An Empirical and Analytic Study of Stack vs. Heap Cost for Languages with Closures

Andrew W. Appel
appel@princeton.edu

Zhong Shao
zsh@cs.princeton.edu

CS-TR-450-94
Princeton University
March 8, 1994

Abstract

It has been proposed that allocating procedure activation records on a garbage collected heap is more efficient than stack allocation. However, previous comparisons of heap vs. stack allocation have been over-simplistic, neglecting (for example) frame pointers, or the better locality of reference of stacks.

We present a comprehensive analysis of all the components of creation, access, and disposal of heap-allocated and stack-allocated activation records. Among our results are:

- Although stack frames are known to have a better cache read-miss rate than heap frames, our simple analytical model (backed up by simulation results) shows that the difference is too trivial to matter.
- The cache write-miss rate of heap frames is very high; we show that a variety of miss-handling strategies (exemplified by specific modern machines) can give good performance, but not all can.
- The write-miss policy of the primary cache is much more important than the write-miss policy of the secondary cache.
- Stacks restrict the flexibility of closure representations (for higher-order functions) in important (and costly) ways.
- The extra load placed on the garbage collector by heap-allocated frames is very small.
- The demands of modern programming languages make stacks quite complicated to implement efficiently and correctly.

Overall, the execution cost of stack-allocated and heap-allocated frames is very similar; but heap frames are simpler to implement and allow very efficient first-class continuations (call/cc).

1 Garbage-collected frames

In a programming language implementation that uses garbage collection, all procedure activation records (frames) can be allocated on the heap. This is quite convenient for higher-order languages (Scheme, ML, etc.) whose "closures" can have indefinite extent, and it is even more convenient for languages with first-class continuations.

One might think that it would be expensive to allocate, at every procedure call, heap storage that becomes garbage on return. But not necessarily [2]: modern generational garbage-collection algorithms[31] can reclaim dead frames extremely efficiently, even cheaper than the one-instruction cost to pop the stack.

But there are many other costs involved in creating, accessing, and destroying activation records—whether on a heap or a stack. These costs are summarized in Figure 1, and explained and analyzed in the remainder of the paper.

These numbers depend on many assumptions. The most critical assumptions are these:

- The runtime system in question has static scope, higher order functions, and garbage collection. The only question being investigated is whether there is an activation-record stack *in addition* to the garbage collection of other objects.
- The compiler and garbage collector are required to be "safe for space complexity;" that is, statically dead pointers (in the dataflow sense) do not keep objects live.
- There are few side effects in compiled programs, so that generational garbage collection will be efficient.

These assumptions, and many others, will be explained in the rest of the paper.

Component	Heap	Stack	see:
Creation	3.1	1.0	§2
Frame pointers	2.0	0.0	§3
Copying, sharing	0.0	3.4	§4
Cache write misses	0.0 or 10.0	0.0	§5.1
Cache read misses	0.1	0.0	§5.3
Disposal (pop)	0.2	1.0	§6
Total Cost	5.4 or 15.4	5.4	
Call/cc	$O(1)$	$O(N)$	§8

The cost for each component of frame creation, access, and disposal for heap-allocated and stack-allocated frames is shown, measured in *instructions per frame* (tail recursions and leaf procedures do not make frames). The numbers for *cache write misses* depend critically on the design of the machine's *primary cache*; we show the cost of two alternatives. The last column has references to the section number (in this paper) of the explanation of each component.

Figure 1: Cost breakdown

The most important factor (other than the efficiency of call/cc) in choosing a stack or heap representation is whether the target machine's *primary cache* has a large write-miss penalty. This is shown in the *Cache write misses* line of the table, and discussed in Section 5.1 of the paper. On machines with a write-miss penalty, or on which write misses bypass the cache, heap-allocated frames are significantly more expensive.

The (perhaps) startling result is that heap-allocated frames have the same cost than stack frames.[1] But since our numbers are not really so accurate as that, we prefer to state our conclusions as:

- Given our assumptions, pure heap allocation gives approximately as good results as stack allocation.
- Given our assumptions, stacks are more complicated to implement (as we will explain).

Finally, we point out that the absolute differences are small: one instruction per frame is about 1% of total execution cost, as can be calculated from Figure 4.

2 Creation

To allocate a stack frame, the program must add a constant to the stack pointer. This takes one instruction.

[1] We assure the reader that we have not fudged the data to make them come out equal; it is purely coincidental.

It's also necessary to check for stack overflow, but since overflow is so rare this can usually be done at no cost using an inaccessible virtual memory page.

Program	**Lines**	**Description**
Boyer	919	Theorem-prover benchmark.
Knuth-B	655	Knuth-Bendix completion.
Lexgen	1185	Lexical-analyzer generator.
Life	148	Game of Life, using lists.
YACC	7432	LALR parser generator.
Simple	990	Spherical fluid-dynamics.
VLIW	3658	VLIW instruction scheduler.

Figure 2: General Information about the Benchmark Programs

Program	Limit Checks per Frame
Boyer	.717
Knuth-B	.783
Lexgen	.864
Life	.456
YACC	.631
Simple	.665
VLIW	.695
Average	.687

Many frame allocations are in the same (extended) basic block as other allocations. In these cases the heap limit check would have to be done anyway, and should not be charged to the frame allocation. This table shows the proportion of frame allocations that are *not* in the same block as another allocation. The results shown are from measurements of ML benchmark programs (see Figure 2) as compiled by the *Standard ML of New Jersey*[7] compiler.

