J. Functional Programming **6** (1): 47–74, January 1996 © 1996 Cambridge University Press

Empirical and analytic study of stack versus heap cost for languages with closures

ANDREW W. APPEL[1] AND ZHONG SHAO[2]

Department of Computer Science, Princeton University, Princeton, NJ 08544-2087, USA

Abstract

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.
- 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 small.
- The demands of modern programming languages make stacks complicated to implement efficiently and correctly.

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

Capsule Review

The paper describes the choice between stack-based and heap-based activation records for SML/NJ. The paper is important since it gives a detailed account of *all* the costs that this choice may incur. This includes not only the obvious costs of the management of the stack or heap, but also the hidden costs of additional cache misses. It is shown that there is no penalty in using heap-based activation records, and that there are significant implementation benefits.

The paper should be of interest to any implementor of languages with automatic storage management, although generalising the findings from the world of SML/NJ will not be entirely straightforward.

[1] Supported in part by NSF Grant CCR-9200790.

[2] Current address: Department of Computer Science, Yale University, New Haven, CT 06520-8285, USA.

Component	Heap	Stack	Stack Chunks (see §9)	Quasi-Stack (see §9)	see:
Creation	3.1	1.0	3.0	3.0	section 2
Frame pointers	2.0	0.0	0.0	2.0	section 3
Copying and sharing	0.0	3.4	3.4	3.4	section 4
Cache write misses	0.0 or 5.3	0.0	0.0	0.0	section 6.1
Cache read misses	1.0	0.0	0.0	0.0	section 6.2
Disposal (pop)	1.4	1.0	1.0	4.0	section 7
Total Cost	**7.5** or 12.8	**5.4**	**7.4**	**12.4**	
Call/cc	$O(1)$	$O(N)$	$O(X)$	$O(1)$	section 9
Implementation	easy	hard	hard	hard	section 10

Fig. 1. Cost breakdown of different frame allocation strategies. 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 accompanying text explains why we count instructions instead of cycles. 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. In the last row, N is the stack depth; X is the size of one stack chunk.

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 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 (Appel, 1987): modern generational garbage-collection algorithms (Ungar, 1986) can reclaim dead frames efficiently, as cheap as the one-instruction cost to pop the stack.

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

These numbers depend upon 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 com-

plexity'; that is, statically dead pointers (in the dataflow sense) do not keep objects live. (See section 5.)

- There are few side effects in compiled programs, so that generational garbage collection will be efficient.

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

Figure 1 clearly shows that there are three important criteria in the choice between a stack or heap representation:

1. The write-miss policy of the machine's primary cache (discussed in section 6.1). On machines with *fetch-on-write* or *write-around* write-miss policies, heap-allocated frames are significantly more expensive.
2. Stacks are harder to implement without space leaks, as explained in section 10.
3. If the programming language supports *call-with-current-continuation* (call/cc) – a primitive often used to support tasking, coroutines, exceptions, and so on – stacks have a much higher cost (see section 9).

The (perhaps) startling result is that heap-allocated frames have almost the same cost as stack frames.

Finally, we point out that the absolute differences are small: two instructions per frame is less than 2% of total execution cost, as can be calculated from Figure 4.

We count *instructions* rather than *cycles*. In general, *load* and *store* instructions for frame management can usually be scheduled to avoid stalls (they are rarely in the critical path of a loop, for example). The *branch* instructions for heap-limit testing will be at least 99% predictable – because hundreds of frames are allocated (heap limit not exceeded) between garbage collections (heap limit exceeded); so branches for heap-limit tests will cause almost no stalls. Thus, instruction counts, plus a separate accounting of the cache misses, form a suitable cost model.

2 Creation

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

Allocating a heap frame is more complicated:

1. Heap overflow must be checked. As explained by Appel and Li (1991), and contrary to the silly ideas of Appel (1989), this should not be done by a virtual memory fault: (1) operating-system fault handling is too expensive, (2) heap overflow is unrelated to locality of reference, and (3) the technique is almost impossible on machines without precise interrupts.
   Thus, a comparison and a conditional branch are 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

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.

Fig. 2. General information about the benchmark programs.

Program	Limit Checks per Frame
Boyer	0.717
Knuth-B	0.783
Lexgen	0.864
Life	0.456
YACC	0.631
Simple	0.665
VLIW	0.695
Average	**0.687**

Many frame allocations are in the same (extended) basic block as other non-frame 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 a non-frame allocation. The results shown are from measurements of ML benchmark programs (see Figure 2) as compiled by the *Standard ML of New Jersey* (Appel and MacQueen, 1991) compiler.

Fig. 3. Shared limit checks.

basic block[1] 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

[1] An extended basic block has one entry point, followed by a tree of control flow with several exits.

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 *move* instruction.

The total cost is about 3.1 instructions, on 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.

Other registers

Efficient heap allocation uses a *free-space pointer* and a *free-space limit* which should be kept in registers.[3] However, the cost of reserving these registers should not be charged to heap allocation of frames, because we are assuming that the implementation in question already has garbage collection (presumably with efficient allocation) for other purposes (lists and closures, for example).

4 Copying and 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 behaviour; 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*

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

[3] Some implementations use a BIBOP (BIg Bag Of Pages (Hanson, 1980)) scheme that allocates each kind of object in a different contiguous space, so that only one g.c.-descriptor is required per space, instead of per object. This requires a free-space pointer and a limit pointer *per space*.

