Generational Stack Collection and Profile-Driven Pretenuring

Perry Cheng   Robert Harper   Peter Lee

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213-3891

Abstract

This paper presents two techniques for improving garbage collection performance: generational stack collection and profile-driven pretenuring. The first is applicable to stack-based implementations of functional languages while the second is useful for any generational collector. We have implemented both techniques in a generational collector used by the TIL compiler (Tarditi, Morrisett, Cheng, Stone, Harper, and Lee 1996), and have observed decreases in garbage collection times of as much as 70% and 30%, respectively.

Functional languages encourage the use of recursion which can lead to a long chain of activation records. When a collection occurs, these activation records must be scanned for roots. We show that scanning many activation records can take so long as to become the dominant cost of garbage collection. However, most deep stacks unwind very infrequently, so most of the root information obtained from the stack remains unchanged across successive garbage collections. *Generational stack collection* greatly reduces the stack scan cost by reusing information from previous scans.

Generational techniques have been successful in reducing the cost of garbage collection (Ungar 1984). Various complex heap arrangements and tenuring policies have been proposed to increase the effectiveness of generational techniques by reducing the cost and frequency of scanning and copying. In contrast, we show that by using profile information to make lifetime predictions, *pretenuring* can avoid copying data altogether. In essence, this technique uses a refinement of the generational hypothesis (most data die young) with a locality principle concerning the age of data: most allocations sites produce data that immediately dies, while a few allocation sites consistently produce data that survives many collections.

The primary author may be contacted at pscheng@cs.cmu.edu.

This research was sponsored in part by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software," ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. Government.

1 Introduction

Garbage collection is a technique for automatic memory management whereby the programmer is freed from explicit deallocation of heap storage (McCarthy 1960; Knuth 1969; Wilson 1994). Copying garbage collectors reclaim space in two steps: scanning the stack for roots and then copying data reachable from these roots into an unused area of memory. The area vacated by the live data is known to contain only garbage and may be reclaimed. A simple kind of copying garbage collector is the semispace collector (Fenichel and Yochelson 1969) using Cheney's algorithm (Cheney 1970). Unfortunately, semispace collectors cannot usually attain efficient memory usage and good performance.(Ungar 1984) Using the observation that most objects die quickly (Ungar 1984), generational collectors can arrange heap areas and schedule collections to improve performance.

Generational collection successfully reduces the cost of copying data. However, for programs with deep call chains, the cost of scanning the stack for roots can be high. In our study, we observe that root processing can take up to 70% of the total garbage collection cost. Since most deep stacks are not frequently unwound (Table 2), most of the old stack frames are unchanged across successive collections. If we can determine which stack frames are unchanged, then the cost of root scanning can be reduced by reusing the information from the previous collection. This technique, called *generational stack collection*, is like generational garbage collection in that old stack frames are "tenured" to reduce processing frequency.

Generational techniques work by dividing the heap into different regions called generations. Objects that survive initial minor collections of the nursery (the first generation) are more likely to survive many more collections. These objects are promoted into areas that are less frequently collected. The advantage is that if the collections of the older areas are sufficiently delayed, then a large fraction of these objects will have died, making the collection worthwhile. However, long-lived objects are typically copied several times before they are tenured. Multiple generations can make the tenuring prediction more accurate but could cause even more copying of the data that survives. An alternative approach to using runtime per-object predictions is to classify objects based on their allocation site and use profile results to predict lifetimes. This technique can yield information concerning the predicted lifetime of objects before the final execution. Its success relies on the information returned by *heap profiling*. If, as we later show, an allocation site is a good predictor for the age of an object, then tenuring policies can be effectively based on allocation sites. Our empirical data shows significant reductions in data copying.

Section 2 gives background material and some details on the implementation. Section 3 presents the initial empirical results from using the standard techniques. Section 4 describes generational stack scanning with performance results. Section 5 motivates and describes pretenuring with performance results. Section 6 explores extensions and improvements to generational stack collection and pretenuring. Section 7 presents related work and section 8 summarizes the results and points to future directions.

2 Background

In order to make sensible empirical comparisons, we have implemented two "baseline" garbage collectors, a semispace collector and a generational collector. Some background material on these baseline collectors is presented in this section. Next, the representation of the data and stack that arises from the TIL compiler is discussed. The explanation of tracing the stack will reinforce the notion that scanning the stack is relatively costly as a result of the optimizing technology. Finally, the hardware, the benchmarks, and the measurement techniques are given.

2.1 Semispace and Generational Collection

Our semispace collector (Fenichel and Yochelson 1969) uses Cheney's copy algorithm (Cheney 1970). The resizing strategy is parameterized over a target liveness ratio r. In particular, if the liveness ratio after a collection was r', then the heap is resized by the factor r'/r. In our tests, a target ratio value of $r = 0.10$ was used.

The generational collector we use is similar to the one used in (Ungar 1984). There are many parameters that can be explored with generational collection. For simplicity, our collector has two generations, a nursery and a tenured generation. The heap resizing policy for the tenured generation is based on deviations from a preset target liveness ratio of 0.3. The nursery is never made larger than the secondary cache (512K for our setup), following the advice of several researchers including Diwan and Tarditi (Tarditi and Diwan 1994). For benchmarking reasons, the nursery is sometimes made significantly smaller.

At each minor collection, we immediately promote all live objects from the nursery to the tenured generation. Large arrays are not allocated in the nursery and promoted to the tenured area; instead, they reside in a region managed by a mark-sweep algorithm. Finally, we use a simple write barrier technique, a sequential store buffer (Appel 1989), to handle pointer updates that may create intergenerational references.*

2.2 TIL

TIL (Tarditi, Morrisett, Cheng, Stone, Harper, and Lee 1996) is an optimizing compiler for Standard ML (SML) that exploits several key technologies: intensional polymorphism, nearly tag-free garbage collection, conventional functional language optimizations, and loop optimizations. With these technologies, it is usually possible to determine the type and therefore representation of values. Unlike traditional functional language compilers that use a universal data representation to implement polymorphism, TIL does not tag integers nor does it always box† floating-point values. In particular, TIL performs sufficient type analysis to provide tag-free unboxed word-sized integers and aligned unboxed floating-point arrays. At runtime, heap-allocated objects are represented by records, pointer arrays, and non-pointer arrays. The intensional polymorphism optimizations improve runtime performance of monomorphic code at the expense of slower polymorphic code that passes types at runtime. Further, the garbage collector's interaction with the program is also more complex. TIL also uses a stack to manage activation records rather than using heap-allocated frames. Again, we are trading the performance benefits of a stack against greater complexity in the garbage collector.

The entire runtime system consists of about 7000 lines of C and assembly code. Of that, 3500 lines comprise the garbage collector (including all of the techniques that were examined) and another 1500 lines provide support code like pretty-printing, heap profiling, and debugging. The rest of the code provides support for I/O, signals, and internally used data structures.