Figure 3: Shared limit checks

Allocating a heap frame is more complicated:

1. Heap overflow must be checked. As explained by Appel and Li [6], and contrary to the silly ideas of Appel [3], this should not be done by a virtual memory fault. Thus, a comparison and conditional branch is required; by keeping the free-space pointer and the limit pointer in registers, this takes about two instructions. However, many of the frame allocations occur in the same extended basic block as other (non-frame) allocations, which would require limit checks anyhow (see Figure 3). The actual cost is therefore $2 \cdot 0.687 = 1.374$.
2. The free-space pointer must be incremented. This costs one instruction. But when the frame allocation is in the same basic block as another allocation, the increment can be shared. So

the cost is 0.687 instructions per frame, on the average.

3. A descriptor word must be written to the frame, so the garbage collector can understand it. However, the frame usually contains a return address; the garbage collector can have a mapping of return addresses to descriptors, so frame need not explicitly contain the descriptor.[2]

4. The free-space pointer must be copied to the frame pointer; this takes one instruction.

The total cost is about 3.1 instructions, on the average.

3 Frame pointers

When a stack frame is popped, the frame pointer must be set back to the caller's frame. Some implementations of stack frames have put a copy of the (previous) frame pointer in each frame, and this is fetched back upon function return. But for contiguous stack frames of known size, this is clearly unnecessary; the stack pointer itself can be used as the frame pointer, and the pop can just be a subtraction from the stack pointer. This is the common modern practice.

But when frames are not contiguous (e.g., when they are heap-allocated), then each frame must contain a pointer to the caller's frame. One instruction will be necessary to store the (previous) frame pointer into a new frame; and one instruction will be necessary to fetch it back.

Thus, heap-allocated frames have a 2-instruction cost, per frame, for frame pointer manipulation; stack-allocated frames incur no such cost.

4 Copying, sharing

A language (such as Scheme, ML, Smalltalk) with higher-order functions needs *closures* to hold the free variables of functions that have been created but not yet called. If one function's free variables overlap with another's, then one closure might point to another (which saves the expense of copying the contents).

So there are two kinds of objects: activation records, whose lifetimes have last-in first-out behavior; and higher-order function closures, which have indefinite extent. The former can be stack allocated (or heap allocated), but the latter must be allocated on a garbage-collected heap. Furthermore, *stack frames may point at heap closures, but heap closures may never point at stack frames*, otherwise there will be dangling pointers.

This means that if the compiler wants to build a closure containing free variables (x, y, z) which are available in a stack frame, all three variables must be copied into the closure; the closure cannot just point to the stack frame.

But if all activation records are heap-allocated, then closures may point at them. This flexibility allows the closure analysis phase of a good compiler to choose much better (smaller, shallower) representations for closures, with more sharing and less copying [28].

The restriction that heaps cannot point to stacks must be counted as a "cost" of using stack-allocated frames. To quantify this cost, we measured two versions of the *Standard ML of New Jersey* compiler [7, 4] outfitted with our recently improved closure-representation analysis phase [28].

Program	Ordinary Heap $i/10^3$	"Stack-like" Heap $i/10^3$	Stack Frames $f/10^3$	Extra Instrs per Frame
Boyer	61966	65770	662	5.75
Knuth-B	212702	222376	3465	2.79
Lexgen	310522	316813	1873	3.36
Life	48016	48437	201	2.09
YACC	114687	119065	1187	3.69
Simple	469543	492126	5516	4.09
VLIW	285370	292474	3274	2.17
Average				3.42

The "Stacklike Heap" allocates all frames on the heap, but is careful to divide into two kinds: "stack" frames, which can point only to other "stack" frames; and "heap" frames, which can point to either kind. This lack of flexibility has a significant cost, as shown in the table. The first two columns show thousands of instructions executed; the thirds column shows thousands of frames created.

Figure 4: Copying/sharing cost

The version shown as *Ordinary Heap* in Figure 4, allocates all frames and closures on the heap. The *"Stacklike Heap"* obeys the restriction that closures cannot point to frames (though frames can point to closures). "Frames" are those objects with LIFO lifetimes. But "Stacklike heap" proceeds to allocate frames and closures on the heap; it does *not* use a stack, and does not gain any advantages of using a stack.

The difference in execution time between the two versions is attributable *only* to the slightly more cumbersome representations that are imposed by

[2] Actually, SML/NJ does write an explicit descriptor to each frame, for simplicity.

the "closures cannot point to frames" restriction. The frames themselves are not much bigger, but the closures are: since they can't point to the frames, data from frames must be copied into the closures.

Some programs suffer more from this than others, but on the average the difference is quite significant: about 3.4 extra instructions are executed per every frame creation because of this restriction.

4.1 Space safety

In any language, it is common for the programmer to have variables in scope that are "dead;" that is, their current values will never again be needed. In a garbage-collected language, the garbage collector need not use such variables as "roots" of live data. Several implementors have independently discovered that this is really quite important: if the collector traverses too many dead variables, the memory use of the program can increase by a large factor[11, 12, 27, 4, 21].

In fact, a collector that starts from only the (statically determinable) live variables can often keep *asymptotically less* data live than a less-careful collector; that is, one system might use $O(N)$ space where another uses $O(N^2)$ space. This theorem, examples, and a description of compiler techniques that are "safe for space complexity" are described by Appel [4]. A brief, illustrative example is this:

```
function f(x) =
        let function g(y) =
                    let function h(z) = z+y
                    in print(x); return h
        in return g(0)
```

Function f calls g, which returns the closure h (but does not call it at this time). The closure for h contains a pointer to the machine code for h, along with values of free variables (in this case y).

But a common implementation of closures, using *static links* that point to activation records of outer functions, will keep x accessible in the closure for h. This can be very bad; for if x is the root of a large data structure, and there are many instances of h (each with a different, unnecessary x), the memory usage of the program will be much bigger than optimal.

Assumption: The results of Figure 4 are based on the assumption that the compiler must be "safe for space complexity," which does put some restrictions on both the heap-allocated and stack-allocated frames.