Program	Ordinary Heap $i/10^3$	'Stacklike' Heap $i/10^3$	Stack Frames $f/10^3$	Instrs per Frame	Extra Instrs per Frame
Boyer	61966	65770	662	94	5.75
Knuth-B	212702	222376	3465	61	2.79
Lexgen	310522	316813	1873	166	3.36
Life	48016	48437	201	239	2.09
YACC	114687	119065	1187	97	3.69
Simple	469543	492126	5516	85	4.09
VLIW	285370	292474	3274	87	2.17
Average				119	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 third column shows thousands of frames created. We count *frames* rather than *calls* because tail calls and leaf procedures do not make frames (stack or heap).

Fig. 4. Copying/sharing cost.

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 (Shao and Appel, 1994).

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 (Appel and MacQueen, 1991; Appel, 1992) outfitted with our recently improved closure-representation analysis phase (Shao and Appel, 1994).

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 the 'closures cannot point to frames' restriction. The frames themselves are not much bigger, but

Program	Stack Frames $f/10^3$	Heap Frame Alloc. *words*$/10^6$	Other Alloc. *words*$/10^6$	$\frac{F}{F+O}$	Avg. Frame Size *words*
Boyer	662	3.17	2.61	0.55	4.8
Knuth-B	3465	12.92	11.55	0.53	3.7
Lexgen	1873	6.90	1.87	0.79	3.7
Life	201	0.63	0.99	0.39	3.1
YACC	1187	5.92	4.16	0.59	5.0
Simple	5516	25.23	13.09	0.66	4.6
VLIW	3274	15.72	15.90	0.50	4.8
Average				0.57	4.2

This table shows the amount of frame allocation, the amount of non-frame allocation, and the proportion of allocation due to heap frames for the heap-based compiler. The last column shows the average frame size, calculated from the previous columns. Even though the number of frames used by the heap-based compiler is slightly less than the number used by the stack-based compiler (because of improved copying/sharing) we use the stack-frame count for calculation, to make comparison between the two compilers more meaningful.

Fig. 5. Heap allocation data.

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 average the difference is quite significant: about 3.4 extra instructions are executed per every frame creation because of this restriction. Perhaps our lambda-lifting (closure analysis) algorithm is better tuned for heaps than it is for stacks, and this 'copying vs. sharing' cost is overstated; it is difficult to tell.

5 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 important: if the collector traverses too many dead variables, the memory use of the program can increase by a large factor (Baker, 1976; Chase, 1988; Runciman and Wakeling, 1993; Appel, 1992; Jones, 1992).

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, where N is the size of the input. This theorem, examples, and a description of compiler techniques that are 'safe for space complexity' are described by Appel (1992).

An illustrative example is shown in Figure 6. Closures are created for function definitions (at the `fun` keyword). The function f returns as its result a nested function g, which returns a nested function h, which returns a nested function i and a value u computed by selecting the head of the list v. The function $big(n)$ makes a list of length n, and *loop* makes a list of n closures over the function h.

```
fun f(v,w,x,y,z) =
  let fun g() =
        let val u = hd(v)
            fun h() =
              let fun i() = w+x+y+z+3
               in (i,u)
              end
         in h
        end
   in g
  end

fun big(n) = if n<1 then [0] else n :: big(n-1)

fun loop (n,res) =
  if n<1 then res
  else (let val s = f(big(N),0,0,0,0)()
         in loop(n-1,s::res)
        end)

val result = loop(N,[])
```

Fig. 6. An illustration of space-complexity traps.

With flat closures,[4] each evaluation of $f(\ldots)()$ yields a closure s for h that contains just a few integers u, w, x, y, and z; the final result (i.e. *result*) contains N copies of the closure s for h, thus it uses at most $O(N)$ space.

With a common implementation of closures, using *static links* that point to activation records of outer functions, each closure s for h contains a pointer to the closure for g, which contains a list v of size N. Since the final *result* keeps N closures for different instantiations of h simultaneously – each with a different (large) value for the variable v – it requires $O(N^2)$ space consumption instead of $O(N)$. This space leak is caused by inappropriately retaining some 'dead' objects (v) that should be garbage collected earlier.

Such space leaks are unacceptable. Closure (and frame) representations must not cause space leaks. *Standard ML of New Jersey*, for example, avoids any closure

[4] A *flat closure* (Cardelli, 1984) is a record that holds only the free variables needed by the function.

representation or compiler 'optimisation' that could cause such a space leak (Appel, 1992; Shao and Appel, 1994).

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 and sharing' penalty for stack frames.

For example, in the Chalmers Lazy ML compiler (Augustsson, 1989) or the Gallium compiler (Leroy, 1992), associated with each return address is a descriptor telling which variables in the caller's frame are live *after the return.*[5] 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 complicated to implement, and we do not know of anyone who has done it.

6 Locality of reference

Stacks have excellent locality of reference: they are (almost) always moving up and down in a 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 behaviour 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 (Ungar, 1986) can keep its youngest generation entirely within the cache (Wilson *et al.*, 1992; Zorn, 1991). 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 die especially young. It will be extremely rare

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

for an activation record to be promoted to a higher generation (Stefanovic and Moss, 1994). Since only the higher generations can cause cache misses,[6] heap-allocated frames will (almost) 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.

