FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

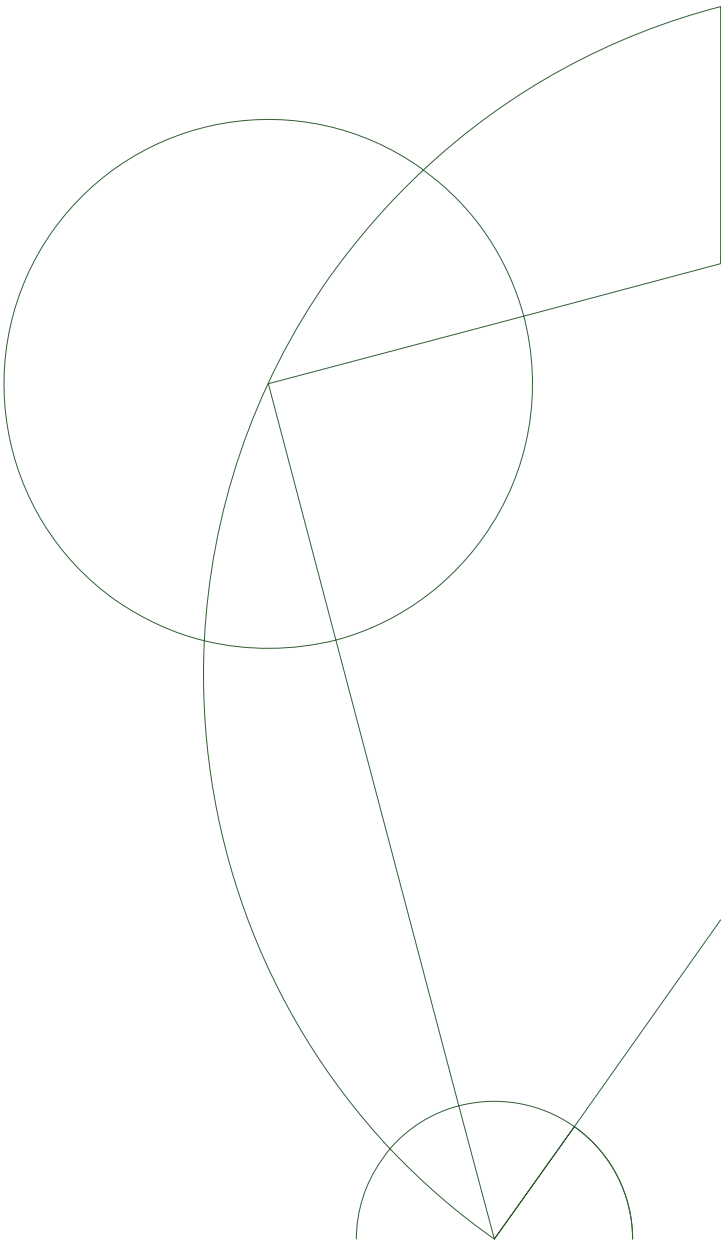
Bachelor Thesis

Cache optimizations in garbage collection

Opportunistically selecting cache-frindly object layouts

David Himmelstrup <vrs552@alumni.ku.dk>

Supervisor
Fritz Henglein <henglein@diku.dk>



Abstract

Automatic garbage collection is widely used and has been shown to be very efficient is given enough physical memory. With memory prices like to continue their downward trajectory of the last couple of decades, automatic garbage collection is set to become even more competitive against manual memory management in the future. However, memory bandwidth hasn't improved at the same pace as price and latency has even gone up. This has lead to the development of memory caches closer to the CPU but traditional garbage collection algorithms do not make good use of these. This report explores two new techniques, named tail-copying and tail-compaction, that extend Cheney's semi-space garbage collection algorithm to improve cache friendliness and therefore performance. Measurements of 7 benchmark programs show that the two techniques are able to reduce the number of cache misses but the performance benefits are often lost due to increased branch misses.

Contents

Chapter 1

Introduction

High level, memory managed languages offer many productivity advantages because developers don't have to worry about how objects are arranged in memory. However, memory layout can hugely affect performance so the productivity advantages can come with efficiency disadvantages. Much research has been done not only to quantify this discrepancy but also to narrow the gap.

In this thesis, I will be looking at the memory layout of semi-space garbage collector in the context of a purely functional programming language. I will implement two optimizations in the garbage collector and evaluate these optimizations against the baseline. These two novel techniques, as well as the detailed measurements of their effects, are the primary contributions of this report.

1.1 Jikes RVM vs. GHC vs. native

Jikes Research Virtual Machine is an environment for running Java programs and it provides an excellent platform for experimenting with new garbage collection techniques, called MMTk (Memory Management Toolkit).[1] While Jikes RVM has been the starting point of a lot of solid research, I've decided not to use it because Haskell[i] (a purely functional language) has different allocation patterns than Java (an object-oriented language).[2–5] Objects in purely functional languages tend to be smaller and allocated at a faster rate than objects in object-oriented languages.

Furthermore, I will not be using the mature Haskell compiler, GHC, since the complexity of its runtime-system (which spans more than 100k lines of code) makes it difficult to modify and reason about. Instead, I will use the LHC compiler. It only supports a subset of the Haskell 2010 standard but its runtime-system is much smaller and more nimble. Key design choices of LHC (and how they differ from GHC's) are explained in chapter 2.

[i] Haskell 2010 Language Report: `https://www.haskell.org/onlinereport/haskell2010/`

1.2 Research Aims

I have implemented a new semi-space garbage collector for a subset of the Haskell programming language. This collector contains two novel optimizations that modify how objects are arranged in memory. It is my hypothesis that these two optimizations will improve how the garbage collector interacts with the memory caches. As part of this work I want to answer the following questions:

- Is the implementation unreasonably complicated? Some previously proposed systems have shown promise on a small scale but were too complicated to gain widespread usage.[ii]
- Are the optimizations beneficial and do the benefits outweigh the costs?
- Can the optimizations be ported to collectors used in industry?

1.3 Thesis organization

Chapter 2 goes over the current approaches to garbage collection, how people have tried to solve the problem in the past, and provides details about LHC which are required by the later chapters. Chapter 3 describes the design and implementation of the main contribution of this thesis, as well as the design and implementation of further optimization that follows as a direct consequence of the tail-copy technique. Chapter 4 evaluates these two new approaches against a baseline collector. This evaluation looks at deterministic factors which are proxies for performance. Chapter 5 concludes the work and presents possible future work.

[ii] Immix Garbage Collector in GHC: https://ghc.haskell.org/trac/ghc/wiki/Commentary/Rts/Storage/GC/Immix

Chapter 2

Background

2.1 Processors, memory and cache

CPUs used to run at similar frequencies as main memory, allowing them to read from memory at roughly the same speed that it could process the information. Sometimes programs have to process larger data sets than can fit into memory and parts of the data set would have to be swapped out of main memory and onto a disk or tape. A lot of caching research from that time was focused on how to manage the virtual memory using limited physical memory (which supports direct access with a constant latency) and a spinning disk drive (which requires seeking).