Complicated descriptors

It is possible to allow dead variables in frames and closures, *if the garbage collector knows they are dead.* This can be accomplished using special descriptors, which would reduce the "copying, sharing" penalty for stack frames.

For example, in the Chalmers Lazy ML compiler [10] or the Gallium compiler [24] for CAML, associated with each return address is a descriptor telling which variables in the caller's frame are live *after the return.* [3] But this is not sufficient; heap closures still cannot point to stack frames. A fully flexible system must be able to let the stack frame point to a heap closure that contains several variables, some of which may die before the frame itself. The return-address descriptor would need to indicate not only which variables *in the frame* are dead, but which live variables point to records in which some of the fields are dead. This is quite complicated to implement, and we do not know of anyone who has done it.

5 Locality of reference

Stacks have excellent locality of reference: they are (almost) always moving up and down in a very small region of memory, so access to the stack should (almost) always hit the cache, no matter how small that cache is.

But heap-allocated frames are scattered throughout memory, so creating and accessing them should cause more cache misses.

Since some machines these days have primary caches as small as 8k bytes, and secondary caches with miss penalties as long as 100 cycles, this is a serious concern.

The analysis of cache behavior of garbage collected systems differs qualitatively depending on the size of the cache.

Large Caches For large (e.g., secondary) caches, a *generational* garbage collection algorithm[31] can keep its youngest generation entirely within the cache[34, 35]. Only the (rare) objects that survive a collection (or two) will be promoted into an older generation where they can cause cache misses. The collector itself helps to *improve* the locality of reference of the mutator. Thus, locality of reference in a large cache is basically a solved problem.

Furthermore, activation records tend to die especially young. It will be extremely rare for an activation record to be promoted to a higher generation [30]. Since only the higher generations can cause cache misses[4], heap-allocated frames will (almost)

[3] The bibliographic citations are merely *pro forma*; the author of neither paper has actually described this technique in print.

[4] This is a slight oversimplification.

never cause cache misses. Thus, while there may be secondary cache misses in a garbage-collected system, these will be on the non-frame objects (closures, records, etc.); the difference between stack-allocated and heap-allocated frames will be insignificant.

Small Caches For small (e.g., primary) caches whose size is less than 100 kbytes, it is impractical to keep the youngest generation in the cache; doing so would cause garbage collections to be too frequent, and this would be expensive.

Let us consider locality in a small, primary cache. We assume that any cache of only 8 kbytes will have only a 10-cycle miss penalty—because there are many programs that cannot achieve a better than 90% hit rate in such a small cache, and machine designers will be forced to make a small miss penalty.

The essence of the locality argument against heap allocation is that stacks can exploit a small primary cache, and heap-allocated frames cannot. Stacks should have very good locality even in a small cache. In a typical sequence of N procedure calls, the stack pointer is expected to go up and down over the same $\log(N)$ frames, re-using them over and over again. These frames should easily fit even in the smallest cache. Heap-allocated frames can have good locality in a large cache, but no one has analyzed locality in a small cache.

We will now demonstrate that heap-allocated frames have adequate locality of reference in a small cache, if the read miss penalty is not too large and the write miss penalty is zero.

5.1 Write misses

The Standard ML of New Jersey compiler [7] uses no stack; all frames are allocated on the garbage-collected heap. If any system should have poor cache locality, this is the one.

Diwan, Tarditi, and Moss[15] simulated the memory-hierarchy performance of SML/NJ on a DECstation 5000, and found two things:

- SML/NJ program executions have an astoundingly high write-miss ratio.
- SML/NJ programs are not much delayed by cache misses.

The reason these two statements are not inconsistent, they discovered, is that the write-miss penalty on this machine is approximately zero—they found that the write buffer can easily keep up with an enormous write miss rate. Read misses stall the processor–which cannot continue computing until the data shows up–but write misses can be handled by the write buffer while the CPU continues its work. Many modern machines have a zero write-miss penalty, especially for their primary caches [22]. Simulating machines with a high write-miss penalty, Diwan *et al.* found that SML/NJ performs very badly, as might be expected.

They also found that *write-allocate* is very important: on a write miss, the written data should be put in the cache. But a cache line is usually larger than a single word; on a write miss, "traditional" caches read the rest of the line from memory; this can cause write misses to be slow, and also causes unnecessary traffic on the memory bus in the common case of sequential writes that will overwrite the just-read data. The simulations of Diwan *et al.*, and our analysis in Section 5.3, both show that this policy is very costly.

Heap allocation (in a system with copying garbage collection) consists of sequential writes to a large contiguous free region. Under such a discipline, there are some equally good cache implementation strategies that will permit (or simulate) write-allocate with zero write-miss penalty.

Sub-block placement: With sub-block placement, a write miss on one word will be written to the cache, and the rest of that cache line will be marked as allocated but invalid. Thus, a write miss does not require reading the rest of the written cache line from memory. Subsequent (sequential) writes will fill the rest of the line.

One-word cache line: The DECstation 5000 has a cache-line size of one word, but four lines are read on a miss [15]. For some applications this is better than sub-block placement, but for sequential writes it is equally good. It is more expensive to implement, since it requires a full tag (not just a valid bit) for each word. Diwan *et al.* found excellent memory-subsystem performance of SML/NJ on this machine.

Cache-line zero instruction: On some machines (e.g., IBM R/S 6000[19] and PowerPC[1]) a cache line (64 bytes) can be allocated and zeroed with a special instruction. This avoids the write miss, with a 0.687-instruction cost per frame.[5]

[5] In detail: the allocation pointer is made always to point exactly 64 bytes ahead of the next allocatable word. On each heap-limit check, a cache-line clear is performed. This does not clear the line currently being stored into (which might overwrite a frame recently allocated) but the line *soon to be entered*. Because the heap-limit check is often shared with a non-frame allocation (see Figure 3), the average net cost *per frame* is only 0.687 instructions.