John Reppy has recently made empirical measurements of a multigeneration collector on a machine with a large (1MB) secondary cache. 'The total CPU time reaches a minimum [significantly less than with the collector described in the current paper] when the allocation arena is the same size as the secondary cache. This provides empirical evidence for the claim that sizing the allocation space to fit into cache can improve performance'. (Reppy, 1993) Unfortunately, the measurements in our current paper were made using the older two-generation collector (Appel, 1989).

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 for 'balanced' performance.

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 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.

6.1 Write misses

The Standard ML of New Jersey compiler (Appel and MacQueen, 1991) 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 *et al.* (1994) simulated the memory-hierarchy performance of SML/NJ on a DECstation 5000, and found two things:

[6] This is a slight oversimplification.

- 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 – the write buffer can easily keep up with an enormous write miss rate.[7] 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 (Jouppi, 1993). Simulating machines with a high write-miss penalty, Diwan *et al.* found that SML/NJ performs badly, as might be expected.

Thus, on machines with a zero write-miss penalty, the average cost per frame of write misses is zero.

On machines with a nonzero write-miss penalty, the cost per frame is high. The average number of cache write misses caused by the creation of a frame is the ratio of frame size to cache line size (there is no fragmentation, because heap allocation is sequential and contiguous). Assuming a cache line size of 8 words (for example), and a frame size of 4.2 words (as in Figure 5), the number of write misses per frame is about 0.53.

Thus, the cost of write misses shown in Figure 1 is either 0 (for zero write penalty) or 5.3 (for 10-cycle write penalty). (But see also section 6.2.)

They also found that *write-allocate* is 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' *(fetch-on-write)* 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 6.3, both show that this policy is 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 (also called *write-validate*), 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 (Diwan *et al.*, 1994). 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)

[7] Reinhold (1994) makes similar observations about the interaction of garbage collection and caches, though not for a compiler with heap-allocated frames.

for each word. Diwan *et al.* found excellent memory-subsystem performance for SML/NJ on this machine.

Cache-line zero instruction. On some machines (e.g. IBM R/S 6000 Hardell *et al.*, 1990 and PowerPC (Allen and Becker, 1993)) 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.[8]

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 (Asprey *et al.*, 1993). But HP 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 a multiprocessor with cache coherence, this technique might be easier to implement than sub-block placement, because no cache line would ever be dirty (but partially full) in two different caches.

Garbage-prefetch. On a machine with a no-write-allocate (*write-around*) cache, write-allocate can be simulated (as long as read misses are nonblocking) by fetching the cache line (with an ordinary **read** instruction) in advance of the write (Appel, 1994). This technique works (providing a modest performance enhancement) on the DEC Alpha 21064 (Digital Equipment Corp., 1992), for example.

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 (or write-allocate can be emulated).

Indeed, this is not true of all machines: the VAX 11/780 and VAX 8800 do *write-around*, bypassing the primary cache on write misses (causing subsequent read misses); and most pre-1993 designs do *fetch-on-write,* stalling the processor on a write miss (Jouppi, 1993). In fact, the bad 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.

[8] 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.