In recent years CPUs have gotten a lot faster, the bandwidth of memory hasn't been able to keep pace and the memory latency has even gotten worse.[2,6–8] This means that fetching data from memory can easily be the bottleneck when processing a data set. To solve this, several levels of memory caches were inserted closer to the CPU. Figure 2.1 illustrates this hierarchy. These caches work in levels with the lowest levels being smallest and fastest. Cache latency (ie. number of cycles it takes to fetch a word) can easily be two

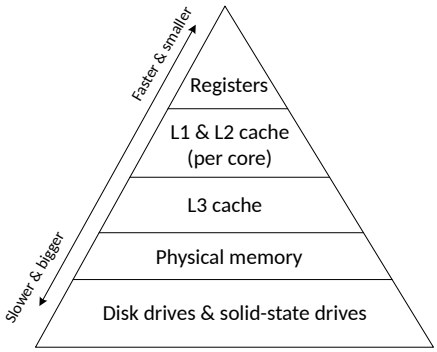


Figure 2.1: Memory hierarchy.

Table 2.1: Approximate access times for memory in an Intel Pentium M

To Where	Cycles
Register	≤ 1
L1d	~ 3
L2	~ 14
Main Memory	~ 240

orders of magnitude lower than that of the main memory. Table 2.1 shows latency numbers from Ulrich Drepper's "What Every Programmer Should Know About Memory."[9] The caches also have higher bandwidth than main memory but the benefit is less extreme here.

The introduction of caches meant that memory fetches no longer completed in a constant amount of time because of the difference between a cache hit and a cache miss. Just as seen with spinning disks earlier, certain access patterns and data layouts can significantly improve performance.[6] The canonical problems that highlight this is matrix multiplication: The elements of the resulting matrix can be computed in any order but choosing a cache-friendly access pattern can be orders of magnitude faster.[6,8]

2.2 Manual memory management and reference counting

Manually allocating and deallocating memory is still widely used in low-level languages and on embedded systems. The onus is on the user to ensure correct usage, and failure to do so can lead to crashes, security breaches, or even performance problems. Modern languages are usually garbage collected (8/10 most popular languages on github[i] in 2017 were garbage collected). Manual management has low overhead and can finalize objects immediately when they're no longer used. This is important for limited resources such as file handlers, file locks, etc. Manual management often has more predictable performance without arbitrary pauses but performs less well in terms of throughput and locality.

In the reference counting scheme, each allocated object is annotated with how many references point to it. Each time the number is used, this number increases. Once it hits zero, the object is deallocated and the reference count of all its children is decreased. A naive reference-counting garbage collector will run into problems with circular references and such cycles will never be deallocated, even when they're not reachable from the root-set anymore. It is, however, possible to combine reference counting with occasional tracing from the root-set to find and deallocate cycles.

Predictable performance (even if not stellar) is a benefit of reference counting but the overhead of the per-object reference count can be significant. The

[i] GitHub Octoverse: https://octoverse.github.com/2017/

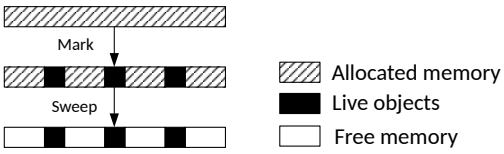


Figure 2.2: Mark-and-Sweep.

garbage collection techniques presented below typically has better throughput performance.

2.3 Mark-and-sweep

Unlike manual management and reference counting which never scans through all the heap allocated objects, tracing garbage collection involves finding all live heap objects by following a root set of objects out through all their children and children's children.[10] Once all the live objects have been found, unused memory and resources can be reclaimed. The mark-and-sweep does exactly this in two phases (illustrated in figure 2.2): First all objects reachable from the root set are marked, then the "sweep" phases reclaim unused memory and release unused OS resources. Unlike the "mark" phase, the time spent in the "sweep" phase is proportional to the total size of the heap, not just the number of live objects.[11] This algorithm correctly reclaims cyclic data structures when they're no longer referenced but, similarly to manual management and reference counting, leads to fragmentation and thus suboptimal cache performance. In his 1987 paper "Brief Survey of Garbage Collection Algorithms", David A. Chase states: "In a computer with virtual memory both mark-and-sweep and reference-counting garbage collection tend to exhibit poor locality of reference and may run slowly because of excessive page faulting."[12] As the cost of memory has gone down, swapping memory in and out (ie. page faulting) has become less common but the poor locality of reference hurts any caching scheme, be it between memory and disk, or CPU and memory.[13]

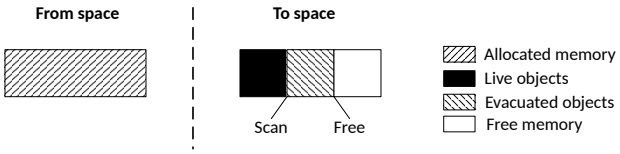


Figure 2.3: Cheney's semi-space algorithm.

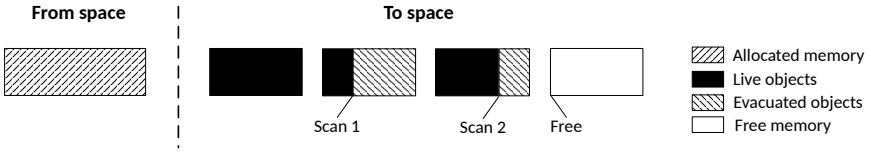


Figure 2.4: Moon's algorithm.

2.4 Copying collectors

In 1969, Fenichel et al published an algorithm for copying the reachable objects into newly allocated memory chunk rather than merely marking them.[14] Once the copying is complete, all previously used memory will no longer contain live objects and can be reclaimed. This makes the "sweep" phases unnecessary and it gets rid of any fragmentation at the cost of requiring 2x available memory. The published algorithm used stack space to traverse and copy the heap objects which is undesirable since the stack space is limited and garbage collection will, therefore, fail if the heap grows too large. One year later, Cheney published a non-recursive algorithm for copying live objects and this algorithm has become the foundation for essentially all modern copying garbage collectors.[15] Since the algorithm is non-recursive, it uses no stack space and works on heaps at any size.

Cheney's and Fenichel's algorithms both solve the same problem of copying a graph with Cheney's algorithm having the decisive advantage of working on graphs of unlimited size. However, the two algorithms also differ in the fact that Fenichel copied the graph in depth-first order and Cheney copies using breadth-first. Figure 2.3 illustrates how objects in the "from-space" are evacuated (ie. copied) into "to-space" and later scavenged (ie. had their children evacuated). This different copy order affects cache performance but the effects weren't too pronounced with the architectures at the time and it wasn't until 1984 that Moon et al investigated the effect copy order has with virtual memory. And it wasn't until 1995 that the effects of copy order were quantified in relation to the CPU memory cache hierarchy by Nakashima and Chikayama.[16]

As illustrated in figure 2.4, Moon modified Cheney's algorithm to have two scanning pointers instead of one.[17] The to-space is divided up into blocks and the second scanning pointer always points at the same block as the free pointer. Since objects are read (scavenged) from the scanning pointer and children are copied (evacuated) to the free pointer, this means the algorithm often only touches a single block even when the graph is large and the free pointer is many blocks away from the first scanning pointer. Since this is optimizing for block/page access, it can perform better when memory is limited and page faults are expensive.