Cache-control hint: On the HP PA7100, a store instruction can have a cache-control hint specifying that the block will be overwritten before being read; this avoids the read if the write misses [9]. But these machines have very large primary caches anyway, so locality can be handled by generational collection.

Smart write buffer: Instead of sub-block placement (which complicates the cache), one might add a feature to the write buffer: write misses normally bypass the cache, but if the write buffer accumulates a full cache line, this line is put in the cache. For sequential writes this is as good as sub-block placement.

On any of these machines, heap-allocated data should not incur a write-miss penalty.

Assumption: Any small cache will have write-allocate and no write-miss latency.

Indeed, this is not true of all machines. In fact, the terrible performance of garbage-collected systems on machines with a write-miss penalty is a good reason not to build such machines.

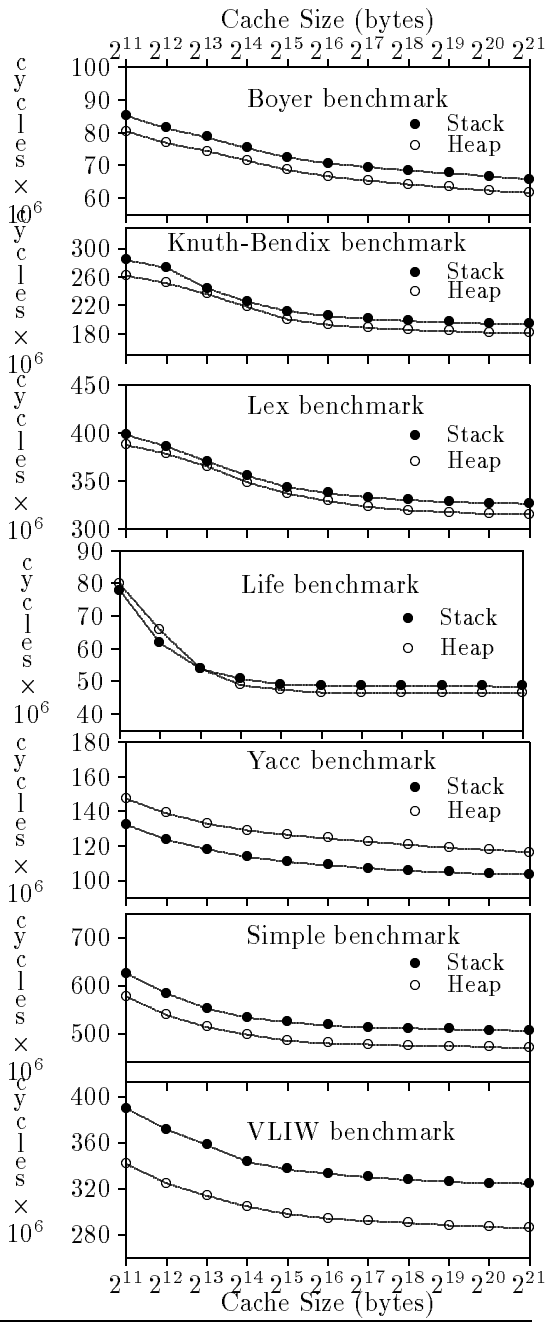
Finally, note that a write-miss penalty on large caches is not particularly problematic; as explained above, generational garbage collection solves that problem. The analysis in the rest of this section applies only to small caches.

5.2 Simulations

To see the effect of small caches on heap-allocated frames, we simulated several "standard" SML benchmarks in two versions of the SML/NJ compiler: Version H with heap-allocated frames, and version S with stack-allocated frames. The simulations counted read misses, write misses, and total instruction count of SML programs compiled to the MIPS instruction set. Our simulations include the instructions and cache misses of garbage collection.

We simulated direct-mapped caches of sizes ranging from 2 kbytes to 2 Mbytes, with a 32-byte line size. We simulated only the primary data cache, and instruction cache of equal size. Instead of a detailed cycle-level simulation, we use the approximation that each cache miss stalls the instruction-execution pipeline for p cycles, where $p = 10$ is the "miss penalty." This is not exactly how modern machines work, but it is still informative.

Our "stack" is really a freelist of 32-byte re-usable frames, similar in some ways to Danvy's[14]. Stack frames are not contiguous, which might seem to affect their locality of reference in a cache. But it should still be the case that a program will go "up and down" over the same small set of (noncontiguous) frames, and that even a small cache should



Simulations of benchmark programs[4] running in direct-mapped D-cache with of various sizes, with 10-cycle miss penalty, and "infinite" I-cache. Instruction count for stack programs reduced by 6× number of calls, to discount the 7-instruction quasi-stack allocation/deallocation sequence. Execution cycles are shown in millions. The heap version of the Yacc benchmark runs slower than the stack version; this is because our (stupid) two-generation collector does one extra major collection. A multi-generation collector would not fall into this trap. The VLIW benchmark (stack version) suffers terribly from the "stack can't point to heap" restriction.

Figure 5: Heap and stack simulated execution times

be able to hold these frames along with other frequently used data.

Our garbage collector "marks" any frame that survives a collection; marked frames are not put back on the freelist upon procedure return. This enables our stacks to work well with generational garbage collection and with first-class continuations. At a youngest-generation collection, the freelist is set to nil; after the collection, new frames will be obtained from the heap (and, when freed, put back on the freelist).