2.3 Stack Scanning

At any execution point, data is live if it is accessed as the program continues to execute. A conservative estimate of accessible data is reachable data. (Some reachable data may be unnecessary for program completion.) Thus, a collector only needs to retain data that is accessible by following all pointers starting from root values. Roots are directly accessible values such as registers and stack slots.

The difficulty with accurately determining the root set stems from the presence of callee-save registers and polymorphism (Tarditi, Morrisett, Cheng, Stone, Harper, and Lee 1996). First, with callee-save registers, the contents of a register or stack slot may come from caller frames so stack frames cannot be decoded in isolation. Instead, the stack scan must start from the initial frame and maintain the pointer status of the register set as the scan progresses. Second, since polymorphism introduces variable types, the compiler cannot statically compute whether a particular value is a pointer or not. As a result, the compiler must sometimes communicate to the collector the correspondence between values and their dynamic types.

When the garbage collector is called by the mutator, the return address indicates the current execution point of the mutator. By indexing into a table of auxiliary information (called a trace table) with this return address, we can determine the frame layout of the GC-caller frame. Then, from the return address of the GC-caller frame, we can decode its caller frame. Continuing in this way, we finally reach the initial frame. From the initial frame, we begin determining the root set by adding registers and stack slots as we traverse the stack downwards again. Note that the stack scan is two-pass as a result of callee-save registers. This complex scheme of decoding stack frames in the runtime system relieves the mutator from always having to tag stack slots even for stack frames that may not survive to a garbage collection.

*If a reference from an older generation is created to a younger generation with a pointer update, then simply collecting the younger generation could lead to a dangling pointer. However, the mutator can store all pointer updates into a table which is used by the runtime system at each garbage collection. For non-pointer updates, no special action is needed.

†A boxed object is stored in memory and represented by a pointer.

Program	lines	Description
Checksum	241	Checksum fragment from the Foxnet (Biagioni, Harper, Lee, and Milnes 1994), 16Kb possibly unaligned arrays are created and checksummed using iterators 10,000 times
FFT	246	Fast Fourier transform, multiplying polynomials up to degree 65,536
Color	110	Brute-force graph coloring
Gröbner	904	Compute Grobner basis of a set of polynomials up to degree 7 (Yan 1996)
Knuth-Bendix	618	An implementation of the Knuth-Bendix completion algorithm
Lexgen	1123	A lexical-analyzer generator (Appel, Mattson, and Tarditi 1989), processing the lexical description of Standard ML
Life	146	The game of Life implemented using lists (Reade 1989)
Peg	458	Solving a peg-jumping game, using the output of a Prolog to ML translator(Hornof 1992)
Nqueen	73	The N-queens problem for n=10
PIA	2065	The Perspective Inversion Algorithm (Waugh, McAndrew, and Michaelson 1990) deciding the location of an object in a perspective video image
Simple	870	A spherical fluid-dynamics program (Ekanadham and Arvind 1987), run for 4 iterations with grid size of 200.

Table 1: Benchmark programs

Program	Total Alloc	Max Live Data	Records Alloc	Arrays Alloc	Max(Avg) Frames in Stack	New Frames in Stack	Number of Pointer Updates
Checksum	1407MB	16KB	1094MB	0.0MB	4(4.0)	4.0	33
Color	98MB	24KB	98MB	0.1MB	482(469.7)	61.2	1215
FFT	93MB	2073KB	0.12MB	81MB	5(4.2)	4.2	27
Gröbner	139MB	128KB	139MB	0.1MB	106(16.5)	16.5	512
Knuth-Bendix	344MB	16098KB	342MB	2.1MB	4234(1336.5)	116.9	29
Lexgen	117MB	3524KB	88MB	2.2MB	1802(714.3)	435.6	213
Life	363MB	24KB	359MB	0.0MB	51(6.2)	6.2	29
Nqueen	88MB	3600KB	87MB	0.3MB	29(22.4)	22.4	193
Peg	241MB	24KB	231MB	10MB	26(19.3)	19.3	2974688
PIA	430MB	648KB	154MB	214MB	910(120.7)	112.8	41
Simple	753MB	6864KB	493MB	158MB	243(16.4)	13.4	27

Table 2: Allocation characteristics of benchmarks

But what type of information is available in this trace table? The return address, which serves as a key to the trace table entry, and the stack frame size are necessary. In addition, for each general-purpose register and stack slot, we record its *trace*. There are four types of traces: pointer, non-pointer, callee-save register, or compute. A **pointer** trace indicates that the compiler has statically determined that the value is a pointer and needs to be traced. A **non-pointer** trace indicates that the value is not a root. Registers or stack slots that are marked **callee-save** have additional information in the table that indicates which register the value is saved from. Finally, the **compute** trace is use when the compiler could not statically determine the pointer status of a value. Additional information indicates where the type of such a value resides. From this type, the runtime computes whether the value should be included in the root set or not.

Figure 1 gives an example of a stack frame. The left portion of the diagram gives a stack frame which is described by the corresponding trace table entry on the right. Their correspondence is established by the return address which is always in the first stack slot and indexes into the trace table. From the second table entry, we see that this frame has six slots. The second slot, described by the third table entry, is a non-pointer while the third and fourth slots are both pointers. Note that the fourth slot contains a runtime type which is used to describe the contents of the fifth slot. In this case, since the type is a record, the collector will know that the fifth slot must be traced. Finally, the sixth slot contains the spilled value of register 10. Whether this slot needs to be traced depends on the state of register 10 from the previous stack frame. Finally, the table entry contains trace information on the register set.

3 Hardware, Benchmarks, and Measurement Methods

To evaluate the effectiveness of generational stack markers and pretenuring, we compare four techniques: semispace collection, generational collection, generational collection with stack markers, and generational collection with stack markers and pretenuring. The validity of the comparison is strengthened by the fact that each technique is tested under identical conditions: the same language and compiler, same hardware, and same benchmarks.

The empirical results were gathered on a DEC 3000/500 Alpha workstation which features a 21064 microprocessor. This chip has split instruction and data caches. Both caches are direct-mapped and sized at 8K with cache lines of 32

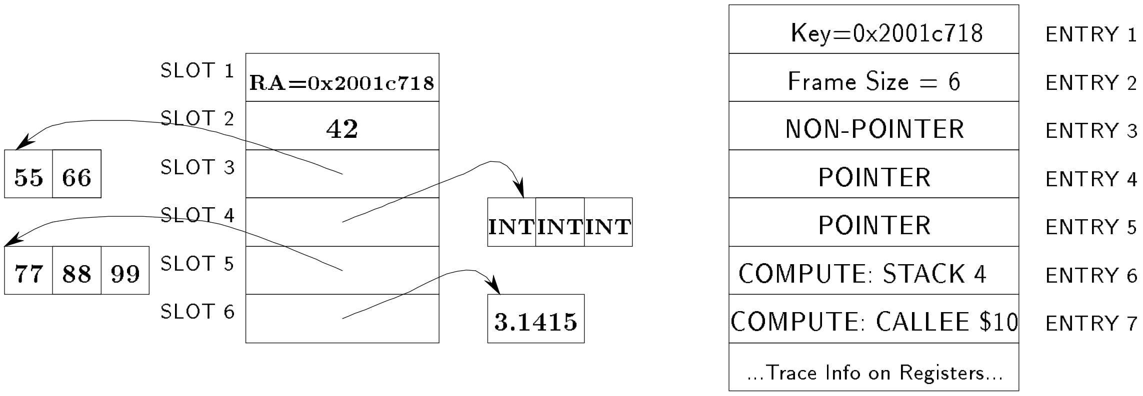

Figure 1: Example of a stack frame (left) and its corresponding table entry (right)

bytes. It uses a write-through policy with a 4 entry write buffer and performs write-around on write misses. This model also has a second-level off-chip 512K cache with a write-allocate policy (DEC 1994b; DEC 1994a). The particular machine used has 96MB of main memory and runs OSF/1 v2.0.