Moon's algorithm was later improved by Wilson, Lam, and Moher to reduce the number of objects that are scavenged twice (Moon's algorithm may scavenge some objects more than once which is safe because scavenging is

idempotent).[18] Siegwart and Hirzel further improved performance by adapting the algorithm for multi-processor machines.[19] Both groups of authors focused primarily on reducing cache and TLB misses.

2.5 Generational collectors

It was found empirically that most allocated objects become unreachable (die) almost immediately and that very few objects live for a significant fraction of a programs total run-time.[20,21] This effect of high infant mortality is exploited by generational collectors by splitting the heap into two or more separate areas (aka generations) that can be collected independently. A common configuration is to have a minor and a major generation: New objects are allocated into the minor generation and promoted into the major generation if they survive a collection. Since it's relatively rare for older objects to contain pointers to younger objects, only a small amount of bookkeeping is necessary to collect the minor generation without tracing through all the live objects in the major generation.[21] In other words, infant objects can be allocated and collected without tracing or copying long-lived objects. This approach has proved so successful that, even though the exact rate of infant mortality and object longevity varies from program to program, generational GC is the default in most popular garbage collected languages.

Furthermore, in "Multicore Garbage Collection with Local Heaps", Marlow et al talks about an "allocation wall" where performance is capped by the rate new objects can be allocated in main memory.[22] Hitting the "allocation wall" means your program will no longer run faster when given access to more cores. Marlow et al solve this problem by having a separate, cache-sized allocation area for each core. Objects allocated in these allocation areas may die and be garbage collected without ever leaving the CPU cache, freeing up main memory bandwidth for long-lived objects.

2.6 Haskell

Haskell, as a purely functional programming language, models the behavior of functions like mathematical functions. That is, they are fixed mappings between domains: Call a function with the same argument and you'll always get the same result. Side-effects and mutable data are still possible through various means but their use is discouraged. Instead of mutating data, persistent data structures are "changed" by creating new copies and letting the garbage collector get rid of the old. Inserting a new node in a tree, for example, would create an entirely new tree. Since none of the data can be mutated, unmodified branches from the old tree can be reused in the new tree. Any fragment of the old tree that was not reused will be found by the garbage collector and deallocated. This style of programming enhances compartmentalization and makes it easier to reason about functions at the cost of a rate of data allocation not seen in other programming paradigms.

Haskell does have implicit mutations caused by laziness, though. Assigning '1+1' to a variable does not have to perform any computations before the value of that variable is used. When the value is demanded, '1+1' is evaluated and the variable is mutated to the value of '2'. These kinds of implicit mutations tend to happen more frequently than explicit mutations.

2.7 LHC

LHC[ii] is a research compiler that compiles a subset of the Haskell 2010 standard. The name of LHC is partly derived from the fact that the compiler generates LLVM[iii] (Low-level Virtual Machine) IR code. It is not unique in this sense but, unlike the widely used Haskell compiler GHC, LHC tries to generate very high-level LLVM IR code and actively tries to avoid duplicating optimizations that have already been implemented in the LLVM compilers.

LLVM has some built-in support for accurate garbage collection but LHC does not use these primitives.[iv] The garbage collection support is still experimental and it forces compilers to make specific choices regarding exceptions and thread handling which do not fit every programming language.

LHC, as a research compiler, offers a mechanism for writing Haskell-specific garbage collectors as plugins. This mechanism takes care of finding the root-set and integrating the collector with the mutator but everything else is left to the plugin itself.

The execution model of LHC is based on eval/apply[23] to implement laziness and higher-order functions. Objects (both data and suspended computations) are identified with a tag number defined at compile-time. Using a tag number rather than a pointer to an object description leaves more room for per-object metadata since tag numbers use fewer bits than pointers. Per-object metadata will play an instrumental role in the implementation of tail-compaction (described in section 3.2).

2.8 Related work

Depth-first copying

In his 1984 paper, Moon wrote: "Depth-first copying generally yields better locality than breadth-first copying, because it tends to put components of a structure on the same page as the parent structure."[17] In order to quantify the locality benefits of depth-first copying, Hiroshi Nakashima and Takashi Chikayama implemented two algorithms for depth-first copying that were not limited by the amount of available stack space.[?] The first algorithm, called the link method, reserves a single word in each allocated object. This space is then used to construct a linked-list of objects to be copied. If, for example,

[ii] https://github.com/Lemmih/lhc/

[iii] LLVM Language Reference Manual: https://llvm.org/docs/LangRef.htm

[iv] Accurate Garbage Collection with LLVM: https://llvm.org/docs/GarbageCollection

the algorithm encountered a branch in a binary tree, it'll push one node to the linked list and immediately copy the other. When the algorithm reaches a terminal node, it pops an object from the linked list and continues from there. Their other algorithm, called the reservation-stack method, is more conventional and reserves a section of the heap for the bookkeeping required for depth-first copying.

The two algorithms were successful at significantly lowering the number of page-faults in their example program but came at the cost of increased memory usage and lower overall performance. Nakashima and Chikayama concluded that their algorithms would be beneficial in situations where page-faults were particularly expensive. I believe that their research indicates that different copy orders can be worth it if their costs are kept minimal.

Custom Object Layout

Another approach altogether is to allow custom data layouts for different data structures. Nowark et al used such a scheme to implement custom data layouts for tree maps, red-black trees, and splay trees, and saw a marked drop in both last-level cache misses as well as in total runtime.[2] This approach lets the authors of data types take advantage of their knowledge to predict how the data will be used and which layout would be most efficient.

Chapter 3

Tail-copying and tail-compaction

3.1 Concept overview

In his foundational "Garbage Collection in a Large LISP System", David A. Moon writes: "A copying garbage collector is free to choose the order in which it copies accessible objects. It can exploit this freedom to improve locality by copying related objects onto the same page."[17] It is this freedom that I want to use, not to copy related objects onto the same page but rather copy related objects onto the same cache-line when possible.

While it has been shown that tracing garbage collectors that use a depth-first copying algorithm have superior data locality, this technique is not used in any of the ten most popular languages on GitHub. The overheads in turns of memory usage and runtime performance are simply too great and the benefits only manifest themselves in niche situations (eg. with limited physical memory and a high cost of page faults). However, I think a compromise is possible that maintains most of the data locality benefits without hampering performance in the common case, making such an algorithm suitable as the default garbage collector rather than just being useful in special cases.

Between depth-first copying and breadth-first copying, there exists a middle-ground which I call tail-first copying. In this scheme, exactly one child of each node (the tail) is copied depth-first and the rest of the children (if there are any) are copied breadth-first. This has several potential benefits.