Using Danvy's "quasi stacks," there is a considerable cost to allocate and deallocate a frame:

1. Test the freelist register.[6]
2. Set freelist register to the next free frame.

To deallocate,

3 Fetch the mark field.

4 Wait for fetch to finish.

5 If marked, stop here (don't put back on free list).

6 Store free list register into newly freed frame.

7 Set the free list register to point to this frame.

Thus, there is an overhead of seven instructions for stack allocation. But "ordinary, contiguous" stacks don't have this seven-instruction penalty—there's just a single "pop" instruction. Therefore, we adjust the execution time of the Stack version of the program by subtracting six cycles per frame.

Figure 5 shows the run times (after adjustment) of several benchmarks using *Heap* and *Stack* frames, running in simulated caches of different sizes. Though there are many differences between the Heap and Stack implementations that affect the run time, it is clear from the shapes of the curves that the *cache locality* behavior of the two versions is almost identical. (That is, if the two curves were translated vertically so that the large-cache points coincide, then the rest of the curves would be extremely close.)

how can this be, given the supposed superiority of stacks in locality of reference?

5.3 Read misses

Stacks continually re-use recently-used frames for new purposes; heap-allocated frames "thrash" through the cache in sequential order. How could it possibly be the case that they both have equally good locality of reference?

Since the simulations give us no analytical understanding of what is really happening, we will focus on three "typical" patterns of procedure call and return:

1. Tail recursion: each call re-uses the same frame, no allocation is performed.

2. Deep single recursion (as for the recursive factorial function): a very deep sequence of calls followed by the returns from all of them.

3. "Towers of Hanoi:" lots of short up-and-down motion, but every Nth call briefly returns to depth $\log(N)$. This is shown graphically in figures 6 and 7.

We claim that these patterns represent "both extremes and the middle" of all procedure-call patterns.

We assume that the cache is direct-mapped and can hold C frames, with a line size equal to the size of a frame (different line sizes would affect our results by a constant factor; the frame size in the compiler used for our simulations (previous section) is 32 bytes). We assume that heap-allocated frames are allocated at sequential addresses. We assume that no heap allocations or memory accesses are performed, other than for activation records.

Then *tail recursion* is easy to analyze: No new frames are allocated by either the stack or heap methods. The miss rate for both is zero.

Deep single recursion is bad for both stacks and heaps: If the recursion is depth N, then there will be no read misses on any of the calls; the first C returns will hit, and the rest of the returns will have read misses. The miss ratio approaches 1 as N becomes much larger than C.

Towers of Hanoi should be ideal to demonstrate the better locality of reference of stacks in a small cache. Let us analyze its cache behavior carefully.

```
procedure T(d) =
  if d>1 then { T(d-1);
                T(d-1)
              }
```

Executing $T(d)$ requires $2^d - 1$ procedure calls. If $d \leq C$, all the frames fit in the cache, so the read-miss rate is zero (we will ignore write misses). If $d > C$, then there will be a miss on every return from a call to $T(e)$, where $e > C$. The number of such returns is $2^{d-C} - 1$. The miss rate is the number of misses divided by the number of calls, or approximately 2^{-C}. For an 8-kbyte cache, $C = 256$, so the miss rate for stacks is indeed negligible.

[6] If the freelist is empty, one must heap-allocate instead of taking from the freelist (the heap-allocated frame will be deallocated back onto the freelist). But this case is so rare that we won't count it in the average cost.

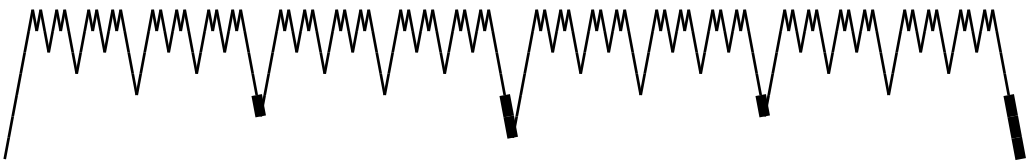


Figure 6: Execution of $T(7)$ in a 16-line cache. Every uptick (procedure call) is a write miss; only the bold downticks (procedure returns) are read misses.

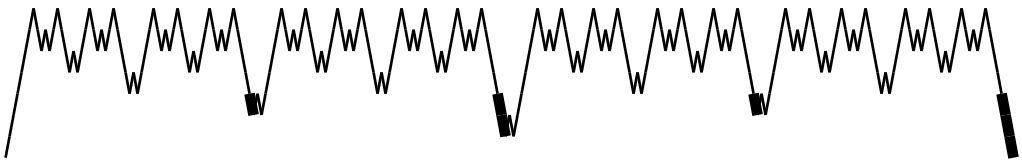


Figure 7: Execution of $T'(6)$ in a 16-line cache. Only the bold downticks (procedure returns) are read misses.

Now consider executing $T(d)$ with heap-allocated frames. The frames are allocated sequentially. If a clock counts one tick for each procedure call, then at time t the frame created at time $t - C$ will be removed from the cache by a newly allocated frame. Therefore, exactly those procedures that return more than C time units after they are entered will cause a cache miss upon return to their callers.

Executing $T(d)$ requires $2^d - 1$ calls, of which only $2^{d-\log_2 C}$ take more than C time units to execute. Thus, the amortized number of misses per call is about $2^{-\log_2 C}$, that is, one miss every C calls.

Figure 6 illustrates $T(7)$ running in a tiny cache ($C = 16$). The cache misses are clustered at intervals of $2C$ returns, but there are an average of 2 misses in each cluster.

Figure 7 shows a more traditional Towers of Hanoi, that calls `print("move disk d")` at each step.

In an 8k cache—which can hold 256 32-byte frames—the stack version will have one miss every 2^{256} calls, but the heap version will have one miss every 256 calls. Assuming that the primary cache miss (with secondary cache hit) costs 10 cycles, this is a cost of *one cycle every 25 calls.*

We claim this cost is affordable; we have rounded it up, in Figure 1, as 0.1 instructions per call.