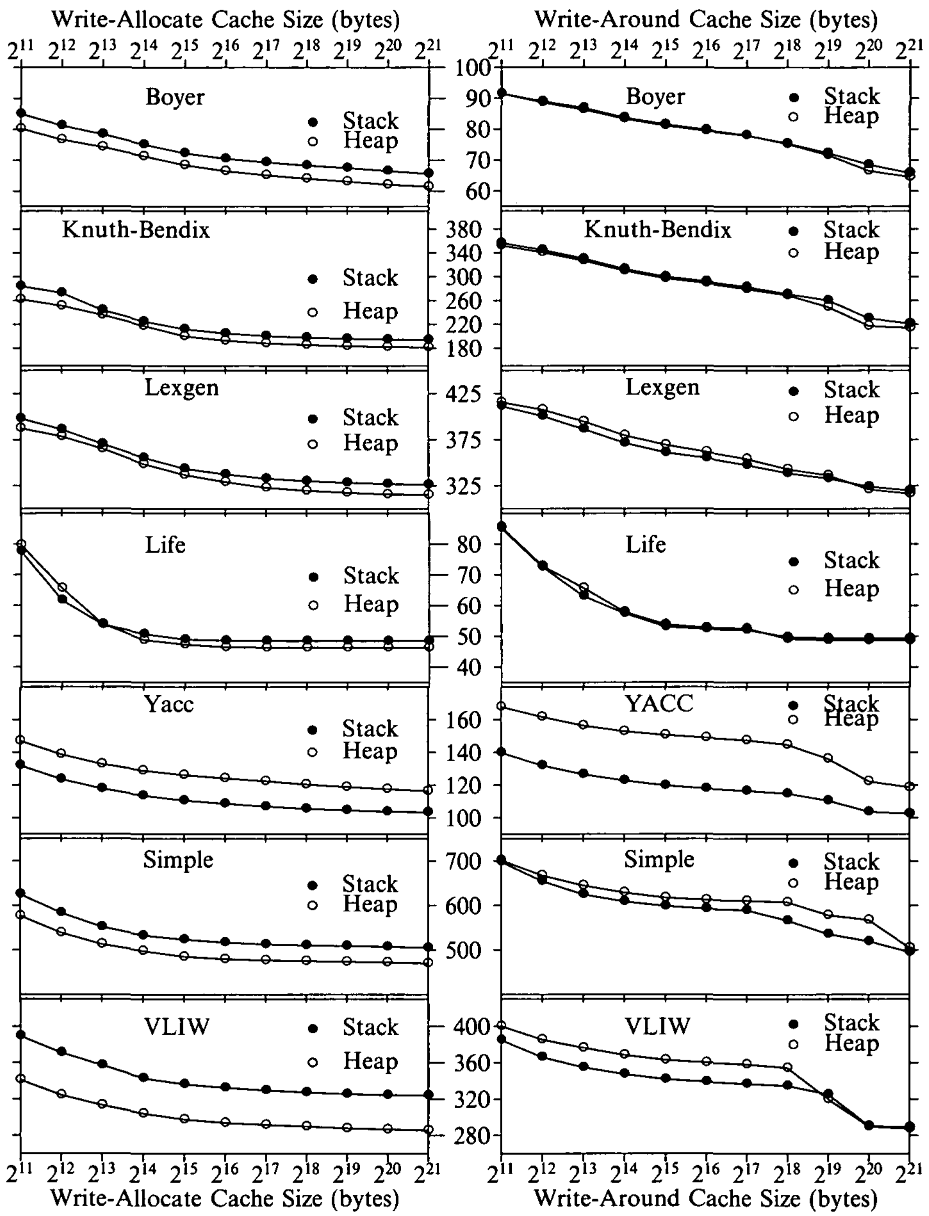

Fig. 7. Write-allocate vs. write-around cache. Simulations of benchmark programs(Appel, 1992) running in direct-mapped D-cache of various sizes, with 10-cycle read miss penalty, no write miss penalty, and 'infinite' I-cache. Left-hand-side shows write-allocate cache with partial fill; right-hand-side shows write-around cache. Vertical axis shows execution cycles in millions. Cycle count for stack programs is reduced by 6× number of frames, to discount the 7-instruction quasi-stack allocation/deallocation sequence. 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.

6.2 Read misses: 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: a *Heap* version with heap-allocated frames, and a *Stack* version 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.

Diwan *et al.* (1994) measured a heap-only ML system; Reinhold and Moss (1994) measured a stack-frame Scheme system. To make a more direct comparison, we measured stack frames vs. heap frames in the same ML system.

We simulated only the primary data cache. We simulated direct-mapped caches of sizes ranging from 2 kbytes to 2 Mbytes, with a 32-byte line size. Most modern machines have direct-mapped caches especially at the first level of the memory hierarchy, so that tag comparison can be overlapped with further computations on the value fetched (Hill, 1988).

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'. Many modern machines do not stall non-memory instructions on a cache miss; for these machines our simulation will provide an upper bound on cache delays, which is sufficient for our analysis.

We did not simulate a conventional, contiguous stack. Instead, we implemented a free list of 8-word re-usable frames (a *quasi-stack*). Frames are popped by putting them back on a free list. This takes more instructions than conventional pushing and popping, but *should not cause more cache misses*: programs will still go 'up and down' over the same tiny set of (noncontiguous) frames, and even a small cache should be able to hold these frames along with other frequently used data.

Measurements of SML/NJ show that most frames are smaller than eight words (see also Figure 5); we don't load frames down with lots of useless overhead. When larger frames are needed, our 'stack' simulation simply links together enough 8-word (32-byte) frames. Aggregate objects (arrays, records) are never kept in frames.

Free-list handling costs six instructions more than stack-pointer incrementing, so we subtract this cost when presenting results of the simulation (Figure 7).

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 a free list of frames, there is a considerable cost to allocate and deallocate a frame:

1. Test the freelist register.[9]

[9] 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.

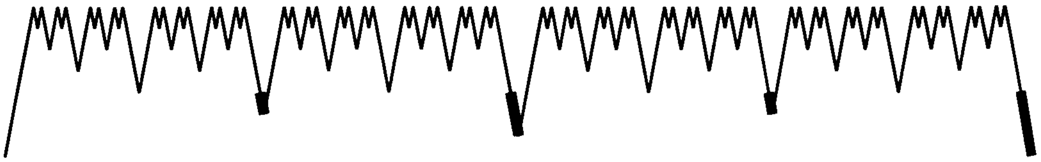

Fig. 8. Execution of $T(7)$ in a 16-line cache. Every uptick (procedure call) is a write miss; only the bold downticks (procedure returns) are read misses.

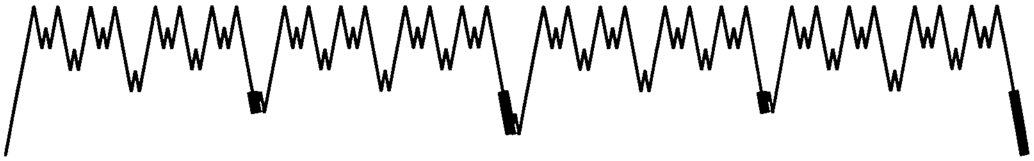

Fig. 9. Execution of $T'(6)$ in a 16-line cache. Only the bold downticks (procedure returns) are read misses.

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 7 shows the run times (after adjustment) of several benchmarks using *Heap* and *Stack* frames, running in simulated caches of different sizes. We simulated a write-allocate cache with partial fill, and also a write-around cache.

Jouppi (1993) simulated both kinds of cache for C programs without garbage collection; Diwan *et al.* (1994) simulated both caches for (almost) purely heap-allocating ML programs. By simulating both caches on stack *and* heap allocation for the same programs, we can compare more straightforwardly.

The results are not too surprising: write-allocate is better on all programs than write-around; and heap allocation is more sensitive to the cache policy than is stack allocation.