- Firstly, as one child is likely (but not guaranteed to be) next to its parent, the mutator could experience improved locality if it traverses the spine of a data structure (ie. down the length of a list or down the right-most path of a tree).
- Secondly, and more importantly, the tail-first copying algorithm operates over blocks of tail-linked nodes and is less likely to interleave unrelated nodes. In contrast, depth-first copying of two linked lists results in each node of the lists being interleaved. And if the heap was extended to include a third list, that list would be interleaved with the first two.

This means memory is typically not copied sequentially which leads to more costly cache misses.

- The final benefit is that tail-first copying does not require any reserved book-keeping data in each node nor does it require any extra stack space. The copying algorithm that places children next to their parents is tail-recursive (ie it uses a fixed amount of stack space), and, just like the breadth-first algorithm, a single pointer can be used to keep track of which objects have been scavenged and which have not.

Tail-first copying in inherently optimistic as there are many situations that make it impossible to put a child node right after its parent. Each of these situations has to be detected and they add complexity as well as logic branches to the algorithm. There are three key cases to consider:

1. Multiple parents. Objects may have any number of parents but can only be placed next to one. In this case, the child would be placed next to the first parent and all subsequently traversed parents will be placed potentially far away in memory. I don't believe this to be a big problem because, while having multiple is definitely not uncommon, the majority of objects should only have a single parent.

2. Unlucky root set traversal. If object A is the parent of object B and the root set contains both objects, there's a chance that object B will be copied (evacuated) before object A and it, therefore, cannot be placed immediately after object A. This should be a minor problem as root set traversals happen infrequently and the size of the root sets are fixed at compile-time.

3. Generational barriers. In generational garbage collectors, objects are separated by age into different areas (called generations). Objects in separate generations must live apart and cannot be placed right next to each other. Fortunately, as objects in older generations rarely point to objects in younger generations and due to techniques such as eager promotion (moving an object to an older generation if it has parents in that generation), data structures are rarely interspersed over many generations. Instead, chunks of linked data tend to be located in the same generation, with possible links to chunks of data in other generations. As such, I don't believe generational garbage collecting would significantly affect the behavior of tail-copying. That said, this report will focus exclusively on adding tail-copying to a non-generational garbage collector.

For nodes that cannot be placed immediately next to their parent, tail-first copying is identical to breadth-first copying.

After the live objects in a heap have been traced by the tail-first copying algorithm, many of the objects will have a peculiar property in common: They'll contain a pointer to a child located right after it. That is, these objects of size

S and located at position X, will contain a pointer to a child at location $X + S$. This common, repeated pattern can be compressed to a single bit set in the parent's object header: If the bit is set, the last child pointer is omitted from the object itself and instead calculated as $X + S$.

If nothing else is changed, this compression of the heap would lead to a smaller allocation area (since the allocation area is sized as a multiple of the live heap size) and the garbage collector would, therefore, be invoked more frequently. On the other hand, it is also possible to use the extra heap space to extend the allocation area which would make the garbage collector run less frequently. Both options have their uses, but, for the sake of comparing apples to apples, I've chosen the middle ground of keeping the allocation area size fixed across the three garbage collection strategies. This should make the heaps slightly smaller but keep the number of garbage collections the same.

Switching between breadth-first, depth-first, and tail-first copy order does not change the semantic layout of the heap objects and this can, therefore, be done without changing any other parts of the evaluator or the runtime system. In contrast, omitting the tail pointer *does* change the memory model and all systems that interact with the heap objects will need to take it into account.

3.2 Implementation

Baseline garbage collector

Before tail-first copy order and tail-pointer elimination can be implemented, a semi-space garbage collector needs to be written for LHC. While LHC didn't ship with a semi-space garbage collector when this project began, it does have excellent support for finding the roots of the heap which happens to be one of the most difficult parts of a garbage collector.

The garbage collector that I've implemented in LHC is nearly identical to the algorithm put forth in Cheney's "A Non-recursive List Compacting Algorithm" without any of the improvements that were later invented (such as cache-sized nurseries, stepping, or generations). The algorithm has five steps and is run when new allocations have hit a certain limit:

1. Allocate a *to-space* of the same size as the previous heap. This ensures that there will be enough space to potentially copy the entire heap.

2. Marking: Evacuate (copy object without copying children) everything in the root-set.

3. Scavenging: For each object in *to-space*, evacuate its children. Once all the objects have been scavenged, the entire heap will have been traced and all live objects are copied.

4. Extend the *to-space* with a new allocation area. This allocation area is the size of the *to-space* multiplied by a factor of 3. That is, if *to-space* contains, say, 2 megabytes of objects, the new allocation area will allow

2∗3 = 6 megabytes of new objects to be allocated before another round of garbage collection is triggered. For performance reasons, the allocation area cannot be smaller than 1 megabyte.

5. Free the old heap and declare *to-space* to be the new heap.

LHC's garbage collection API made it simple to implement the above algorithm without having to worry how the rest of the compiler is constructed. Additionally, the garbage collection API is timed with high-precision clocks to give an accurate summary of how much time is spent in the mutator compared to the garbage collector.

Tail-first copy order

The copy order of the garbage collector is entirely determined by the 'evacuate' procedure in the algorithm outlined in 3.2. This produce takes the address of a heap object and performs the following steps:

1. If the object is an indirection, follow the indirection. This step is repeated until an actual heap object is found.
2. Check if the object is already in the *to-space*. If so, nothing more needs to be done and we can exit the procedure.
3. Lookup the size of the object and copy it to the next free spot in *to-space*.
4. Update the given address of the heap object to point to its new position in the *to-space*.
5. Overwrite the old heap object with an indirection to its new position.

In some Haskell compilers, like GHC, indirections take up two words of memory (one for the tag and one for the pointer). This can cause problems when you're trying to overwrite heap objects that are smaller than two words. Luckily, in LHC, indirections only take up a single word and no special precautions are necessary. LHC is able to squeeze an indirection into a single word because all heap objects are word aligned and pointers to them, therefore, don't use the least significant 2 or 3 bits (for 32bit and 64bit systems, respectively).

A key thing to note is that the 'evacuate' procedure only copies a single heap object and not the children of that object. This was the key changed proposed by Cheney to Fenichel's algorithm which recursively copied all children as well.[14,15] Between these two approaches, there's a middle ground: Recursively copy a single child (if it exists). Doing so would be more likely to put related objects next to each other than breadth-first copy, and it still wouldn't require the stack space that a depth-first copy does.

Introducing a check for children to the procedure, and recursing if it is true, does not add a significant amount of complexity. However, it is important to verify that the procedure is indeed tail-recursive. Otherwise, it'll use

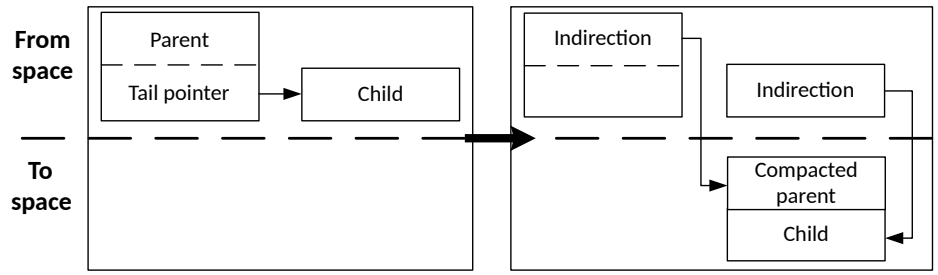


Figure 3.1: Evacuation of both child and parent nodes.

an unbounded amount of stack space and undoubtedly fail for sufficiently large heaps.

