Stack Allocating Objects in Java (Extended Abstract)*

David Gay†
EECS Department
University of California, Berkeley
dgay@cs.berkeley.edu

Bjarne Steensgaard
Microsoft Research
rusa@microsoft.com

Abstract

Allocating objects on the stack rather than the heap reduces the overhead of garbage collection and permits further optimizations. This paper presents a simple and fast algorithm for stack allocating a significant fraction of objects for Java programs. Our implementation in an optimizing batch Java compiler demonstrates the benefit of stack allocation by achieving speed improvements of up to 11% for medium-size programs.

1 Introduction

In garbage-collected object-oriented programming languages, objects are explicitly allocated by the program and implicitly reclaimed when they are no longer used. The lifetimes of some objects are often obvious, while the lifetimes of other objects are more uncertain. Because some objects have an uncertain lifetime, the convention is to treat all objects as equal, allocating and reclaiming them all the same way (on a heap, using garbage collection). A compiler can improve the performance of programs by allocating some of the objects with known, limited, lifetime on a stack. By allocating objects on the call stack, the objects will be automatically reclaimed when the stack frames are reclaimed as functions return. This transformation is safe if the compiler can show that the lifetimes of the objects are delimited by the lifetime of the stack frames.

We present the first published stack allocation system for Java [GJS96]. Our system is simple but effective: for medium-size programs it typically places 10%-20% of all objects on the stack and gives speed improvements up to 11%. Stack allocation is by no means new [Sch75] but most existing work has been performed for functional languages, e.g., [Deu97]. Our speedup is better than that observed for previous work on functional languages [Bla98]; this may be due to the different characteristics of functional and object-oriented languages.

The simplicity of our system ensures its practicality on large programs. Our stack allocation algorithm has a linear time and memory complexity in terms of the size of the program's text and a static approximation of its call graph. We implemented the stack allocation system in Marmot [FKR+98], a batch Java bytecode to native code compiler. Marmot applies most of the conventional optimizations of imperative and object-oriented compilers and its code quality is competitive with other available Java bytecode to native code compilers. We therefore believe that the positive results of stack allocation in Marmot will be reproducible in other compilers.

Stack allocation increases performance in several different ways. Most obviously, garbage collection overhead is reduced. Method stack frames typically have a much shorter lifetime than the interval between

*4389 words, excluding figures.

†The work was performed while the author was working at Microsoft Research.

garbage collections, so stack allocated objects are typically reclaimed faster than heap allocated objects, leading to a reduction in memory usage. Stack allocation of objects typically also leads to improvements in memory locality as memory locations used by short-lived stack allocated objects are reused with the fluctuations of the stack. In some cases it is possible to avoid creating an object entirely, instead representing the fields of the object by local variables, allowing further optimizations.

In our framework, stack allocated objects can only be allocated in the stack frame of the method being executed. In some cases it is desirable to stack allocate an object returned by a method (e.g., an `Enumeration` object). To do this, the object is pre-allocated in the calling method and passed as an argument to a suitably modified copy of the "creating" method. This feature is essential to our results; if we disable it we typically only stack allocate a third as many objects. The same benefit could be obtained in some cases by inlining the "creating" method but this may lead to significant code expansion, which our approach avoids.

In Section 2 we present the mechanisms we use to allocate objects on the call stack rather than on the heap. In Section 3 we present the analysis algorithm that determines which objects can be stack allocated. In Section 4 we present the results of our system when applied to a set of benchmarks, in Section 5 we discuss related work, and in Section 6 we present our conclusions.

2 Program Transformations

We first explain how our system allocates objects on the stack, and the restrictions necessary to guarantee the efficiency and safety of stack allocation (Section 2.1). Using the results of the analyses we present in Section 3 we show how we can transform Java programs to use stack allocation (Section 2.2). In some cases it is possible to decompose an object into a set of local variables, exposing the code that operates on this object to all standard optimization techniques. This process, *exploding an object*, is detailed in Section 2.3. The analysis necessary for stack allocation provides extra type information that enables a number of further transformations, presented in Section 2.4.

For the purposes of this paper we will assume that the Java program has been converted to Static Single Assignment (SSA) form [CFR+91] and that intermediate values of expressions are eliminated by storing such values in extra local variables—we can thus identify expressions with the variable to which they are assigned. Object creation is performed in two steps (as in the Java Virtual Machine [LY97]): (1) memory allocation (x `= new` T) and (2) object initialization (a call to a constructor for T). We also distinguish *direct method calls*, where the compiler can determine the target method, from normal (virtual) method calls. Marmot uses an analysis similar to Rapid Type Analysis [BS96] augmented with local type propagation to determine direct calls.

2.1 Stack Allocation Framework

To represent stack allocation, a new kind of statement is added: x = `newStack` T. There are several restrictions on the use of `newStack` in a method, f. First, none of the objects assigned to x may be live when f returns—this is obviously necessary to ensure the safety of stack allocation. Secondly, T must have a trivial `finalize` method. Allowing stack allocation of objects with a non-trivial `finalize` method makes the implementation more complicated as the compiler would have to insert explicit calls to the `finalize` method. Objects allocated in a loop by the same statement must have non-overlapping lifetimes. Finally, T must not be an array type. The final two restrictions make it possible to statically allocate space for the objects allocated by each instance of `newStack` in f's stack frame.