The sampled data is taken from runs of eleven SML (Milner, Tofte, and Harper 1990) programs (many are *de facto* standard benchmarks (Appel 1992)) compiled with TIL. Table 1 describes the benchmark programs, which range in size from 73 lines to about 2000 lines of code. They cover a range of application areas including scientific computing, list-processing, systems programming, and compilers. Larger benchmarks could not be included because the current version of TIL does not yet support the SML module system. Table 2 describes the allocation behaviors of the benchmarks. The "New Frames in Stack" column refers to stack frames encountered by the collector that were not present during the last collection.

Tables 3, 4, 5, and 6 show the time performances of the benchmarks with various collection techniques. In each of the configurations explored, data from ten runs were collected and the arithmetic mean is reported. In all cases, the standard error is under 1%. Times are reported in seconds and measured with UNIX virtual timers. `Total`, `GC`, `Client` denote the total execution time of the program, the time spent in garbage collections, and the time spent in the mutator, respectively. (Of course, `Total` = `GC` + `Client`.) Also, `NumGC` and `Copied` respectively denote the number of garbage collections and the amount of data in bytes copied during all collections.

Since different garbage collection techniques use different amounts of memory (if unconstrained), a direct comparison of the techniques without some control would be unfair. A reasonable way to render the comparison fair would be to limit each technique to a fixed amount of memory. For copying collectors, the absolute minimum amount of memory (`Min`) needed to run would be twice the maximum amount of live data a program has during execution since a garbage collection may occur at any moment. Thus, we choose various multiples (designated k) of this minimal value for each program and compare the performance of various techniques where the collector is permitted $k *$ `Min` memory.

4 Initial results

We consider the initial result obtained by using the two baseline collector techniques. Table 3 shows the time and space performance of the benchmarks under semispace collection with k values of 1.5, 2, and 4. For programs with little long-lived data, the time spent in garbage collection should be inversely proportional to k since the time spent in any one garbage collection is approximately constant. In particular, FFT and Checksum experienced improvements in GC times of 1.7 and 2.4 times as k increases from 1.5 to 4. (Note that an ideal linear improvement would cause a speedup of $\frac{4}{1.5} = 2.7$.) Programs that have long-lived data benefit even more greatly from large values of k since each collection is relatively more expensive for such programs. In particular, Gröbner and Knuth-Bendix experienced improvements of 4.1 and 4.4. Thus, these numbers are in accordance with the observation that FFT and Chksum generate no long-lived data while Gröbner and Knuth-Bendix do. Although there are minor fluctuations, the client time is relatively undisturbed by k. Consequently, changes in total times directly reflect changes in GC times.

Table 4 shows the performance of the benchmarks under generational collection. Most programs' GC times improved by a factor of 2.5 when k is increased from 1.5 to 4. A notable exception is Knuth-Bendix where the time worsened by 5%. In the case of Knuth-Bendix, almost all the data that survives the nursery remains alive to the end of the program. Because there are no major collections, the extra amount of memory given is unused. It is plausible that the time slightly worsens since the runtime system now has the overhead of managing more memory without any client benefit. On the other hand, we note that PIA's GC time improved dramatically from 71s to 4.2s. The 17-fold decrease indicates some interesting behavior. Although the total number of garbage collections remains approximately the same as k varies, the number of major garbage collections is severely affected by k since there is a significant amount of data that is copied on major collection. In particular, PIA exhibits an allocation behavior that is not amenable to generational collection: PIA's tenured data tends to die rapidly. Even multiple generations would not ameliorate the problem because any data that is tenured will most likely die before the next collection.

Even a cursory glance at tables 3 and 4 show that gen-

Program	Total Time (sec) $k = 1.5$	$k = 2.0$	$k = 4.0$	GC Time (sec) $k = 1.5$	$k = 2.0$	$k = 4.0$	Client Time (sec) $k = 1.5$	$k = 2.0$	$k = 4.0$
Chksum	34.16	32.96	31.45	6.83	5.59	4.08	27.33	27.38	27.36
Color	78.96	52.98	30.57	63.05	38.00	15.00	15.91	14.98	15.00
FFT	45.38	44.71	44.32	1.94	1.31	0.82	43.44	43.40	43.50
Gröbner	22.00	15.61	9.90	16.03	9.64	3.87	5.97	5.98	6.03
Knuth-Bendix	57.73	44.69	34.25	30.62	17.55	6.96	27.11	27.14	27.29
Lexgen	29.68	27.00	24.85	8.02	5.06	1.96	21.66	21.94	22.89
Life	30.16	29.16	28.03	3.90	2.90	1.72	26.26	26.26	26.32
Peg	21.42	18.06	13.72	11.52	8.20	3.88	9.90	9.87	9.83
PIA	128.16	87.29	57.48	93.25	51.46	19.94	34.91	35.84	37.54
Simple	80.90	71.09	63.18	24.71	14.85	6.57	56.19	56.25	56.61

Program	Number of GCs $k = 1.5$	$k = 2.0$	$k = 4.0$	Data copied (bytes) $k = 1.5$	$k = 2.0$	$k = 4.0$
Chksum	11742	8795	4388	6294192	4714200	2353408
Color	5579	3422	1302	109582416	64778048	15230424
FFT	59	39	17	63237932	42109916	22159764
Gröbner	1107	656	250	152993208	90205460	34312888
Knuth-Bendix	107	65	26	262692124	148075968	54428764
Lexgen	31	19	7	77696860	46909028	16992816
Life	4070	2987	1459	23594284	15146272	6244820
Peg	3479	2404	1069	88500416	61710880	27084864
PIA	1145	620	222	678858984	362369260	128563500
Simple	149	87	33	815835344	467707256	176086208

Table 3: Time and space usage for semispace collector

erational collection wins. However, there are special cases where semispace collectors can do much better than generational collectors. Some programs allocate data intensively but have almost no long-lived data or have relatively expensive root processing. In these cases, the cost of GC is almost constant so GC cost is solely dependent on collection frequency. With such programs and fixed memory constraints, the size of a semispace is larger than the nursery area of a generational collector. Thus, GC frequency decreases in a semispace collector so the overall cost of GC is lower.