But now consider what happens in a cache without *write-allocate*. The vast majority of reads in the Towers of Hanoi example are to blocks that recently caused write misses. Even if the write misses themselves do not stall the processor, the read misses will. This will cost perhaps 10 cycles per frame, a very high cost.

Real programs are neither the factorial function (deep recursion) nor the Towers of Hanoi; they are not so easy to analyze in closed form. For real programs one must go back to the benchmark simulations, which also show that heap allocation of frames does not seem to increase the number of read misses very much.

6 Disposal

To de-allocate a stack frame, one instruction is required to subtract a constant from the stack pointer.

No explicit pop instruction is necessary to deallocate a heap frame. The (previous) frame pointer must be fetched, but we have counted this already under the heading "Frame pointers."

However, there is some cost for the eventual garbage collection of dead frames. We can also analyze the cost, in instructions, of garbage collection for the three typical call-return patterns discussed in the previous section. We assume a generational collector with a youngest generation that holds G frames (for generation size of 64 kbytes, frame size of 32 bytes, $G = 2048$). On a youngest generation collection, all live frames are promoted to the next generation.

1. Tail recursion does not allocate, so the amortized cost per call is zero.

2. Towers of Hanoi will garbage-collect every G calls. At this collection, there will be at most $\log(d)$ live frames; but only $\log(G)$ of them will be "new" (not already promoted). The cost of collecting them will be $c\log(G)$, where c is the cost of copying one frame (perhaps 20 instructions).[7] The amortized cost per call is

[7] We do not include the cost of scanning the frame for pointers, because this has to be done for either the stack or the heap case.

$c\log(G)/G$, or about 0.1 instruction per call. Cheaper than popping the stack!

3. A very deep recursion (deeper than 2000 calls) will promote almost every frame, at a cost of cG instructions per G calls, or c instructions per call. This is costly; but recursions this deep also begin to miss in the secondary cache! This is particularly so, since the size of the youngest generation should be about half the size of the secondary cache [35]. The secondary cache misses (which will occur for stacks or heaps) will probably be just as important as the garbage-collection overhead.

More frequent collections

In principle, the heap allocation of frames should cause more frequent collection (and undesirable promotion) of the other data. We use lifetime statistics as reported by Stefanovic[30] to find those objects that survive G calls but not $G/2$. Because of the shape of the object-survival curve, such objects are quite rare (1 object in 1000 allocations).

Consider a program that allocates one "ordinary" (non-frame, "random" lifetime) object per procedure call. One in 1000 of these objects will be promoted "to excess" (in the heap-frame version) because the frame allocations cause more frequent collection. The cost of each such promotion (forwarding and scanning) is about 100 instructions. Thus the average cost per frame is about 0.1 instructions. The sum of the two 0.1 costs (for copying frames, and for excess promotion of non-frames) is shown in Figure 1.

As with the closed-form locality analyses, these are not real programs. The simulations show a low overhead for garbage collection of real programs.

7 Finding roots

In any garbage-collected system, local variables in activation records (e.g., stack frames) may point to the heap. At the beginning of each garbage collection, the collector must scan the frames to locate "roots" of the live data.

In a system with generational garbage collection, there is often very little live data in the youngest generation. Scanning a very large stack would take more time than the rest of the collection! Therefore, the collector should scan only those stack frames *created since the last collection* and not yet popped.

It is trivial to treat heap-allocated frames this way. They are promoted (along with other live data) to older generations; older-generation data need not be scanned at a youngest-generation collection. Only the newly-allocated (and not yet dead) frames will be scanned at a typical collection.

With stacks, a special trick is required. After a collection, the collector must mark the top stack frame. All frames underneath this are known to be "old." At the next collection, the stack must be scanned only from the top of the stack down to the "high-water mark;" for only these frames can contain pointers to the youngest generation.

But there is a complication. Between collections, if the "high-water" frame is popped, the mark must be moved down to the next-lower frame [33]. The simplest way to do this would be to test for the mark on every return, but this would be quite expensive. Instead, the mark consists of a "special" return address, which replaces the real return address of a frame. When control returns to this point, the program at this special location executes, placing the mark (that is, the special return address) in the next-lower frame, and jumping to the real return address.[8]

The cost of this technique is quite low. The cost of placing and removing the high-water mark is between 10 and 100 instructions. Every frame that survives its first garbage collection will eventually hold the high-water mark. The cost of moving the high-water mark (in a stack-based system) is similar to the cost of promoting a live stack frame to the older generation (in a heap-based system); and it is exactly the same frames (new frames live at a collection) that need this service in either case.

The proportion of stack frames live at a collection is usually extremely low, so the cost is negligible for both stacks and heaps. In rare cases (very deep one-way recursions) the cost will be higher, but the stack-based systems and heap-based systems will pay approximately the same price.

Gonthier and Doligez[16] have suggested that the collector put a one-bit mark in *every* live stack frame that it scans; this mark will be ignored by the collector but will be cleared in new frames. This is fine, if there is already some word in every frame that has a free bit.

Keeping track of the high-water mark in heap-based system has *no* implementation complexity: it is a natural consequence of garbage-collecting live frames. In contrast, in a stack-based system similar results can be achieved but it requires extra work.

[8] This complicates the compiler and runtime system, particularly the implementation of exception handlers that must pop the stack.

7.1 Updating activation records

In order to guarantee that only "new" heap frames can be roots for garbage collection, it is necessary to prohibit any writes to frames after they have been allocated. Compilers using continuation-passing style (such as Rabbit [29], Orbit [23], and SML/NJ[5]) naturally initialize frames as soon as they are allocated, and then never write to them again. In effect, they save up any "changes" in registers, then dump everything out all at once. With good use of callee-save registers[8, 4, 28] it is even easier to accumulate any changes in registers and write immutable frames in big chunks.