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 heap and stack *in a write-allocate cache* 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.)

The simulation measurements (Figure 7) show a cache-read-miss cost (for 16k write-allocate cache) of 1.0 cycles per frame. We calculate this by averaging

$$((D_{16k,heap} - D_{2M,heap}) - (D_{16k,stack} - D_{2M,stack})) * P/F$$

over all the benchmarks, where $D_{c,x}$ is the number of read misses in data cache size c with frame strategy x, $P = 10$ is the miss penalty, and F is the number of frames created (taken from Figure 4).

A nonzero write miss penalty is usually found only on processors that *fetch on write*. Such processors will then allocate the line in the cache, so the average read miss cost will be low (as described in the previous paragraph). For a 10-cycle miss penalty, the write-miss cost per frame (as explained in section 6.1) should be about 5.3 cycles. When the cost of read misses for a write-allocate cache is included, the total cost of read+write misses is 6.3 cycles per frame.

Write-around caches do not need to stall the processor on a write miss, at least for the well behaved sequential writes performed by a heap allocator. Thus, the write miss cost will be zero; but since (almost) every newly written cache line will soon be fetched (Stefanovic and Moss, 1994; Reinhold, 1994), we should expect the read-miss cost per frame for write-around caches to be similar to the cost per frame for fetch-on-write caches. For a 16k write-around cache, the cost of read misses (calculated from the simulations) is 5.1 cycles per frame – somewhat smaller than the 6.3-cycle read+write miss cost for fetch-on-write caches. We have not shown write-around caches in Figure 1, but their total cost will be similar (though slightly smaller, for heap frames) to that of fetch-on-write caches.

Clearly, the small size of frames in SML/NJ is important in achieving good performance, especially for fetch-on-write or write-around caches.

6.3 Read misses: analytically

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 – in a write-validate cache – 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 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 8 and 9.

We claim that these patterns represent 'both extremes and the middle' of all procedure-call patterns. Furthermore, the Towers of Hanoi should be a *worst case* for heaps (as compared to stacks) – because the stack implementation has excellent locality – so its analysis will be instructive.

We assume that the cache is direct-mapped[10] 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). 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 analyse: 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 analyse its cache behaviour 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 stack 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.

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 8 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 two misses in each cluster.

Figure 9 shows a more traditional Towers of Hanoi, that calls `print("move disk d")` at each step.

In a 16k cache – which can hold 512 thirty-two-byte frames – the stack version will have one miss every 2^{512} calls, but the heap version will have one miss every 512 calls. Assuming that the primary cache miss (with secondary cache hit) costs 10 cycles, this is a cost of *one cycle every 50 calls.*

The analytical prediction (0.02 cycles/call) does not completely agree with the simulation (1.0). We believe this is because the simulation cannot directly measure the cache miss cost, because the 'stack' and 'heap' programs do not execute the same instructions, or even the same number of instructions. Instead, we measure the

[10] The results *for these three simple programs* would be the same for set-associative caches with LRU replacement in each set.

total cost of two different implementations (with different frame creation sequences and frame disposal sequences, different garbage collection times, garbage collectors trashing the cache at different frame layout, etc.) and attempt to subtract out the components not related to cache behavior. Small errors in the estimation of any of these components will be additive.

On the other hand, the simulation can capture the effect of 'real' programs that cannot be analysed in closed form like our three paradigmatic examples. Such programs will have interference effects from old objects and nonframe objects. However, Reinhold and Moss (1994) find that such interference does not much affect the cache behavior of recently allocated objects (such as, in our case, frames).

Therefore, we cannot say with confidence whether the simulation or the analytical prediction more accurately characterizes the cache behaviour. To be conservative, we use the 'worse' number (1.0) for Cache read misses in Figure 1.

But now consider what happens in a write-no-allocate (e.g. write-around) cache. 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 five cycles per frame (assuming the frame size is roughly half the cache line size, and a 10-cycle miss penalty).

Future cache designs

What cache write-miss policies can we expect in the future? We must assume that hardware designs of 'commodity' microprocessors will be driven by the SPEC benchmark suite, not by arguments about what's best for functional programs or garbage collection.

Jouppi's measurements of C and Fortran programs (1993) may perhaps be influential. He concludes that *write-validate* (that is, *write-allocate, no-fetch-on-write*) is the policy with best performance. This is exactly the policy that we and others (Diwan *et al.*, 1994; Reinhold, 1994; Stefanovic and Moss, 1994) find best for garbage-collected strict functional programs.

On the other hand, as Jouppi points out, write-validate is difficult – though not impossible – to implement on a shared-memory multiprocessor with cache coherence. Such machines require each writable cache line to have a single owner. Since manufacturers will wish to make multiprocessor-compatible CPU chips, and won't wish to have two different primary-cache designs, this could mean that write-validate will not be common on multiprocessors *or* uniprocessors.

It is safe to expect machines of the late 1990's to have nonblocking read instructions, allowing the "garbage prefetch" technique to give good results even on machines with write-around caches.

7 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'.

What is the garbage collection cost of heap-allocated frames? There are three components:

1. Live heap frames must be copied to an older generation and then scanned, whereas live stack frames need only be scanned.
2. Allocating frames causes more frequent garbage collection, leading to the premature promotion of non-frame objects that might have died if given just a little more time.
3. More frequent collections means more frequent executions of the garbage collector's entry-exit sequence.

We will analyse these costs separately.

