the mutator in the nursery and S_N is the nursery size.

We can break down the time it takes to run a garbage collection invocation:

$$t_{GC} = t_{setup} + t_{cleanup} + (t_{trace} + t_{move})\, N_{live} \tag{3}$$

where t_{setup} and $t_{cleanup}$ are fixed times for each garbage collector invocation and t_{trace} and t_{move} are the time it takes to trace and move a single object. N_{live} is the number of objects that need to be moved.

We can further break down N_{live} by modeling it as an exponential decay function:

$$N_{live} = \frac{S_N}{S_O} e^{-\lambda t} \tag{4}$$

where S_O is the average size of an object and λ is the death rate of objects.

Since garbage collection is performed at intervals, we can set t to be the interval between garbage collection invocations:

$$t = \frac{S_N}{R_{PA}} \tag{5}$$

where R_{PA} is the rate the mutator allocates nursery space in *bytes/cycle*.

Through substitution, we can combine the above equations to get the following:

$$T_{GC} = \frac{S_{PA}}{S_N}\left[t_{setup} + t_{cleanup} + (t_{trace} + t_{move})\frac{S_N}{S_O} e^{-\lambda \frac{S_N}{R_{PA}}}\right] \tag{6}$$

In the equation, there are some variables that are dependent on the application. λ is fully dependent on the application that is running. S_{PA}, S_O, R_{PA} are also dependent on the application that is running, but we can make minor adjustments by changing the object space implementation in the run time. Decreasing the size of the object will decrease the total space allocated by the program and the allocation rate. This will result in an overall reduction in the garbage collection time.

In terms of the garbage collector implementation, we have control over t_{setup}, $t_{cleanup}$, t_{trace}, t_{move}, and S_N. Improving the first four would require optimization of the garbage collector. S_N can easily be adjusted before each run or even at runtime.

By increasing the nursery size (i.e. S_N), we are spreading t_{setup} and $t_{cleanup}$ over a longer time period. In other words, if we increase the nursery size by M, the contribution over the same time period becomes $\left(t_{setup} + t_{cleanup}\right)/M$.

Interestingly, the trace and move times are not affected by the nursery size in the same way. The larger nursery size makes the magnitude of the exponent larger indicating a larger decay constant. This means that more live objects will die between garbage collections with a larger nursery size. As a result, the contribution from tracing and moving will be smaller.

The observations from the equation match our intuition. A larger nursery will allow us to run garbage collection less frequently and there will be less live objects to trace and move during each execution. As a result, the total garbage collection overhead will decrease.

Acknowledgments

This work was partially supported by the Office of Naval Research grant N00014-15-1-2175.

References

[1] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*. ACM, 18–25.

[2] Browserbench. 2017. JetStream 1.1. (2017). http://browserbench.org/JetStream/

[3] Coding Dojo. 2016. The 9 Most In-Demand Programming Languages of 2016. (2016). http://www.codingdojo.com/blog/9-most-in-demand-programming-languages-of-2016/

[4] Google. 2018. Chrome V8. (2018). https://developers.google.com/v8/

[5] Shiwen Hu and Lizy John. 2006. Avoiding store misses to fully modified cache blocks. In *Performance, Computing, and Communications Conference, 2006. IPCCC 2006. 25th IEEE International*. IEEE, 8–pp.

[6] IBM. 1994. *PowerPC Microprocessor Family: The Programming Environments*. IBM Microelectronics, Motorola Corporation.

[7] Spectrum IEEE. 2017. The 2017 Top Ten Programming Languages. (Jul 2017). https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages

[8] José A Joao, Onur Mutlu, and Yale N Patt. 2009. Flexible reference-counting-based hardware acceleration for garbage collection. In *ACM SIGARCH Computer Architecture News*, Vol. 37. ACM, 418–428.

[9] Jarrod A Lewis, Bryan Black, and Mikko H Lipasti. 2002. Avoiding initialization misses to the heap. In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*. IEEE, 183–194.

[10] Matthias Meyer. 2004. A novel processor architecture with exact tag-free pointers. *IEEE Micro* 24, 3 (2004), 46–55.

[11] Matthias Meyer. 2005. An on-chip garbage collection coprocessor for embedded real-time systems. In *Embedded and Real-Time Computing Systems and Applications, 2005. Proceedings. 11th IEEE International Conference on*. IEEE, 517–524.

[12] Sun Microsystems. 2006. Memory Management in the Java HotSpot Virtual Machine. (2006). http://www.oracle.com/technetwork/java/javase/tech/memorymanagement-whitepaper-1-150020.pdf

[13] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. 2009. CACTI 6.0: A tool to model large caches. *HP laboratories* (2009), 22–31.