A stack-based compiler could update the topmost frame at any time, and the collector could always scan this frame for roots. But a heap-based compiler that wants to support efficient call/cc (see Section 8) should never update a frame after its initialization, because if a continuation is invoked more than once the two invocations will stomp on each others' data. In such a compiler, it is best to keep the top frame in callee-save registers and not in memory at all. Of course, this is probably a good idea in any case.

8 First-class continuations

The notion of "first class continuations" using the *call-with-current-continuation* (call/cc) primitive originated in the Scheme language [25] and has since been adopted in other systems as well [17]. First class continuations are extremely useful for implementing coroutines[32], concurrency libraries[26] and multitasking.

But call/cc is much harder to implement efficiently if there is a stack; with an ordinary contiguous stack implementation, the entire stack must be copied on each creation or invocation of a first-class continuation. This is unacceptable if (for example) call/cc is the primitive used in implementing a concurrency library or exception-handling system.

With purely heap-allocated frames (that are not updated after their initialization), call/cc is no more expensive than an ordinary procedure call: the live registers must be written to a closure record, and that is all.

There have been mixed stack/heap implementations intended to support call/cc efficiently in the presence of stacks[13, 20]. The basic idea is to make a "stack chunk" that holds several stack frames; if this fills, it is linked to another chunk allocated from the heap. This turns out to be quite complicated to implement.

Danvy made a free list of re-usable frames (we call this a "quasi-stack"); these reduce the load on the garbage collector and have very good locality; but they are expensive to create and destroy, and require a frame pointer.

Both methods suffer from the same "copying, sharing" penalty as ordinary stacks. Their performance is summarized in Figure 8, and does not appear competitive (especially given the implementation complexity).

Component	Stack Chunks[20]	Quasi-Stack[14]
Creation	3.0[9]	3.0
Frame pointers	0.0	2.0
Copying, sharing	3.4	3.4
Cache read misses	0.0	0.0
Cache write misses	0.0	0.0
Disposal (pop)	1.0	4.0
Total Cost	7.4	12.4
Call/cc	X[10]	$O(1)$

Figure 8: Cost of other stack-like disciplines (instructions per frame).

The simplicity and efficiency of call/cc in a pure heap discipline is a very strong motivation for avoiding stacks.

9 Implementation

One reason to avoid stacks is that they are complicated to implement, especially with all the tricks that are necessary to achieve good performance. Let us compare the implementation complexities of heaps vs. stacks, in a garbage-collected environment:

9.1 Implementation of Heap Frames

1. To achieve good performance with heap frames, it is necessary to have an sophisticated algorithm to choose closure representations. This algorithm must preserve space-complexity, promote closure-sharing, and use callee-save registers to minimize the number of distinct frames written. Shao[28] describes an implementation of such an algorithm, which is not particularly hairy.

2. To avoid having a descriptor in each frame, the runtime system must maintain a mapping of return addresses to frame layout descriptors.

[9] "Unfortunately, it has been our experience that memory exceptions are not a tenable means for detecting stack overflow...." [20]

[10] Proportional to the size of one stack chunk.

9.2 Implementation of Stacks

1. A good closure analysis algorithm must be used to preserve space-complexity while still trying to avoid too much copying. It is not clear that such an algorithm will be much simpler than the one for pure heaps. In particular, most conventional stack implementations are not safe for space complexity.

2. To preserve space complexity and correctly implement tail recursion, certain activation records require a very complicated scheme to determine when they must be popped[18]. (Or these frames could be heap allocated, even in a stack discipline; but they must be identified by static analysis.)

3. A high-water mark must be maintained to achieve efficiency in the generational collector.

4. If call/cc is to be supported, then stack copying or some more complicated technique must be implemented [20].

5. To avoid having a descriptor in each frame, the runtime system must maintain a mapping of return addresses to frame layout descriptors.

6. In a system with multiple threads, each thread must have its own stack. A large contiguous region of virtual memory must be reserved.[11]

7. Stack-overflow detection must be implemented. In most cases this is handled automatically by the operating system using virtual-memory page faults.[12]

No stack implementation that we know of handles all of these necessary complexities. As a result, some are not safe for space complexity; some don't implement call/cc; some scan too many frames on each collection. It is an open question whether all of these tricks can fit together in a real system.

10 Conclusion

Heap allocation of activation records is simple, and competitively efficient. It is quite tricky to make stacks "safe for space complexity;" and the locality of reference of heap allocation is not too bad. This is particularly true in the implementation of modern programming languages (with closures, garbage collection, few side effects, first class continuations) on modern machines (efficient write miss handling).

The *Standard ML of New Jersey* compiler is an example of such an implementation: heap allocation of activation records has proved to be a great success.

When *call-with-current-continuation* is needed, heap frames are *much* better than stack frames. Various hybrid systems (Figure 8) designed to support *call/cc* efficiently with a stack are less efficient than heaps for *both* normal call/return *and* call/cc.

On machines with a write-miss penalty, or where writes entirely bypass the cache, the results are quite different, of course. But since many C programs—not just ML programs—also have high write-miss ratios [22], perhaps we should not expect to see very many processors of this design.

[11] In contrast, one heap-allocation region is necessary *per processor*, not per thread.

[12] Heap overflow detection must also be implemented, but this is true whether or not there is a stack.

Acknowledgements

Gün Sirer and Marcelo Gonçalves implemented and adapted the MIPSI instruction simulator that we used in our measurements. Hans Boehm, Amer Diwan, Damien Doligez, Lorenz Huelsbergen, Xavier Leroy, Paul Wilson, and others made useful comments on early drafts of the paper.