Copying frames

We will analyse 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 128 kbytes, frame size of 32 bytes, $G = 4096$). 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).[11] The amortized cost per call is $c \log(G)/G$, or about 0.06 instruction per call.
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 less than the size of the secondary cache (Zorn, 1991; Reppy, 1993). The secondary cache misses (which will occur for stacks or heaps) will probably be just as important as the garbage-collection overhead.

Summary: 0.06 instructions per frame.

More frequent collections

With stack-allocated frames, N garbage collections of average cost $E + L$ will occur, where E is the entry-exit overhead of the collector and L is the cost of copying non-frame live data.

[11] 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.

With heap-allocated frames, N' collections of average cost $E + L' + F$ will occur, where F is the the cost of copying frames. We have accounted for $N' \cdot F$ in the previous subsection. We will account for $N'L' - NL$ in the the next subsection.

Here we calculate $E(N' - N)$. $N' - N$ is simply the number of frames created divided by G, the number of frames the youngest generation can hold. Thus the cost *per frame* is just E/G; with $G = 4096$ and assuming $E = 800$ instructions, this is 0.2 instructions per frame.

Premature promotion of non-frames

In principle, the heap allocation of frames should cause more frequent collection (and undesirable promotion) of the non-frame data. We use lifetime statistics as reported by Stefanovic and Moss (1994) to find those objects that survive G calls but not $p \cdot G$. (The proportion $p = 0.57$ is the proportion of frames to total heap allocation (see Figure 5).) Because of the shape of the object-survival curve, such objects are 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.

Sum of the collection costs

The three components (0.06, 0.2, 0.1) sum to 0.36 instructions per frame attributable to garbage collection.

Direct measurement of garbage collection

To support our analytical calculations of garbage-collection overhead, we measured the garbage collection time for stack vs. heap frames on benchmark programs. Figure 10 shows garbage collection costs for the execution of the benchmark programs on a DEC 5000/240 computer.

For each benchmark, three versions of the program were run: *Heap* (heap-allocated frames); *Stack* ('stack-allocated' frames, implemented as a free-list of re-usable frames, which have a different frame layout and choice of closures); and *Q-Heap* (heap-allocated frames that have exactly the same frame layout (and padding to eight words) as the *Stack* frames).

The mutator time μ and garbage-collection time γ are shown for each benchmark. Times were calculated by executing each benchmark command five times consecutively (from within the same Unix execution) and dividing by five. Ten runs of each such test were made, and the fastest taken (as recommended, for example, by the SPEC benchmark consortium (System Performance Evaluation Corp., 1989)).

Time spent in the operating system is not shown, but was small in all cases (and did not much differ among the three versions of each program).

	Heap		Q-Heap		Stack		MIPS	G.C.Instrs/Frame	
	μ_h	γ_h	μ_q	γ_q	μ_s	γ_s	ρ	Z_h	Z_q
Boyer	1.11	0.86	1.18	0.87	1.17	0.88	31	-0.95	-0.40
Knuth-B	5.96	0.86	6.31	0.93	6.56	0.88	31	-0.20	0.19
Lexgen	9.45	0.60	9.56	0.72	9.77	0.59	31	0.03	0.99
Life	1.24	0.04	1.29	0.06	1.28	0.03	38	2.62	2.20
Yacc	3.02	0.92	3.28	1.18	3.21	0.60	29	7.86	8.85
Simple	15.30	0.54	14.42	0.65	15.29	0.55	30	-0.03	0.33
VLIW	11.29	0.54	15.39	0.57	15.05	0.51	24	0.22	0.25
Average								1.36	1.77

Fig. 10. Garbage collection cost. Mutator time μ and garbage-collection time γ are shown (in seconds) for each benchmark. MIPS (ρ) and garbage-collection overhead per frame (Z_h and Z_q) are calculated as shown in the accompanying text.

We calculated ρ, the effective MIPS (millions of instructions per seconds) for the DEC 5000/240 on each program, by dividing the instructions executed for the heap version of each program (taken from Figure 4) by $\mu_h + \gamma_h$. The peak performance of this machine is 40 MIPS.

To calculate the extra garbage-collection cost attributable to heap-allocated frames, we compared stack g.c. time from heap g.c. time, converted from seconds to instructions, and divided by the number of frames F taken from Figure 5:

$$Z_h = (\gamma_h - \gamma_s)\rho/F.$$

In many cases this is negative! This indicates that any garbage-collection overhead of heap-allocated frames is less important than improved closure layouts.

We then tried an alternate method of calculation. Since *Q-Heap* and *Stack* use exactly the same frame layout, the *only* difference is the failure of *Q-Heap* to free its frames. Thus, the garbage-collection overhead can be more consistently isolated. However, *Q-Heap* frames are all artificially padded to eight words. This will overestimate the load on the collector; we expect any added load to be (roughly) proportional to the total size of all heap-allocated frames. Therefore, in our estimate of the overhead Z we will multiply by the proportion of the frames that are not just padding:

$$Z_q = (U/8)(\gamma_q - \gamma_s)\rho/F,$$

where U is the average frame size of each benchmark, taken from Figure 5.

The Yacc benchmark is anomalous in showing a very high cost, in extra garbage collection, for heap-allocated frames. Closer examination of the Yacc execution showed that there were three major-generation collections with heap frames, but only two with stack frames.

Excluding Yacc, the average Z_h is 0.28 instructions/frame, close to the analytically predicted value of 0.36. Yacc must do something not foreseen by our analytical methods; or our simple two-generation collector is at fault, and a modern multi-generation collector would do better on Yacc.