Tail-compaction

The elimination of tail pointers is straight forward in the common case but there are two situations that add complexity: Firstly, objects are always mutated by replacing them with an indirection to a new object. In contrast to in-place updates, this approach allows small objects to be replaced by larger objects and the update happens atomically. The indirections are removed during garbage collection and should not inhibit tail-pointer elimination as long as all other requirements are met. Secondly, objects may have multiple parents but only a single pointer to an object can be eliminated. The pointer elimination happens on a first-come, first-served basis and the order may change between garbage collections such that children get placed next to a different parent each time. When this happens, the pointer that was previously eliminated will have to be restored.

The tail-pointer elimination algorithm depends on whether the parent has a tail-pointer or not and whether the child has already been evacuated or not. This leads to four new cases that need to be handled in the 'evacuate' procedure:

1. Parent and child both in *from-space*. Figure 3.1 illustrates how a compacted parent node is created in *to-space* when both the parent and child

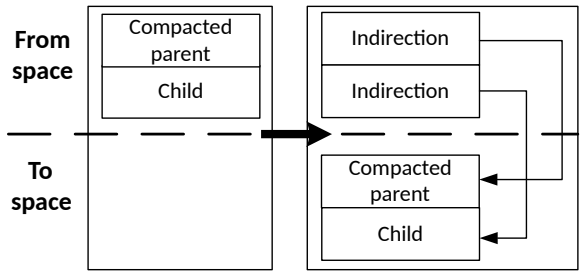


Figure 3.2: Evacuation of compacted parent and child.

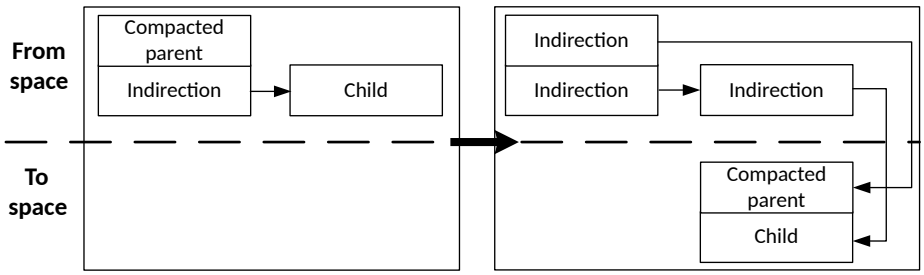


Figure 3.3: Evacuation of compacted parent and modified child.

nodes were in *from-space*. It should be noted that this algorithm is recursive and the child node may itself be compacted if it itself is a parent and its child satisfies the requirements. The figure also shows that both the parent and child nodes in *from-space* are replaced by indirections. This guarantees that nodes are never duplicated (ie. copied twice) during garbage collection, even when the nodes have multiple parents.

2. Compacted parent and child both in *from-space*. Figure 3.2 illustrates how compaction is maintained when the child has been evacuated or moved. If the child is still available, it is immediately moved together with its parent and no intermediate pointers have to be introduced. As the algorithm is recursive and the child may also be compacted, this leads to sequential blocks of memory being copied rather than individual objects that may be scattered in memory.

   Indirections that still point to *from-space* can occur before any child node and that case needs to be handled as well. Figure 3.3 displays a situation where a child node was previously placed next to its parent but then later modified (ie. replaced with an indirection to a new object). Conceptually, indirections are transparent and do not have any effect on semantics but they do add complexity to the implementation. As a side note, a compacted parent with an indirection is isomorphic to an ordinary parent with a tail-pointer.

3. Parent with evacuated child. Figure 3.4 illustrates what happens when a

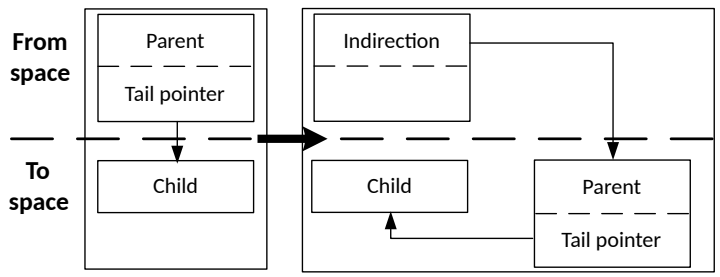


Figure 3.4: Evacuation of parent only.

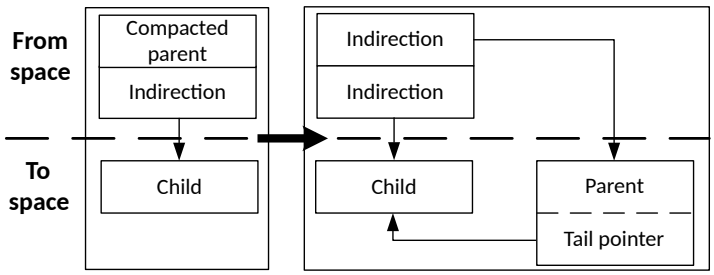


Figure 3.5: Evacuation of compacted parent while introducing new child pointer.

child node has already been evacuated. This can happen when the child has multiple parents or if the child was part of the initial root set. In this case, the parent cannot be placed right above the child and its therefore copied verbatim instead, keeping the tail-pointer intact.

4. Compacted parent with evacuated child. Figure 3.5 illustrates a situation where the child node most likely has multiple parents and is now placed next to a different parent. Here a new tail-pointer has to be introduced between the parent and the child.

Since the new 'evacuate' procedure guarantees that tail pointers always point to *to-space*, the scavenging procedure can be optimized to omit the call to 'evacuate' on the tail pointer.

To complete the implementation of tail-pointer elimination, the rest of the runtime-system has to be aware of how to find the address of a child by either following a pointer or computing it from the address of the parent. The LHC compiler tries to be agnostic towards the layout of objects and provides an API access point for finding the address of an object's last child. The full code is tiny (see appendix A) and it simply either follows the pointer to the child of it exists or directly returns the address of the child if the pointer has been eliminated.

Chapter 4

Evaluation

4.1 Methodology

The effects of tail-copying and tail-compaction are measured over 7 small benchmark programs with metrics collected in three different ways. GC statistics (heap size, allocations, bytes copied) are reported by the LHC RTS, and, since these metrics do not change between runs, the benchmark suite is only executed once to collect this data. How the runtime is split between the mutator and the garbage collector is measured using a high-precision clock with a resolution of 1ns. These runtime measurements are inherently stochastic and the benchmark suite is executed fives and only the median values are used. Hardware performance counter statistics are collected using the Linux *perf*[i] tool. This tool is used to collect the number of page faults, cycles, instructions, branches, branch misses, LLC (last-level cache) loads and LLC load misses. Again, these measurements are stochastic and the benchmark suite is executed five times and the median values are used.