Aside from such special cases, we see that generational collection generally improves GC time dramatically, from 20% to 80%. Moreover, the generational collector has improved cache effects: the allocation area remains permanently in the secondary cache and long-lived data is now grouped together. Most programs experience a client time improvement of 10% to 35%. Only FFT did not have this benefit since the relative lack of garbage collections in this program dampens any possibility of cache interactions resulting from data movement.

We now consider individually the performance of the benchmarks under generational collection to find opportunities for improvement. FFT spends about 0.2% of its time in GC and further optimization is not worthwhile. Chksum spends about 10% of its time in garbage collection but each collection on average takes only about 0.2 milliseconds. Most of the cost here is overhead associated with calling the garbage collector. In this case, the cost of each GC is nearly constant, and the only way to further reduce the cost is by tuning the collector code and increasing the nursery size. The high 32% GC time for Peg is an artifact of the high mutation rate of this program. This is quite apparent from the last column of Table 2 which shows Peg having 4 orders of magnitude more updates than any other benchmark. The simple sequential store list records a mutated site repeatedly, causing a great overhead in root processing. A more realistic approach such as card-marking (Sobalvarro 1988) would probably ameliorate most of the problems.

As for Knuth-Bendix, Color, and Lexgen, all three are characterized by deep stacks and will be analyzed in section 5. Section 6 then explores heap profiling and pretenuring and their applicability to the remaining benchmarks.

5 Generational Stack Collection

We begin by breaking down the GC cost of Knuth-Bendix, Color, and Lexgen in the left column of Table 5. In most programs, the cost of generational garbage collection is usually dominated by the cost of scanning and copying the heap (`GC-copy`) rather than by the cost of root processing (`GC-stack`), since the stack is typically much smaller then the heap. Some functional programs (Knuth-Bendix, Color, and Lexgen), however, are highly non-tail recursive and have a deep stack. For such programs (see column 2 of Table 5), the cost of scanning the stack can be as high as 95% of the cost of garbage collection. However, the stack allocation pattern of such programs typically does not involve having rapid alternation of stack growth and shrinkage. For example, Table 2 indicates that for Knuth-Bendix, only 116.9 of the 1336.5 stack frames that the collector traverses on average have changed since the last collection. Rather, once the stack is deep, most of the ancestral frames tend to remain activated for many garbage collections. We can take advantage of this by not rescanning the part of the stack that did

Program	Total Time (sec)			GC Time (sec)			Client Time (sec)		
	$k = 1.5$	$k = 2.0$	$k = 4.0$	$k = 1.5$	$k = 2.0$	$k = 4.0$	$k = 1.5$	$k = 2.0$	$k = 4.0$
Chksum	30.84	28.40	25.27	7.34	5.49	2.93	23.51	22.91	22.34
Color	56.91	34.21	23.65	13.87	20.63	9.77	15.91	13.59	14.94
FFT	45.28	44.29	43.78	0.07	0.08	0.09	45.21	44.21	43.69
Gröbner	21.24	15.28	6.10	16.06	10.22	1.11	5.18	5.06	4.99
Knuth-Bendix	26.88	26.96	26.90	7.66	8.00	8.07	19.22	18.96	18.83
Lexgen	22.31	21.35	21.43	3.20	2.58	2.43	19.11	18.77	19.00
Life	28.43	27.04	25.73	4.19	2.89	1.42	24.24	24.15	24.31
Nqueen	14.90	15.16	15.20	1.83	1.86	1.95	13.07	13.30	13.24
Peg	21.23	16.83	12.89	12.15	7.82	4.07	9.08	9.01	8.82
PIA	103.39	70.53	36.12	71.04	38.07	4.22	32.35	32.46	31.90
Simple	51.37	50.04	49.65	5.05	4.81	4.33	46.31	45.23	45.32

Program	Number of GCs			Data copied (bytes)			Avg Frame Depth
	$k = 1.5$	$k = 2.0$	$k = 4.0$	$k = 1.5$	$k = 2.0$	$k = 4.0$	
Chksum	29513	21986	10993	1012	1832	536	4.0
Color	4209	2806	1402	22519364	2948588	1120044	469.7
FFT	191	191	191	167284	167284	167284	4.9
Gröbner	1372	1029	512	184864856	115777416	11832036	16.4
Knuth-Bendix	807	807	807	14569800	17869436	17695560	1115.3
Lexgen	213	213	213	27427544	18647632	16435292	714.6
Life	9680	7207	3603	21432324	15449868	7615976	6.0
Nqueen	410	410	410	5312548	5312548	5312548	22.4
Peg	7139	5315	2653	78911996	48196128	23440724	19.3
PIA	1161	1035	1035	665596588	349143644	30575584	120.7
Simple	1644	1644	1644	25771348	25431144	25430284	14.4

Table 4: Time and space usage for generational collector

not change. Instead, we save the information from a previous garbage collection and reuse the parts of the information corresponding to the unchanged part of the stack.

In our generational collector, since objects in the nursery are always promoted, we do not need to consider roots residing in frames that were present in previous collections. Even if our collector were designed so that it must recognize all stack roots at each minor collection, it is still advantageous to have amortized the cost of decoding the stack frames by storing the decoded results, namely the register state and root list.

Maintaining such information is a matter of bookkeeping. The tricky part is knowing how much of the stack has not changed since the last collection. Certainly one can maintain a special slot per stack frame containing an initially unset flag that indicates whether it has been scanned by the collector. On a future collection, if the flag is set, then the collector knows that this frame has not changed. However, this increases the stack frame size and requires extra instructions for every stack frame. Since most frames die before a collection, this technique penalizes programs that do not have deep stack behavior or do not require frequent garbage collections.

To achieve good performance for deep stacks while not penalizing the average program, the technique used must not significantly affect the mutator but instead place the bulk of the cost in garbage collection. This can be achieved with the following addition to the collector: each time we scan the stack, we change the return address of every n^{th} stack frame (n is a parameter best chosen to balance the gains of information reuse against the cost of the bookkeeping) to a special stub function while recording the original return address in a table. When a function returns from such a marked frame, it transfers to the stub function which notes that this particular frame has been deactivated, and then continues at the original return address recorded in the table. Our tests use a value of $n = 25$.

This almost works. Unfortunately, functions do not always return normally. If an exception is raised, the exception handlers are invoked in stack order until there is a matching handler. This matching handler may correspond to a frame that is arbitrarily far up in the stack. In particular, control may jump past many marked frames without the normal return. Consequently, some action must be taken or else the collector will not later know that the intervening frames are stale. Fortunately, it suffices to maintain a value M that is updated at every raised exception so that it contains the shallowest stack pointer value that occurred as a result of raised exceptions. Later the garbage collector simply takes the shallower value of M and the value(s) in the table to obtain the deepest stack frame that is safe to reuse.