In Figure 1 we show the measured value of $Z_h = 1.4$ instructions for disposal of heap-allocated frames.

8 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 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 (Wilson, 1991). The simplest way to do this would be to test for the mark on every return, but this would be 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.[12]

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 new 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

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

recursions) the cost will be higher, but the stack-based systems and heap-based systems will pay approximately the same price.

Doligez and Gonthier (1994) 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.

8.1 Updating activation records

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 (Steele, 1978), Orbit (Kranz *et al.*, 1986), and SML/NJ (Appel and Jim, 1989)) naturally initialise 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 (Appel and Shao, 1992; Appel, 1992; Shao and Appel, 1994) 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 9) 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.

9 First-class continuations

The notion of 'first class continuations' using the *call-with-current-continuation* (call/cc) primitive originated in the Scheme language (Rees and Clinger, 1986) and has since been adopted in other systems as well (Duba *et al.*, 1991). First class continuations are useful for implementing coroutines (Wand, 1980), concurrency libraries (Reppy, 1991) and multitasking.

But call/cc is hard 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 unacceptably slow 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 initialisation), call/cc is trivial, and 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 (Clinger *et al.*, 1988; Hieb *et al.*, 1990). 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 complicated to implement.

Stack chunks require a stack-overflow test on every frame[13], so creation costs three instructions (add to SP, compare, branch).

Danvy (1987) made a free list of re-usable frames (we call this a ‘quasi-stack’); these reduce the load on the garbage collector and have good locality; but they are expensive to create and destroy, and require a frame pointer. The *‘stack’* implementation that we have implemented and measured is actually a simplification of Danvy’s method. For applications using first-class continuations (call/cc) our simplification would need an extra mechanism to copy part of the continuation, whereas Danvy’s method does not.

Both methods suffer from the same ‘copying and sharing’ penalty as ordinary stacks. Their performance is summarized in Figure 1, and does not appear competitive (especially given the implementation complexity).

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

10 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:

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 minimise the number of distinct frames written. Shao and Appel (1994) describe an implementation of such an algorithm, which is not particularly hairy.
2. To avoid having a descriptor in each frame, the runtime system can maintain a mapping of return addresses to frame layout descriptors. Kranz’s ORBIT compiler used this technique (Kranz, 1987). *Standard ML of New Jersey* does not bother, so it does indeed pay the price of a descriptor in each frame.

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

[13] ‘Unfortunately, it has been our experience that memory exceptions are not a tenable means for detecting stack overflow....’ (Hieb *et al.*, 1990)

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 complicated scheme to determine when they must be popped (Hanson, 1990). (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 (Hieb *et al.*, 1990).
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.[14]
7. Stack-overflow detection must be implemented. In most cases this is handled automatically by the operating system using virtual-memory page faults.[15]

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.

11 Conclusion

Heap allocation of activation records is simple and competitively efficient. The conclusion that heap allocation is about as cheap as stack allocation, when all effects including cache locality are counted, certainly contravenes the conventional wisdom.

Heap frames are much easier to implement correctly: it is tricky to make stacks 'safe for space complexity', or to support generation garbage collection efficiently, or first-class continuations (call/cc). In the *Standard ML of New Jersey* compiler, which supports all of these features, 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 (stack chunks, quasi-stacks) 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 different: heap-frame handling is about twice as expensive as stack-frame handling (about 7% penalty in overall performance), except for first-class continuations.

Finally, for languages without closures (nested first-class functions with static

[14] In contrast, one heap-allocation region is necessary *per processor*, not per thread.
[15] Heap overflow detection must also be implemented, but this is true whether or not there is a stack.

scope), there is no 'copying and sharing' cost. In this case stacks have a 6% overall performance advantage. (Without closures, call/cc is not an issue, of course.)

Acknowledgements

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

References

Allen, M. S. and Becker, M. C. (1993) Multiprocessing aspects of the PowerPC 601. In: *IEEE COMPCON*, pp. 117–126, February. IEEE Press.

Appel, A. W. (1987) Garbage collection can be faster than stack allocation. *Infor. Process. Lett.*, **25**(4): 275–279.

Appel, A. W. (1989) Simple generational garbage collection and fast allocation. *Software—Practice and Experience*, **19**(2): 171–183.

Appel, A. W. (1992) *Compiling with Continuations*. Cambridge University Press.

Appel, A. W. (1994) Emulating write-allocate on a no-write-allocate cache. *Technical Report CS-TR-459-94*, Princeton University.

Appel, A. W. and Trevor, J. (1989) Continuation-passing, closure-passing style. In: *16th ACM Symp. on Principles of Programming Languages*, pp. 293–302. ACM Press.

Appel, A. W. and Kai Li. (1991) Virtual memory primitives for user programs. In: *4th Int. Conf. Architectural Support for Programming Languages and Operating Systems* (*SIGPLAN Notices* **26**(4): 96–107). ACM Press.

Appel, A. W. and MacQueen, D. B. (1991) Standard ML of New Jersey. In: Wirsing, M., editor, *3rd Int. Symp. on Programming Language Implementation and Logic Programming*, pp. 1–13. Springer-Verlag.

Appel, A. W. and Shao, Z. (1992) Callee-save registers in continuation-passing style. *Lisp and Symbolic Computation*, **5**: 189–219.