The *perf* tool supports other counters but this subset was chosen since each of these counters should something relevant to the performance of the two novel algorithms. Page faults (in this benchmark suite) are triggered when newly allocated memory is used for the first time. They find an unused page of physical memory and make sure its filled with zeros. This can be fairly time consuming and much research has gone into reducing the number of page faults.[12,13,16,17] The number of cycles is related to the number of instructions but seemingly innocent algorithmic changes can have unintuitive effects. With the right scheduling, multiple instructions can be executed in parallel and latency from the memory systems can be hidden (eg. by prefetching data). Getting this scheduling wrong will make the cycle count go up even when the number of executed instructions stays the same. Branches and particularly branch-misses are another factor in how efficiently a CPU can execute instructions. A branch mispredict flushes the instruction pipeline, which

[i] perf: Linux profiling with performance counters. `https://perf.wiki.kernel.org/index.php/Main_Page`

Table 4.1: "BinTree" pointer distances

Object distances:		
Bucket	**Baseline**	**Tail-copy**
<64 b	523	114053463
<4 kb	26349	6616
<64 kb	423445	211347
<512 kb	2810463	1564876
<2 mb	7874711	4414888
>2 mb	216971852	107856153

Table 4.2: "Allocate" pointer distances

Object distances:		
Bucket	**Baseline**	**Tail-copy**
<64 b	79930413	79930435
<4 kb	3378	3360
<64 kb	51200	51200
<512 kb	361736	361736
<2 mb	1029416	1029408
>2 mb	78484699	78484703

are quite deep in modern architectures, wasting 10-20 cycles.[9]

All measurements were gathered on an Intel i7-8705 3.1GHz running Linux 4.18 64bit.

Finally, for the reasons stated in section 3.1, tail-compaction depends on tail-copying and automatically enables it.

4.2 Verification

The seven benchmark program used all have heap sizes measured in gigabytes, making it nigh impossible to visually inspect for verification. Fortunately, there's a lot of aggregate data which can give us an indication of whether tail-copying and tail-compaction work as theorized in section 3.1.

Table 4.1 and 4.2 show the pointer distances between parent and child objects, grouped into 6 buckets. Siegwart et al uses this metric of pointer distances to quantify the improvements of their garbage collection algorithm.[19] Since the purpose of the tail-copying algorithm is to put children right next to their parent, the number of pointers that refer to objects within 64 bytes should increase dramatically and this is exactly what is seen for the *BinTree* benchmark. The *Allocate* benchmark, however, does not exhibit this pattern.

Table 4.3: Heap size

Heap size:			
Name	**Baseline**	**Tail-copy**	**Tail-compact**
Allocate	1.8 gb	+0.0%	-8.3%
BinTree	1.8 gb	+0.0%	-6.2%
Bush	1.3 gb	+0.0%	-3.1%
BushTail	5.1 gb	+0.0%	-1.7%
Imbalanced	1.6 gb	+0.0%	-6.2%
MemBench	6.7 gb	+0.0%	-5.0%
PowerSet	10.4 gb	+0.0%	-5.0%
Min		0.0%	0.0%
Max		0.0%	0.0%
Mean		0.0%	0.0%

Table 4.4: Allocations

Allocated:			
Name	**Baseline**	**Tail-copy**	**Tail-compact**
Allocate	4.0 gb	+0.0%	+0.0%
BinTree	9.8 gb	+0.0%	+0.0%
Bush	6.6 gb	+0.0%	+0.0%
BushTail	5.2 gb	+0.0%	+0.0%
Imbalanced	7.8 gb	+0.0%	+0.0%
MemBench	9.7 gb	+0.0%	+0.0%
PowerSet	15.1 gb	+0.0%	+0.0%
Min		0.0%	0.0%
Max		0.0%	0.0%
Mean		0.0%	0.0%

Table 4.5: Mutator time

Mutator times:			
Name	**Baseline**	**Tail-copy**	**Tail-compact**
Allocate	1.6s	+0.1%	-0.2%
BinTree	3.4s	-0.4%	-0.5%
Bush	2.4s	-0.2%	-0.2%
BushTail	3.6s	-0.3%	-0.5%
Imbalanced	2.7s	-0.0%	-0.3%
MemBench	3.4s	+0.0%	-0.2%
PowerSet	6.9s	+0.0%	+0.1%
Min		-0.4%	-0.5%
Max		0.1%	0.1%
Mean		-0.1%	-0.3%

Table 4.6: GC time

GC times:			
Name	**Baseline**	**Tail-copy**	**Tail-compact**
Allocate	1.5s	+14.5%	+3.1%
BinTree	4.6s	+16.2%	+10.2%
Bush	2.8s	+12.5%	+9.6%
BushTail	3.6s	+10.7%	+5.9%
Imbalanced	4.0s	-0.7%	-6.2%
MemBench	7.8s	-27.5%	-31.4%
PowerSet	12.6s	-31.1%	-34.5%
Min		-31.1%	-34.5%
Max		16.2%	10.2%
Mean		-0.8%	-6.2%

This is due to the way the program manipulates a single list: For lists, there's no significant difference between breadth-first copying, depth-first copying or tail-first copying. The *Allocate* benchmark is an outlier, though, and none of the other programs behave like this (as shown by the pointer distances in appendix B).

Furthermore, as shown by table 4.4, neither of the two new strategies have an effect on the number of allocations. This is exactly as expected since both copying and compaction happens well *after* objects are allocated. Looking at table 4.7, it can be seen that the number of bytes copied during garbage collection when tail-compaction is enabled, always drops by a fraction: -1/3 or -33.3%, -1/4 or -25%, -1/5 or -20%, -1/8 or -12.5%. This is because each benchmark problem mainly uses a single data structure of fixed size. If the size of this data structure is, say, 4 words, eliminating the one-word pointer would yield a saving of 1/4. Finally, table 4.3 shows a modest reduction in heap size when tail-compaction is enabled, just as predicted earlier. These results indicate that the garbage collection strategies work as theorized.

4.3 Performance interpretation

Tail-copying does not change the code of the mutator but changing the layout of the data may very well still affect the performance. Table 4.5 shows that both tail-copying and tail-compaction tends to improve the performance of the mutator ever so slightly. This effect is so tiny as to be insignificant, though, and the key insight to glean is that neither strategy hurts the performance of the mutator.

GC times, as seen in table 4.6, reveals a more complicated landscape: for both tail-copying and tail-compaction, three benchmarks become faster and four benchmarks become slower. It is important to note, however, that all seven benchmarks are improved by tail-compaction compared to just tail-copying. Understanding the performance of tail-copying and tail-compaction requires a deeper dig into the performance counters.