An alternative implementation for dealing with exceptions that is consistent with stack-allocated activation records moves the bookkeeping cost from the raising of an exception into the collector by having the collector walk the chain of exception handlers at each collection to determine if any handlers that were raised since the last collection jumped past a marked frame. Deferring the handling of exceptions to a collection is advantageous for programs that frequently raise exceptions. In compilers where the run-

Program	Without stack markers				With stack markers				GC% decreased
	`GC`	`stack`	`copy`	`stack%`	`GC`	`stack`	`copy`	`stack%`	
Chksum	2.93	0.60	2.33	20.61%	2.96	0.83	2.13	28.02%	-1.0%
Color	9.77	2.64	7.13	26.99%	2.51	1.05	1.46	41.77%	74.3%
FFT	0.09	0.01	0.08	13.48%	0.10	0.02	0.07	21.74%	-5.7%
Grobner	1.11	0.06	1.05	5.18%	1.13	0.09	1.04	7.69%	-1.6%
KB	8.07	6.14	1.93	76.09%	2.62	0.70	1.92	26.83%	67.5%
Lexgen	2.43	0.86	1.57	35.37%	2.11	0.53	1.58	25.08%	13.0%
Life	1.42	0.21	1.21	14.63%	1.52	0.30	1.22	19.50%	-7.0%
Nqueen	1.95	1.85	0.11	95.01%	2.03	1.85	0.18	90.99%	-4.0%
Peg	4.07	0.47	3.59	11.64%	4.18	0.56	3.63	13.30%	-2.8%
PIA	4.23	0.49	3.73	11.69%	4.36	0.50	3.87	11.36%	-3.3%
Simple	4.33	0.23	4.11	5.21%	4.18	0.25	3.93	5.98%	-3.6%

Table 5: Breakdown of GC cost at $k = 4$ of generational collection without and with stack markers. All times are measured in seconds.

time system is responsible for unwinding the stack on raised exceptions, remembering which stack frames are no longer active is simple.

Given the two possible implementation strategies, we chose the first one since it does not entail modifying the compiler. (Such modifications tend to make the performance comparisons questionable.) The results are displayed in the rightmost column of Table 5. The GC times are drastically reduced for all three target programs, with relative decreases ranging from 13% to 74%. For completeness, we show the result of the other benchmarks. It is worth noting that the stack markers poses only a very slight cost for the other programs, averaging at 3%.

6 Heap Profiling and Pretenuring

As stated previously, generational collection takes advantage of the widely varying lifetimes of different heap-allocated objects. The traditional tenuring policy promotes an object if it survives one or several collections. This policy has the disadvantage that an object may be copied several times before being placed in the "right" generation. Rather than discovering at runtime whether objects survive through promotion, we could try to use profile data to predict the survival rate without copying. Clearly, it would be infeasible to uniquely identify each object because that would require too much data and the same objects do not recur in different program runs. Instead, we speculate that objects allocated from the same point in the program would tend to have similar lifetimes.

To test this hypothesis, heap profiles were gathered to study the average lifetimes of objects created at different program allocation sites. To obtain heap profiles, the compiler was modified so that, when emitting allocation code, an allocation site identifier is prepended to each allocated object. During a garbage collection, each copied object's allocation site identifier is inspected and the entry corresponding to that site is updated. To gather information about object deaths, we also scan the allocation area after each collection to locate dead objects and update their allocation site entries. This information allows the profiler to compute the number, size, and average age of objects emanating from each allocation site. Profiled programs typically run 50% to 200% slower than their unprofiled versions.

For concreteness, abbreviated outputs of the heap profile generated for the Knuth-Bendix and Nqueen benchmarks are included in Figure 2. Only allocation sites that contribute at least 1% of allocated or copied objects are included. The `alloc%` shows the relative amount of data allocated at each site while the `copied%` shows the relative amount of data allocated at each site that is eventually copied. The `old%` indicates the percentage of data generated at each site that survives at least one minor collection. Two complementary trends are obvious. In Knuth-Bendix, 90% of allocated data is generated by sites whose survival rate, to 4 significant digits, is 0. Conversely, over 96% of data that is copied are allocated from sites whose survival rate is at least 80%. (These sites are marked with `<--` in the table.) The polarity is even more striking for the Nqueen benchmark, where 99% of the copied data are generated from only 4 sites.

From such profiles, we can identify allocation sites that consistently produce long-lived objects and *pretenure* these objects. That is, objects from these sites are directly allocated into the older generation. This is beneficial in that the liveness ratio of the younger generation is decreased, thus lowering the amount of objects copied at each collection and speeding up the collection process. In our tests, we pretenure objects allocated from sites whose `old %` cutoff is at least 80%. Considering the bimodality of the data, this pretenuring policy is relatively insensitive to the particular cutoff chosen.

Unfortunately, simply allocating data into the older area breaks an invariant. An object directly allocated into an older generation may have a reference to an object in a younger generation. Of course, pointer mutations also break this invariant and we could deal with these new intergenerational references in the same way. This write barrier approach would be correct but too slow (Zorn 1989). One might suggest relaxing the tenuring condition for such objects so that they are promoted at every minor collection. Unfortunately, particularly for young generations, survival of even one collection indicates long-lived-ness. Therefore, even allocating them in a young generation and then immediately promoting them will lead to substantially more