Asprey, T., Averill, G. S., DeLano, E., Mason, R., Weiner, B. and Yetter, J. (1993) Performance features of the PA7100 microprocessor. *IEEE Micro*, **13**(3).

Augustsson, L. (1989) Garbage collection in the $< v, g >$-machine. *Technical Report PMG memo 73*, Department of Computer Sciences, Chalmers University of Technology.

Baker, H. G. (1976) The buried binding and stale binding problems of LISP 1.5. Unpublished paper.

Cardelli, L. (1984) Compiling a functional language. In: *Symposium on LISP and Functional Programming*, pp. 208–217. ACM Press.

Chase, D. R. (1988) Safety considerations for storage allocation optimizations. In: *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pp. 1–9. ACM Press.

Clinger, W. D., Hartheimer, A. H. and Ost, E. M. (1988) Implementation strategies for continuations. In: *ACM Conf. on Lisp and Functional Programming*, pp. 124–131. ACM Press.

Danvy, O. (1987) Memory allocation and higher-order functions. In: *Proc. SIGPLAN'87 Symp. on Interpreters and Interpretive Techniques*, pp. 241–252. ACM Press.

Digital Equipment Corp. (1992) *DECchip(tm) 21064-AA Microprocessor Hardware Reference Manual*. First edition. DEC, Maynard, MA.

Diwan, A., Tarditi, D. and Moss, E. (1994) Memory subsystem performance of programs using copying garbage collection. In: *Proc. 21st Ann. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pp. 1–14. ACM Press.

Doligez, D. and Gonthier, G. (1994) Re: stack scanning for generational g.c. Email message `<9403041606.AA07877@lix.polytechnique.fr>`.

Duba, B., Harper, R. and MacQueen, D. (1991) Typing first-class continuations in ML. In: *18th Ann. ACM Symp. on Principles of Programming Languages*, pp. 163–173. ACM Press.

Hanson, D. R. (1980) A portable storage management system for the Icon programming language. *Software—Practice and Experience*, **10**: 489–500.

Hanson, C. (1990) Efficient stack allocation for tail-recursive languages. In: *ACM Conf. on Lisp and Fucntional Programming*, pp. 106–118. ACM Press.

Hardell, W. R., Hicks, A. A., Howell, L. C., Maule, W. E., Montoye, R. and Tuttle, D. P. (1990) Data cache and storage control units. In: *IBM RISC System/6000 Technology*, pp. 44–50. IBM.

Hieb, R., Dybvig, R. K. and Bruggeman, C. (1990) Representing control in the presence of first-class continuations. In: *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pp. 66–77. ACM Press.

Hill, M. D. (1988) A case for direct-mapped caches. *IEEE Computer*, **21**(12): 25–40.

Jones, R. (1992) Tail recursion without space leaks. *J. Functional Programming*, **2**(1): 73–79.

Jouppi, N. P. (1993) Cache write policies and performance. In: *Proc. 20th Ann. Int. Symposium on Computer Architecture*, pp. 191–201. ACM Press.

Kranz, D., Kelsey, R., Rees, J., Hudak, P., Philbin, J. and Adams, N. (1986) ORBIT: An optimizing compiler for Scheme. *SIGPLAN Notices (Proc. Sigplan '86 Symp. on Compiler Construction)*, **21**(7): 219–233.

Kranz, D. (1987) *ORBIT: An optimizing compiler for Scheme*. PhD thesis, Yale University.

Leroy, X. (1992) Unboxed objects and polymorphic typing. In: *19th Ann. ACM Symp. on Principles of Programming Languages*, pp. 177–188. ACM Press.

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

Reinhold, M. B. (1994) Cache performance of garbage-collected programs. In: *Proc. SIGPLAN Symp. on Programming Language Design and Implementation*, pp. 206–217. ACM Press.

Reppy, J. H. (1991) CML: A higher-order concurrent language. In: *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pp. 293–305. ACM Press.

Reppy, J. H. (1993) A high-performance garbage collector for Standard ML. *Technical memorandum*, AT&T Bell Laboratories, Murray Hill, NJ.

Runciman, C. and Wakeling, D. (1993) Heap profiling of lazy functional programs. *J. Functional Programming*, **3**(2): 217–246.

Shao, Z. and Appel, A. W. (1994) Space-efficient closure representations. In: *Proc. ACM Conf. on Lisp and Functional Programming*, pp. 150–161. ACM Press.

Steele, G. L. (1978) Rabbit: a compiler for Scheme. *Technical Report AI-TR-474*, MIT, Cambridge, MA.

Stefanovic, D. and Moss, J. E. B. (1994) Characterization of object behaviour in Standard ML of New Jersey. In: *Proc. ACM Conf. on Lisp and Functional Programming*, pp. 43–54. ACM Press.

System Performance Evaluation Corp. (1989) *SPEC Benchmark Suite Release 1.0*. October.

Ungar, D. M. (1986) *The Design and Evaluation of a High Performance Smalltalk System*. MIT Press, Cambridge, MA.

Wand, M. (1980) Continuation-based multiprocessing. In: *Conf. Record of the 1980 Lisp Conf.*, pp. 19–28. ACM Press.

Wilson, P. R., Lam, M. S. and Moher, T. G. (1992) Caching considerations for generational garbage collection. In: *ACM Conf. on Lisp and Functional Programming*, pp. 32–42. ACM Press.

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

Zorn, B. (1991) The effect of garbage collection on cache performance. *Technical Report CU-CS-528-91*, University of Colorado, Boulder, CO.