References

[1] Michael S. Allen and Michael C. Becker. Multiprocessing aspects of the PowerPC 601. In *IEEE COMPCON Spring '93*, pages 117–126. IEEE Computer Society Press, February 1993.

[2] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–79, 1987.

[3] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software—Practice and Experience*, 19(2):171–83, 1989.

[4] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[5] Andrew W. Appel and Trevor Jim. Continuation-passing, closure-passing style. In *Sixteenth ACM Symp. on Principles of Programming Languages*, pages 293–302, New York, 1989. ACM Press.

[6] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In *Fourth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (SIGPLAN Notices v. 26, no. 4)*, pages 96–107. ACM Press, April 1991.

[7] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Martin Wirsing, editor, *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13, New York, August 1991. Springer-Verlag.

[8] Andrew W. Appel and Zhong Shao. Callee-save registers in continuation-passing style. *Lisp and Symbolic Computation*, 5:189–219, September 1992.

[9] Tom Asprey, Gregory S. Averill, Eric DeLano, Russ Mason, Bill Weiner, and Jeff Yetter. Performance features of the PA7100 microprocessor. *IEEE Micro*, 13(3), June 1993.

[10] Lennart Augustsson. Garbage collection in the $< \nu, g >$-machine. Technical Report PMG memo 73, Dept. of Computer Sciences, Chalmers University of Technology, Goteborg, Sweden, December 1989.

[11] Henry G. Baker. The buried binding and stale binding problems of LISP 1.5. unpublished, undistributed paper, June 1976.

[12] David R. Chase. Safety considerations for storage allocation optimizations. In *Proc. ACM SIGPLAN '88 Conf. on Prog. Lang. Design and Implementation*, pages 1–9. ACM Press, 1988.

[13] William D Clinger, Anne H Hartheimer, and Eric M Ost. Implementation strategies for continuations. In *1988 ACM Conference on Lisp and Functional Programming*, pages 124–131, New York, June 1988. ACM Press.

[14] Olivier Danvy. Memory allocation and higher-order functions. In *Proceedings of the SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques*, pages 241–252. ACM Press, June 1987.

[15] Amer Diwan, David Tarditi, and Eliot Moss. Memory subsystem performance of programs using copying garbage collection. In *Proc. 21st Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 1–14. ACM Press, 1994.

[16] Damien Doligez and Georges Gonthier. Re: stack scanning for generational g.c. E-mail message `<9403041606.AA07877@lix.polytechnique.fr>`, March 1994.

[17] Bruce Duba, Robert Harper, and David MacQueen. Typing first-class continuations in ML. In *Eighteenth Annual ACM Symp. on Principles of Prog. Languages*, pages 163–73, New York, Jan 1991. ACM Press.

[18] Chris Hanson. Efficient stack allocation for tail-recursive languages. In *1990 ACM Conference on Lisp and Fucntional Programming*, pages 106–118, New York, June 1990. ACM Press.

[19] William R. Hardell, Dwain A. Hicks, Lawrence C. Howell, Warren E. Maule, Robert Montoye, and David P. Tuttle. *Data Cache and Storage Control Units*, pages 44–50. IBM, 1990.

[20] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In *Proc. ACM SIGPLAN '90 Conf. on Prog. Lang. Design and Implementation*, pages 66–77, New York, 1990. ACM Press.

[21] Richard Jones. Tail recursion without space leaks. *Journal of Functional Programming*, 2(1):73–79, 1992.

[22] Norman P. Jouppi. Cache write policies and performance. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 191–201. ACM Press, May 1993.

[23] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams. ORBIT: An optimizing compiler for Scheme. *SIGPLAN Notices (Proc. Sigplan '86 Symp. on Compiler Construction)*, 21(7):219–33, July 1986.

[24] Xavier Leroy. Unboxed objects and polymorphic typing. In *Nineteenth Annual ACM Symp. on Principles of Prog. Languages*, pages 177–188, New York, January 1992. ACM Press.

[25] J. Rees and W. Clinger. Revised report on the algorithmic language Scheme. *SIGPLAN Notices*, 21(12):37–79, 1986.

[26] John H. Reppy. CML: A higher-order concurrent language. In *Proc. ACM SIGPLAN '91 Conf. on Prog. Lang. Design and Implementation*, pages 293–305. ACM Press, 1991.

[27] Colin Runciman and David Wakeling. Heap profiling of lazy functional programs. *Journal of Functional Programming*, 3(2), April 1993.

[28] Zhong Shao and Andrew W. Appel. Space-efficient closure representations. In *Proc. 1994 ACM Conf. on Lisp and Functional Programming*, page (to appear). ACM Press, 1994.

[29] Guy L. Steele. Rabbit: a compiler for Scheme. Technical Report AI-TR-474, MIT, Cambridge, MA, 1978.

[30] Darko Stefanovic and J. Eliot B. Moss. Characterization of object behaviour in Standard ML of New Jersey. In *Proc. 1994 ACM Conf. on Lisp and Functional Programming*, page (to appear). ACM Press, 1994.

[31] David M. Ungar. *The Design and Evaluation of a High Performance Smalltalk System*. MIT Press, Cambridge, MA, 1986.

[32] Mitchell Wand. Continuation-based multiprocessing. In *Conf. Record of the 1980 Lisp Conf.*, pages 19–28, New York, August 1980. ACM Press.

[33] Paul R. Wilson. Some issues and strategies in heap management and memory hierarchies. *SIGPLAN Notices*, 26(3):45–52, March 1991.

[34] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Caching considerations for generational garbage collection. In *1992 ACM Conference on Lisp and Functional Programming*, pages 32–42, New York, June 1992. ACM Press.

[35] Benjamin Zorn. The effect of garbage collection on cache performance. Technical Report CU-CS-528-91, University of Colorado, Boulder, CO, May 1991.