[14] Kelvin D Nilsen and William Schmidt. 1996. System and hardware module for incremental real time garbage collection and memory management. (Sept. 24 1996). US Patent 5,560,003.

[15] Stephen O'Grady. 2016. The RedMonk Programming Language Rankings: January 2016. (Feb 2016). http://redmonk.com/sogrady/2016/02/19/language-rankings-1-16/

[16] Hannes Payer and Ross McIlroy. 2015. Getting Garbage Collection for Free. (2015). https://v8project.blogspot.com/2015/08/getting-garbage-collection-for-free.html

[17] Chih-Jui Peng and Gurindar S Sohi. 1989. *Cache memory design considerations to support languages with dynamic heap allocation.* University of Wisconsin-Madison. Computer Sciences Department.

[18] The PyPy Project. 2014. Garbage Collection in PyPy. (2014). https://pypy.readthedocs.io/en/release-2.4.x/garbage_collection.html

[19] PyPerformance. 2017. Python Performance Benchmark Suite. (2017). http://pyperformance.readthedocs.io/

[20] Vimal K Reddy, Richard K Sawyer, and Edward F Gehringer. 2006. A cache-pinning strategy for improving generational garbage collection. In *International Conference on High-Performance Computing*. Springer, 98–110.

[21] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. 2011. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Computer Architecture Letters* 10, 1 (Jan 2011), 16–19. https://doi.org/10.1109/L-CA.2011.4

[22] Rubinius. 2013. Concurrent Garbage Collection. (Jun 2013). https://github.com/rubinius/rubinius-archive/blob/cf54187d421275eec7d2db0abd5d4c059755b577/_posts/2013-06-22-concurrent-garbage-collection.markdown

[23] Hou Rui, Fuxin Zhang, and Weiwu Hu. 2005. A memory bandwidth effective cache store miss policy. In *Asia-Pacific Conference on Advances in Computer Systems Architecture*. Springer, 750–760.

[24] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-core Systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 475–486.

[25] Jennifer B. Sartor, Wim Heirman, Stephen M. Blackburn, Lieven Eeckhout, and Kathryn S. McKinley. 2014. Cooperative Cache Scrubbing. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT '14)*. ACM, New York, NY, USA, 15–26.

[26] William J Schmidt and Kelvin D Nilsen. 1994. Performance of a hardware-assisted real-time garbage collector. *ACM SIGOPS Operating Systems Review* 28, 5 (1994), 76–85.

[27] Jonathan Shidal, Zachary Gottlieb, Ron K Cytron, and Krishna M Kavi. 2014. Trash in cache: detecting eternally silent stores. In *Proceedings of the workshop on Memory Systems Performance and Correctness*. ACM, 8.

[28] Jonathan Shidal, Ari J Spilo, Paul T Scheid, Ron K Cytron, and Krishna M Kavi. 2015. Recycling trash in cache. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 118–130.

[29] Mohit Tiwari, Banit Agrawal, Shashidhar Mysore, Jonathan Valamehr, and Timothy Sherwood. 2008. A Small Cache of Large Ranges: Hardware Methods for Efficiently Searching, Storing, and Updating Big Dataflow Tags. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 41)*. IEEE Computer Society, Washington, DC, USA, 94–105.

[30] Maira Wenzel, Alan Dawkins, Luke Latham, Tom Pratt, Mike Jones, Michal Ciechan, and Xaviex. 2017. Fundamentals of Garbage Collection. (Mar 2017). https://msdn.microsoft.com/en-us/library/ee787088(v=vs.110).aspx

[31] Paul R Wilson, Michael S Lam, and Thomas G Moher. 1992. Caching considerations for generational garbage collection. In *ACM SIGPLAN Lisp Pointers*. ACM, 32–42.

[32] David S Wise, Brian Heck, Caleb Hess, Willie Hunt, and Eric Ost. 1997. Research demonstration of a hardware reference-counting heap. *Lisp and Symbolic Computation* 10, 2 (1997), 159–181.

[33] Xi Yang, Stephen M. Blackburn, Daniel Frampton, Jennifer B. Sartor, and Kathryn S. McKinley. 2011. Why Nothing Matters: The Impact of Zeroing. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '11)*. ACM, New York, NY, USA, 307–324. https://doi.org/10.1145/2048066.2048092

[34] Yi Zhao, Jin Shi, Kai Zheng, Haichuan Wang, Haibo Lin, and Ling Shao. 2009. Allocation Wall: A Limiting Factor of Java Applications on Emerging Multi-core Platforms. *SIGPLAN Not.* 44, 10 (Oct. 2009), 361–376. https://doi.org/10.1145/1639949.1640116