Table 4.7: Copied by GC

Copied:			
Name	**Baseline**	**Tail-copy**	**Tail-compact**
Allocate	1.8 gb	+0.0%	-33.3%
BinTree	3.8 gb	+0.0%	-25.0%
Bush	2.5 gb	+0.0%	-12.5%
BushTail	3.1 gb	+0.0%	-19.9%
Imbalanced	3.0 gb	+0.0%	-25.0%
MemBench	4.9 gb	+0.0%	-20.0%
PowerSet	7.6 gb	+0.0%	-20.0%
Min		0.0%	-33.3%
Max		0.0%	-12.5%
Mean		0.0%	-22.2%

Table 4.8: Page faults

Page faults (millions):			
Name	**Baseline**	**Tail-copy**	**Tail-compact**
Allocate	1.8	-0.0%	-8.7%
BinTree	3.6	+0.0%	-6.8%
Bush	2.6	+0.0%	-3.2%
BushTail	3.3	+0.0%	-5.0%
Imbalanced	2.9	+0.0%	-7.0%
MemBench	4.1	-0.0%	-6.3%
PowerSet	7.7	+0.0%	-5.2%
Min		-0.0%	-8.7%
Max		0.0%	-3.2%
Mean		-0.0%	-6.0%

Table 4.9: Cycles

Cycles (billions):			
Name	**Baseline**	**Tail-copy**	**Tail-compact**
Allocate	12.7	+7.1%	+1.1%
BinTree	32.6	+9.1%	+6.1%
Bush	21.2	+6.1%	+5.0%
BushTail	29.3	+5.3%	+3.0%
Imbalanced	27.3	-0.5%	-3.8%
MemBench	45.9	-18.9%	-21.5%
PowerSet	79.7	-19.6%	-22.0%
Min		-19.6%	-22.0%
Max		9.1%	6.1%
Mean		-1.6%	-4.6%

Table 4.10: Instructions

Instructions (billions):			
Name	**Baseline**	**Tail-copy**	**Tail-compact**
Allocate	23.6	-2.4%	-1.0%
BinTree	49.8	-1.1%	+0.4%
Bush	33.0	-0.1%	+0.7%
BushTail	40.4	-2.3%	-1.4%
Imbalanced	40.0	-0.5%	+0.6%
MemBench	54.1	-1.0%	+0.2%
PowerSet	94.6	-0.9%	+0.0%
Min		-2.4%	-1.4%
Max		-0.1%	0.7%
Mean		-1.2%	-0.1%

Table 4.11: Branches

Branches (billions):			
Name	**Baseline**	**Tail-copy**	**Tail-compact**
Allocate	4.4	-2.0%	+0.9%
BinTree	9.1	+0.1%	+2.7%
Bush	6.0	+1.4%	+2.4%
BushTail	7.2	-1.0%	+1.4%
Imbalanced	7.3	+0.3%	+2.9%
MemBench	10.0	-0.2%	+2.3%
PowerSet	17.1	-0.0%	+2.1%
Min		-2.0%	0.9%
Max		1.4%	2.9%
Mean		-0.2%	2.1%

Table 4.12: Branch misses

Branch Misses (millions):			
Name	**Baseline**	**Tail-copy**	**Tail-compact**
Allocate	3.7	-6.1%	-13.8%
BinTree	29.0	+107.5%	+102.5%
Bush	17.4	+78.3%	+77.0%
BushTail	16.2	+44.4%	+44.2%
Imbalanced	39.4	+6.7%	+4.3%
MemBench	10.7	-7.9%	-12.8%
PowerSet	20.0	-6.7%	-10.7%
Min		-7.9%	-13.8%
Max		107.5%	102.5%
Mean		30.9%	27.3%

Table 4.13: LLC loads

LLC Loads (millions):			
Name	**Baseline**	**Tail-copy**	**Tail-compact**
Allocate	9.0	-16.7%	-22.9%
BinTree	71.1	-2.8%	-5.6%
Bush	38.3	+7.3%	+3.3%
BushTail	61.4	+1.0%	+0.1%
Imbalanced	50.5	-9.6%	-14.5%
MemBench	180.1	-88.0%	-88.2%
PowerSet	299.6	-88.2%	-88.4%
Min		-88.2%	-88.4%
Max		7.3%	3.3%
Mean		-28.1%	-30.9%

Table 4.14: LLC load misses

LLC Load Misses (millions):			
Name	**Baseline**	**Tail-copy**	**Tail-compact**
Allocate	3.9	+5.5%	+6.2%
BinTree	60.9	-4.5%	-4.9%
Bush	33.5	+4.1%	+2.8%
BushTail	50.9	+0.8%	+0.6%
Imbalanced	40.9	-10.3%	-11.0%
MemBench	84.1	-80.3%	-80.0%
PowerSet	130.3	-78.7%	-78.7%
Min		-80.3%	-80.0%
Max		5.5%	6.2%
Mean		-23.3%	-23.6%

Table 4.8 shows that enable tail-compaction reduces page faults across the board and the drop corresponds almost exactly to the reduced heap size. This, together with LLC loads which are also reduced across the board, appears to be why tail-compaction improves the performance of every single benchmark program (relative to tail-copying) even though the total number of branches go up (table 4.11).

The reason *Allocate*, *BinTree*, *Bush* and *BushTail* perform so poorly appears to be due to CPU stalls: Number of cycles go up while number of instructions go down (as shown in tables 4.9 and 4.10). In the cases of *BinTree*, *Bush* and *BushTail*, this appears to be primarily caused by a large increase in branch misses, ranging from +44.4% to +107.5% (see table 4.12), and partly due to a small increase in LLC load misses in the case of *Bush* (see table 4.14). *Allocate* follows a different pattern with a reduction in both branch misses and LLC loads but a sizable increase in LLC load misses, leading to a poor instructions-per-cycle count (see table 4.13).

Only four of the benchmark programs (*BinTree*, *Imbalanced*, *MemBench* and *PowerSet*) exhibit the reduction in LLC load misses that tail-copying and tail-compaction aims for. Furthermore, the benefits of the reduction in LLC load misses is impeded by the extra cost of branch misses in the case of *Imbalanced*. For *BinTree* the situation is even worse and the benefits are dwarfed by the extra costs. All in all, the two new garbage collection strategies succeed in arranging the heap in a more cache-friendly manner but the gains are eaten by unpredictable branches.

4.4 Possible improvements

Using the heap size saving to increase the allocation area would be the easiest way of improving the performance of the tail-compaction strategy. A larger allocation area means fewer objects survive between each collection and the garbage collector, therefore, has less work to do and fewer objects to copy. Given that the heap savings can be quite significant (8.3% in the case of *Al-*

locate), this might very well push the tail-compaction strategy from being a slight performance loss to being a slight performance gain.

It may also be possible to reduce the number of branch misses. The *perf* tool identifies the 'evacuate' procedure as the culprit due to the radically different behavior that depends on whether a node can be compacted or not. Predicting this is difficult for the CPU but there *are* patterns that could be exploited. For example, compacted nodes usually appear in chains so if the algorithms find one compacted node then the next one is likely to also be compacted. Furthermore, only tail pointers are candidates for compaction. These two facts could be exploited by having two separate 'evacuate' procedures, one which assumes a node will be compacted and one which assumes it will not.