The 3% should actually be lower. Its measurability is an artifact of debugging code and timing code. Calling UNIX getrusage to perform time measurements of the stack markers technique distorts the times and most of the 3% increase is attributable to this. Removal of this code will result in a small uniform improvement on all benchmarks.

```
============================= Knuth-Bendix =========================================
          alloc      alloc     alloc               avg   copied  copied  copied size/
site        %         size     count     % old    age     size     %     alloc size
------------------------------------------------------------------------------------
10897    41.07%  113345036  16192148      0.00  1344.6      231    0.00  0.00
10911    17.60%   48576375  16192125      0.00     0.0        0    0.00  0.00
10842    10.14%   27981107   3997301      0.00   941.7       84    0.00  0.00
10764     7.84%   21646982   3092426      0.00  1217.5       84    0.00  0.00
10856     4.34%   11991840   3997280      0.00     0.0        0    0.00  0.00
10789     3.36%    9269365   1324195      0.00   969.6       56    0.00  0.00
10778     3.35%    9259011   3086337      0.00     0.0        0    0.00  0.00
10803     1.44%    3967959   1322653      0.00     0.0        0    0.00  0.00
10891     0.05%     139076     34769     86.60  1205.8   120496    1.01  0.87 <--
10926     0.04%     119142     13238     95.94   521.9   164817    1.39  1.38 <--
10920     0.05%     139076     34769     99.52  1404.0   236996    2.00  1.70 <--
10707     0.21%     593016    148254     99.83  1075.8   972328    8.19  1.64 <--
10921     0.05%     139076     34769     99.33  1399.9   236740    1.99  1.70 <--
10709     0.24%     667024    166756     99.81  1092.7  1096688    9.24  1.64 <--
10701     0.48%    1316872    329218     99.80  1336.8  2255800   19.00  1.71 <--
10711     0.06%     175508     43877     99.58  1294.7   297180    2.50  1.69 <--
10692     0.06%     158640     39660     99.60  1476.0   275420    2.32  1.74 <--
10703     0.46%    1276600    319150     99.79  1297.0  2175008   18.32  1.70 <--
10702     0.46%    1276600    319150     99.79  1297.0  2175008   18.32  1.70 <--
10708     0.24%     667024    166756     99.81  1092.7  1096688    9.24  1.64 <--
10710     0.06%     175508     43877     99.59  1294.8   297200    2.50  1.69 <--
------------- heap profile end : short ----------------------------------------
Showing only entries with alloc % > 1.00
                  or with copy %  > 1.00
21 of 2048 entries displayed.
Using a (% old) cutoff of 80%,
  targeted sites comprise 96.02% copied and  2.48% allocated.
```

```
================================ Nqueens =========================================
          alloc      alloc     alloc               avg   copied  copied  copied size/
site        %         size     count     % old    age     size     %     alloc size
----------------------------------------------------------------------------------
10757    39.98%   11662960   2915740      0.19    38.6    22560    0.62  0.00
10758    20.20%    5893923    841989      0.21    39.5    12404    0.34  0.00
10759    14.43%    4209945    841989      0.21    39.5     8860    0.24  0.00
10750     3.80%    1107600    184600      0.00     0.0        0    0.00  0.00
10751     3.16%     923000    184600      0.00     0.0        0    0.00  0.00
10752     3.16%     923000    184600      0.01    49.9       65    0.00  0.00
10753     3.16%     923000    184600      0.00     0.0        0    0.00  0.00
10755     3.16%     923000    184600      0.01    45.9      120    0.00  0.00
10754     2.53%     738400    184600      0.02    42.2      176    0.00  0.00
10748     2.92%     852000    170400     99.88   129.7  1872295   51.23  2.20 <--
10749     2.34%     681600    170400     99.88   129.7  1497836   40.98  2.20 <--
10756     0.19%      56800     14200     99.88   129.7   124816    3.42  2.20 <--
10764     0.19%      56800     14200     99.88   129.7   124816    3.42  2.20 <--
------------- heap profile end : short ------------------------------------
Showing only entries with alloc % > 1.00
                  or with copy  % > 1.00
13 of 2048 entries displayed.
Using a (% old) cutoff of 80%,
  targeted sites comprise 99.04% copied and  5.65% allocated.
```

`alloc size`	number of bytes allocated from this site
`alloc %`	percentage of bytes allocated from this site
`alloc count`	number of objects allocated from this site
`old %`	percentage of objects that survive the first collection after its creation
`avg age`	average age of objects when they die
`copied size`	number of bytes allocated from this site that were copied during all collections
`copied %`	percentage of bytes allocated from this site that were copied during all collections

Figure 2: Heap Profiler Output for Knuth-Bendix and Nqueens

Program	Total Time (sec)			GC Time (sec)			Client Time (sec)			% Decrease		
	$k = 1.5$	$k = 2.0$	$k = 4.0$	$k = 1.5$	$k = 2.0$	$k = 4.0$	$k = 1.5$	$k = 2.0$	$k = 4.0$	GC	Client	Total
Knuth-Bendix	20.57	20.95	21.12	1.44	1.76	1.88	19.12	19.19	19.24	33%	-2%	2%
Lexgen	21.60	21.07	20.41	2.63	2.00	1.55	18.97	19.08	18.85	27%	1%	4%
Nqueen	14.79	14.97	14.51	13.88	14.03	13.53	0.94	0.90	0.97	50%	2%	5%
Simple	50.28	50.36	47.08	3.58	3.74	3.71	46.70	46.62	43.37	12%	4%	5%

Program	Number of GCs			Data copied (bytes)		
	$k = 1.5$	$k = 2.0$	$k = 4.0$	$k = 1.5$	$k = 2.0$	$k = 4.0$
Knuth-Bendix	779	779	779	2050212	5376156	5151708
Lexgen	206	206	206	24278388	15452696	13397340
Nqueen	387	387	387	194256	194256	194256
Simple	1645	1645	1645	14241500	14734176	14133376

Table 6: Time and space usage for generational collector with pretenuring

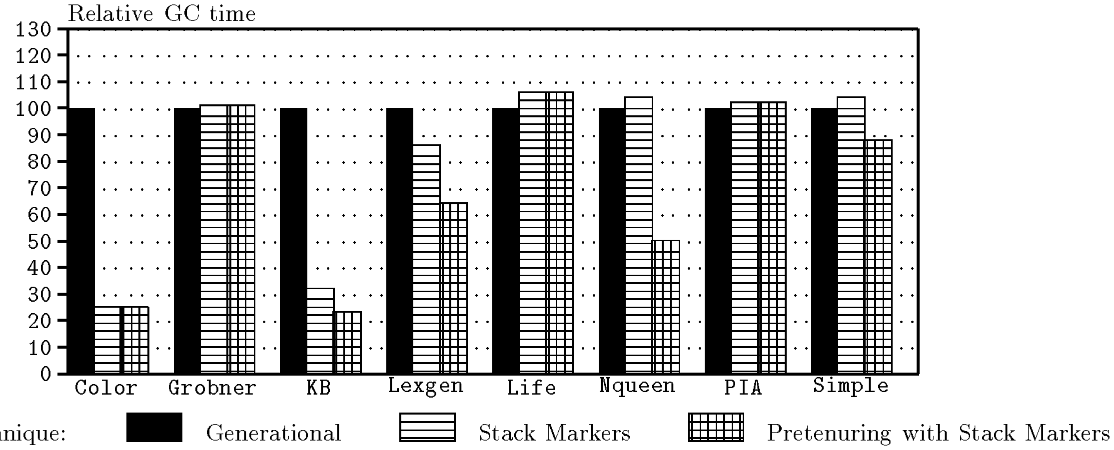

Table 7: Relative GC time at $k = 4.0$

copying than the pretenuring. Instead, we remember the area of the older generation that has been directly allocated into and scan this region with the collector on the next collection. This is a win over copying since copying objects is slower than only scanning them for the (hopefully) few young generation pointers.

From the profiles, it is clear that this optimization would be useful only for four of the benchmarks: Knuth-Bendix, Lexgen, Nqueen, and Simple. Since this optimization is selective based on the heap profile results, the remaining programs cannot suffer from this optimization. Table 6 contains the performance figures under a generational collector with generational stack collection and pretenuring.

The code sequence for allocating objects into the pretenured region is somewhat longer than the normal allocation code sequence, so there is a possibility that the client time might increase as a result of pretenuring. On the other hand, there may be improved cache effects since all the surviving objects are pretenured into a single memory region. Quite likely both of these effects are small and so do not greatly change the client time. In fact, the Client% Decrease column of Table 6 shows that, on average, there is a slight decrease in the client time. More importantly, the addition of pretenuring reduces the GC times of the four benchmarks by 33%, 27%, 50%, and 12%, respectively. The final column indicates the relative decrease in total execution time. Since GC is already taking a relatively small percentage of total execution time (around 10%), Amdahl's Law restricts the magnitude of the decrease in total execution time. In these benchmarks, the average decrease in total time is 4%.

The reduction in GC time due to pretenuring is not as large as one might expect from the reductions in the amount of data copied (71%, 18%, 4%, and 44%). Although data is pretenured accurately, we must still scan the data to check for references to the younger generation. Thus, the cost of the collection is still proportional to the amount of live data though with a smaller constant. This is rather unfortunate since "most" of the pretenured data will not have references to the younger generation.

There are different approaches one could take to reduce the cost of the heap scan associated with pretenuring. When the data is allocated into the older generation directly, we could at the same time group them according to their allocation sites. This permits the scan to be specialized or omitted altogether and should lead to a reduction in instruction count and memory traffic. A more interesting but much harder improvement is to perform a control-flow and dataflow analysis on objects allocated from the targeted sites. Abstractly, we have a set of target sites S. At each allocation site s, we would like to determine a set of sites P(s) such that any object that reaches s was allocated at a site in P(s). Then we can classify each target site s by whether $P(s) \subseteq S$. If so, then there is no need to scan objects arising from s since all such objects will have already been pretenured anyhow. There is even the possibility of additionally pretenuring objects in P(s) so that objects from site s do not have to be scanned. However, there is a danger of over-tenuring if P(s) is so large that even objects that are unlikely to live on their own become tenured. We indicate some preliminary results of such an extension in the next section.

7 Discussion and Extensions

7.1 Generational Stack Collection

Generational stack collection can also be used with non-generational collectors since the issue of root processing and heap copying are orthogonal. Further, the placement policy of stack markers presented above is just one of several possible choices. Also, a more dynamic policy of marker placement may achieve better performance with fewer markers. If activation records are heap-allocated, then the benefit of generational stack collection depends on the relative cost of the bookkeeping to that of interpreting activation records that are heap-allocated.

Given the cost of scanning a deep stack, one might ask whether heap-allocated activation record might be more advantageous than using a stack as generational collectors with heap-allocated record provide automatic tenuring of the frames. However, if the call depth is high, many heap-allocated records might have to be copied at some point whereas the stack implementation never copies frames. Further, in a compiler that supports callee-save registers, the state of the registers cannot be determined from the most recent frame. Thus, either a generational technique like the one presented here would have to be added or the mutator must bear some runtime cost by maintaining a mask register (Appel 1992). In a system like TIL where types are passed at runtime for polymorphic code, maintaining a mask register would entail an even greater cost.

7.2 Pretenuring

Since the TIL compiler does not currently have a data-flow analysis on allocated objects, the effectiveness of reducing scans of pretenured objects is tested by manually analyzing the allocation pattern of the Nqueens benchmark. After data flow analysis reveals the dependence of allocation sites, pretenured objects are divided into two groups. One group is shown to contain objects that point only to pretenured objects while the other group contains objects that may point to objects allocated in the nursery. By removing the scans on objects in the first group, the GC time of Nqueens is further reduced by another 80%, reducing the overall GC% time to 1.3%. This result is promising and an automated system for detecting such sites would be needed to make this optimization useful. Some useful techniques that may help with this problem are region inference (Tofte and Talpin 1993; Tofte and Talpin 1994; Aiken, Fähndrich, and Levien 1995; Birkedal, Tofte, and Vejlstrup 1996), dataflow analysis (Allen and Cocke 1970; Kennedy 1981), and control flow analysis (Shivers 1991).

For the remaining objects which may point to the nursery, another optimization is possible. We pretenure data from different allocation sites into separate areas. Then, some areas may require no scanning because they contain no pointers. Other areas may permit specialized scans. For example, in an area that contains records[‡], no decoding of the tag word is needed. Instead, the collector can directly examine only the fields that are known to contain pointer.

In general, the effectiveness of pretenuring is dependent on the tenuring policy of the generational collector used. In some systems, objects in the nursery are not immediately promoted but are copied/compacted back to the nursery. Counter bits within each object record the number of minor collections the object has survived. When some threshold is reached, the object is then promoted to the tenured generation. Since objects that are tenured are copied several times before being promoted, pretenuring in such systems is likely to yield an even greater benefit than in the system we studied. Finally, most recent versions of generational collectors support several generations. To obtain the maximum benefit from pretenuring, one must consider the average age of the target sites (column 6 of Tables 6 and 7) to determine which generation an object should be pretenured to. Underestimations of the correct generation would lead to extra data copying while overestimations would lead to overtenuring (which unnecessarily ties up memory until the next sufficiently major collection).

8 Related Work

Fateman found that some Franz Lisp programs spend from 25% to 40% of their time in garbage collection (Fateman 1983). Of the 300 or so combinations of LISP systems and benchmarks that could report GC times, the average was 38%. Ungar (Ungar 1984) showed the effectiveness of generational garbage collection in reducing pause times and improving overall performance. Shaw analyzed extensively four programs running on a commercial Common Lisp system and found that generation checks alone can cost as much as 15% of total execution time (Shaw 1988). Zorn investigated the GC cost of eight large programs using a commercial Common Lisp system and found that simulated GC times should be between 10% to 20% (Zorn 1989). Barrett and Zorn (Barrett and Zorn 1993) used lifetime predictors to improve memory overhead and reference locality in the context of explicit memory management. They also studied a mechanism that allows effective reclamation of tenured garbage through a process of untenuring (Barrett and Zorn 1995). Diwan and Tarditi (Tarditi and Diwan 1994) found the overall cost of automatic storage management under SML/NJ to be between 22% to 40%. They found that allocation and root processing can be a significant fraction of the total cost. Röjemo and Runciman (Röjemo and Runciman 1996) used heap profiling to study the the lifetime behavior of data in the context of Haskell. Wilson (Wilson 1994) pointed out the importance of keeping root scanning costs low in incremental garbage collection techniques.

9 Conclusions and Future Work

Generally, even a simple generational collector outperforms a semispace collector by a factor of two or more in terms of

[‡] generated by monomorphic code

garbage collection time, often also accompanied with a significant gain in the client time from improved cache locality. However, there are special cases when a semispace collector can outperform a generational collector: restricted memory usage such as when the total amount of memory used fits inside the cache, a monotonically growing set of long-lived data, or when long-lived data dies quickly once tenured. It might be advantageous for a collector to alternate between these strategies by testing for the above conditions dynamically. For deeply non-tail recursive programs with a runtime implementation that uses stack-allocated activation records, caching the results of stack scans from previous collections can speed up collections. We observe relative decreases in GC times of up to 74% with generational stack collection. Finally, profiling the heap to gather lifetime data for objects allocated from certain program points seems to provide highly accurate data for some of the programs we studied. Even a simple pretenuring policy can reduce the collection time by up to 50% *without* increasing client times.