A third possible improvement would be to identify which data structures benefit the most from tail-copying and tail-compaction. It tentatively appears that lists and binary trees benefit more than bushy data types but I have no empirical evidence for this.

4.5 Code complexity

The implementation of tail-copying takes up only 15 lines out of the 224 lines that define the full semi-space garbage collector in LHC. The implementation of tail-compaction only takes up 10 lines in the RTS but leans heavily on the garbage collection API in LHC which clocks in at 230 lines of Haskell code. All in all, both the tail-copying and tail-compaction algorithms fit in well in the semi-space garbage collector and do not impose excessive complexity.

Chapter 5

Conclusion

The aim of the thesis was to shed light on whether tail-copying and tail-compaction could be added to a semi-space garbage collector with a reasonable amount of effort, whether doing so would lead to any performance gains, and whether the performance gains (if any) could reasonably be expected to translate over to a modern semi-space garbage collection outside of an academic setting. While the garbage collector implemented as a baseline in this report is as simple as can be and does not support features such as generations and in-place mutations (needed for mutable arrays, hash tables, etc.) which are an absolute must in production systems, such features are quite orthogonal to tail-copying/tail-compaction and should not complicate their implementation by much. In the LHC RTS, the new strategies take up about 10% of the garbage collector and did not require any large structural changes to the baseline collector, making the effort to implement quite modest.

Although tail-compaction did manage to reduce the number of LLC loads in nearly every benchmark program, this rarely translated into a tangible performance increase for a variety of reasons. And while increasing the heap size to match that of the baseline would definitely help performance, the numbers still represent a best-case scenario which is unlikely to be seen in non-academic settings. Commercial applications see much greater use of sharing, generational garbage collectors, mutable data structures, and non-uniform data structures, all of which make fewer nodes compactable and thus lowers the possible benefits of tail-compaction. All-in-all, it seems unlikely that tail-compaction will be useful in a commercial garbage collector until the branch prediction issue has been solved.

Bibliography

[1] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Oil and water? high performance garbage collection in java with mmtk. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 137–146, Washington, DC, USA, 2004. IEEE Computer Society.

[2] Gene Novark, Trevor Strohman, and Emery D. Berger. Custom object layout for garbage-collected languages, 2006.

[3] Stephen M. Blackburn and Kathryn S. McKinley. Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance. *SIGPLAN Not.*, 43(6):22–32, June 2008.

[4] Matthew Hertz. Quantifying the performance of garbage collection vs. explicit memory management. In *in: Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA*, pages 313–326. ACM Press, 2005.

[5] Narendran Sachindran, J. Eliot B. Moss, and Emery D. Berger. Mc2: High-performance garbage collection for memory-constrained environments. In *IN PROCEEDINGS OF THE ACM CONFERENCE ON OBJECT-ORIENTED SYSTEMS, LANGUAGES AND APPLICATIONS*, 2004.

[6] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. *SIGPLAN Not.*, 34(5):1–12, May 1999.

[7] Mark B. Reinhold. Cache performance of garbage-collected programs. *SIGPLAN Not.*, 29(6):206–217, June 1994.

[8] N. Eiron, M. Rodeh, and I. Steinwarts. Matrix multiplication: A case study of enhanced data cache utilization. *J. Exp. Algorithmics*, 4, December 1999.

[9] Ulrich Drepper. What every programmer should know about memory, 2007.

[10] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Inf. Process. Lett.*, 25(4):275–279, June 1987.

[11] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, IWMM '92, pages 1–42, London, UK, UK, 1992. Springer-Verlag.

[12] David R. Chase. Brief survey of garbage collection algorithms, 1987.

[13] Dirk Grunwald, Benjamin Zorn, and Robert Henderson. Improving the cache locality of memory allocation. *SIGPLAN Not.*, 28(6):177–186, June 1993.

[14] Robert R. Fenichel and Jerome C. Yochelson. A lisp garbage-collector for virtual-memory computer systems. *Commun. ACM*, 12(11):611–612, November 1969.

[15] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, November 1970.

[16] Hiroshi Nakashima and Takashi Chikayama. Depth-first copying garbage collection without extra stack space, 1995.

[17] David A. Moon. Garbage collection in a large lisp system. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 235–246, New York, NY, USA, 1984. ACM.

[18] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Caching considerations for generational garbage collection. *SIGPLAN Lisp Pointers*, V(1):32–42, January 1992.

[19] David Siegwart and Martin Hirzel. Improving locality with parallel hierarchical copying gc. In *Proceedings of the 5th International Symposium on Memory Management*, ISMM '06, pages 52–63, New York, NY, USA, 2006. ACM.

[20] Robert Courts. Improving locality of reference in a garbage-collecting memory management system. *Commun. ACM*, 31(9):1128–1138, September 1988.

[21] A. W. Appel. Simple generational garbage collection and fast allocation. *Softw. Pract. Exper.*, 19(2):171–183, February 1989.

[22] Simon Marlow and Simon Peyton Jones. Multicore garbage collection with local heaps. *SIGPLAN Not.*, 46(11):21–32, June 2011.

[23] Simon Peyton Jones. How to make a fast curry: push/enter vs eval/apply. pages 4–15, September 2004.

Appendix A

loadLast routine.

```
word* _lhc_loadLast(word *ptr, word idx) {
  if(_lhc_getTail(*ptr)) {
    return ptr+idx;
  } else {
    return ((word**)ptr)[idx];
  }
}
```

Appendix B

Pointer distances

Table B.1: "Bush" pointer distances

Object distances:		
Bucket	**Baseline**	**Tail-copy**
<64 b	122	41819119
<4 kb	13250	4232
<64 kb	163606	110942
<512 kb	1221204	1067398
<2 mb	3568091	3461401
>2 mb	162307510	120810691

Table B.2: "BushTail" pointer distances

Object distances:		
Bucket	**Baseline**	**Tail-copy**
<64 b	63486839	83491247
<4 kb	4266	865
<64 kb	53303	38177
<512 kb	418026	402746
<2 mb	1231390	1428856
>2 mb	78308537	58140470

Table B.3: "Imbalanced" pointer distances

Object distances:		
Bucket	**Baseline**	**Tail-copy**
<64 b	440	88991152
<4 kb	31107	2089
<64 kb	507161	72776
<512 kb	3428454	721718
<2 mb	8952659	2434901
>2 mb	165061074	85758259

Table B.4: "MemBench" pointer distances

Object distances:		
Bucket	**Baseline**	**Tail-copy**
<64 b	29959	132133061
<4 kb	2860192	71
<64 kb	56830312	13479
<512 kb	204545590	83117
<2 mb	0	177361
>2 mb	0	131858964

Table B.5: "PowerSet" pointer distances

Object distances:		
Bucket	**Baseline**	**Tail-copy**
<64 b	500068	204577727
<4 kb	4133952	0
<64 kb	99486789	0
<512 kb	305010962	47458
<2 mb	8	190440
>2 mb	23602	204339756