Incremental and concurrent collectors reduce pause times by performing collections that are more frequent but less disruptive. Nonetheless, they must synchronize on the root set. (That is, though garbage in the heap may be gradually removed, the stack scan is still performed atomically.) In this setting, caching the results of stack scans can be helpful in reducing synchronization costs. The usefulness of heap pretenuring rests largely on the predictability of object lifetimes based on allocation sites. Barrett and Zorn observed this predictability for four substantial C programs in (Barrett and Zorn 1993). One might speculate that this condition is more likely to hold for languages that allocate heavily such as Haskell, LISP, and Java.

There are many directions to go from here: exploration of more generations, generation resizing policies, tenuring policy, control-flow analysis to automatically eliminate pretenuring scans, opportunistic garbage collection, tag-free collection, write barrier techniques. One interesting direction is to reexamine all of the GC ideas with the attitude of aggressively using profile data and type information to generate specialized hybrid garbage collectors. Finally, more and bigger programs need to be analyzed to avoid having too few or only unrealistic datapoints.

Acknowledgements

Chris Stone deserves thanks for general discussions about garbage collection, advice on benchmarking, and, as usual, a thorough review of this paper. Greg Morrisett provided useful discussions for the idea of using stack markers to implement generational stack collection. David Tarditi, Lars Birkedal, Edoardo Biagioni, Ken Cline, and David Eckhardt deserve thanks for careful reviews on a draft of this paper.

References

Aiken, A., M. Fähndrich, and R. Levien (1995). Better static memory management: Improving region-based analysis of higher-order languages. Technical Report CSD-95-866, University of California at Berkeley.

Allen, F. and J. Cocke (1970). A program data flow analysis procedure. In *Communications of the ACM*, pp. 137–147.

Appel, A. W. (1989, February). Simple generational garbage collection and fast allocation. In *Software Practice and Experience*, pp. 171–183.

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

Appel, A. W., J. S. Mattson, and D. Tarditi (1989). A lexical analyzer generator for Standard ML. Distributed with Standard ML of New Jersey.

Barrett, D. and B. Zorn (1993). Using lifetime predictors to improve memory allocation performance. In *ACM Programming Languages Design and Implementation (PLDI)*, pp. 187–196.

Barrett, D. and B. Zorn (1995). Garbage collection using a dynamic threatening boundary. In *ACM SIGPLAN*, pp. 301–314.

Biagioni, E., R. Harper, P. Lee, and B. Milnes (1994). Signatures for a network protocol stack: A systems application of Standard ML. In *LFP*, pp. 55–64.

Birkedal, L., M. Tofte, and M. Vejlstrup (1996). From region inference to von neumann machines via region representation inference. In *Proc. of Principles of Programming Languages (POPL)*.

Cheney, C. (1970). A nonrecursive list compacting algorithm. In *Communication of the ACM*, pp. 677–678.

DEC (1994a). *DEC 3000 300/400/500/600/700/800/900 AXP Models: System Programmer's Manual*. Maynard, Massachusetts: Digital Equipment Corporation.

DEC (1994b). *DECchip 21064 and DECchip 21064A Alpha AXP Microprocessors*. Maynard, Massachusetts: Digital Equipment Corporation.

Ekanadham, K. and Arvind (1987). SIMPLE: An exercise in future scientific programming. Technical Report Computation Structures Group Memo 273, MIT, Cambridge, MA, July 1987. Simultaneously published as IBM/T. J. Watson Research Center Research Report 12686, Yorktown Heights, NY.

Fateman, R. (1983, August). Garbage collection overhead. Provate communication. cited in D. Ungar, *The Design and Evaluation of a High Performance Smalltalk System*, Ph.D. Thesis, UC Merkeley, UCN/CSE 86/287, March 1986.

Fenichel, R. R. and J. C. Yochelson (1969). A LISP garbage-collector for virtual memory computer systems. In *Communications of the ACM*, pp. 4611–612.

Hornof, L. (1992, May). Compiling Prolog to Standard ML: Some optimizations. Undergraduate honors thesis, Carnegie Mellon University. Available as Technical Report CMU-CS-92-166, September 1992.

Jones, R. and R. Lins (1996). *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. New York, NY: John Wiley and Sons.

Kennedy, K. (1981). A survey of data flow analysis techniques. In S. Munchnick and N. Jones (Eds.), *Program Flow Analysis: Theory and Applications*, pp. 5–54. Prentice-Hall.

Knuth, D. (1969). *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*. Reading, Massachusetts: Addison-Wesley.

McCarthy, J. (1960). Recursive funtions of symbolic expressions and their computation by machine. In *Communications of the ACM*, pp. 184–195.

Milner, R., M. Tofte, and R. Harper (1990). *The Definition of Standard ML.* Cambridge, Massachusetts: The MIT Press.

Reade, C. (1989). *Elements of Functional Programming.* Reading, Massachusetts: Addison-Wesley.

Röjemo, N. and C. Runciman (1996). Lag, drag, void and use - heap profiling and space-efficient compilation revisited. In *ACM SIGPLAN International Conference on Functional Programming*, pp. 34–41.

Shaw, R. (1988, February). *Empirical Analysis of a LISP System.* Ph. D. thesis, Stanford University.

Shivers, O. (1991, May). *Control Flow Analysis of Higher Order Languages.* Ph. D. thesis, Carnegie Mellon University.

Sobalvarro, P. (1988, February). A lifetime-based garbage collector for LISP systems on general-purpose computers. Master's thesis, MIT.

Tarditi, Morrisett, Cheng, Stone, Harper, and Lee (1996). TIL: A type-directed optimizing compiler for ML. In *ACM Programming Languages Design and Implementation (PLDI)*, pp. 181–192.

Tarditi, D. and A. Diwan (1994). Measuring the cost of storage management. Technical Report CMU-CS-94-201, Carnegie Mellon University.

Tofte, M. and J.-P. Talpin (1993, July). A theory of stack allocation in polymorphically typed languages. Technical Report Computer Science 93/15, University of Copenhagen.

Tofte, M. and J.-P. Talpin (1994). Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proc. of Principles of Programming Languages (POPL).*

Ungar, D. M. (1984). Generation scavenging: A non-disruptive high-performance storage reclamation algorithm. In *ACM SIGSOFT/SIGPLAN*, pp. 157–167.

Waugh, K. G., P. McAndrew, and G. Michaelson (1990, August). Parallel implementations from function prototypes: a case study. Technical Report Computer Science 90/4, Heriot-Watt University, Edinburgh.

Wilson, P. (1994). Uniprocessor garbage collection techniques. In *Technical report, University of Texas. Expanded version to appear in ACM Computing Surveys.*

Yan, T. (1996). Grobner basis computation. Personal communications.

Zorn, B. (1989, December). *Comparative Performance Evaluation of Garbage Collection Algorithms.* Ph. D. thesis, University of California at Berkeley.