Figure 1 shows the stack when a method f calls a method g with a `newStack A` and a `newStack B` statement. Our implementation does not currently attempt to minimize the size of stack frames by overlapping stack-allocatable objects and spilled local variables based on their respective lifetimes.

With static allocation of objects in the method's stack frame, stack allocation is very cheap; extra space needs to be reserved in the method's stack frame but this is free as long as the method already required a stack frame. A `newStack` expression only costs one instruction (set the destination to the appropriate offset

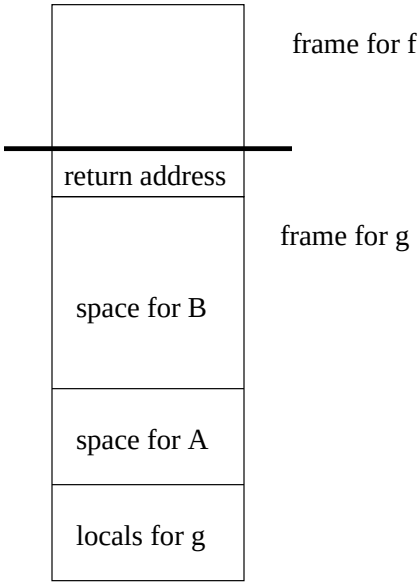

Figure 1: Stack frame layout.

from the stack pointer) plus the cost of memory initialization which is required of all allocations. There is no need for a frame pointer register in methods that perform stack allocation, as would be required with a more dynamic form of stack allocation.

Chase [Cha88] outlined a safety criterion for stack allocation systems: essentially, stack allocation should not make a program that used to run successfully with garbage collection fail because of a lack of memory. A garbage collector can reclaim unreachable objects that a function creates before that function returns. Thus, a recursive function with stack-allocatable objects could use arbitrarily more memory than without stack allocation. We do not believe that this is an important issue for our system because Java programs make much less use of recursion than the functional programs that Chase considered.

2.2 Stack-allocating Objects

Our stack allocation transformation is based on identifying *fresh methods*, *fresh variables* and *stackable variables* (the analyses of Section 3 computes conservative approximations of these properties):

Definition 2.1 (Fresh Method) A method f is *fresh* if, ignoring exceptions, all calls to f allocate and return an unaliased object of some type T. If a statement creating an object that may be returned is executed, then the object must be returned. This restriction simplifies the transformation necessary for stack allocation. The actual allocation does not have to occur in the method itself. For example, if f is defined as `return` $g()$ and g is fresh then so is f.

Definition 2.2 (Fresh Variable) A variable v is *fresh* if the (single) definition of v is $v =$ `new` T or a direct call $v = o.f(\ldots)$ where f is a fresh method. Only direct calls are considered in order to simplify the transformations necessary for stack allocation.

All objects assigned to fresh variables or returned by fresh methods are called *fresh objects*. The fresh variables are the candidates for stack allocation meeting the following definition:

Definition 2.3 (Stackable Variable) A fresh variable v of method f is *stackable* if none of the objects assigned to v are live when f returns or when the definition of v is executed again. In addition, the type of

the object assigned to v may not be an array type and must have a trivial `finalize` method. The objects stored in stackable variables are called *stackable objects*.

It is trivial to stack allocate a stackable object whose definition is $x =$ `new` T: replace the definition with $x =$ `newStack` T. Stackable objects which are the result of a call to a fresh method g require a more complex transformation: replace $x = o.g(\ldots)$ by x `= newStack` T`;` $o.g'(\ldots, x)$, where g' is a method which behaves exactly like g except that the new object is passed to g' as an argument rather than being obtained from the heap. Formally, g' is obtained from g as follows: g' has an extra argument *site* and a `void` result type. The body of g' is the body of g modified as follows: for each fresh variable r whose value may be returned from g:

- If the (single) definition of r is $r =$ `new` T, eliminate it.
- If the definition of r is $r = h(v_1, \ldots, v_n)$, replace it with $h'(v_1, \ldots, v_n, site)$, where h' is obtained from h using this same procedure.
- Substitute all remaining uses of r with *site*.

Also, replace all '`return` v' statements with '`return`' statements.

Stack allocation may be performed on all methods of the program in arbitrary order.

2.3 Exploding Objects

In some cases it is possible to replace a stackable object of type T by a set of local variables, one per field of T. We call this transformation *exploding an object*. A stackable object defined by $v =$ `newStack` T may be exploded if it is possible to inline all methods that take v as an argument and if, after inlining, all uses of v are only to read and write fields of v. Our implementation does not explode objects when a simple heuristic indicates that the inlining would lead to excessive code growth.

For example, many Java objects of type `java.util.Enumeration`, typically used to loop through data structures, are stack allocatable because each object is only used in a single context to step through the elements of the enumeration, which typically only involves accessing the fields of the object. Therefore, it is often possible to eliminate all overhead due to the enumeration abstraction and obtain efficient loop code.

2.4 Incidental Transformations

Knowing that a variable is fresh enables some further optimizations on uses of that variable that are not directly related to stack allocation: null checks can be removed, indirect method calls can be transformed into direct calls, type checks can be removed or transformed into unconditional exceptions, and synchronization can be removed.[1] Our implementation only does null check removal and transformation into direct calls.

3 Program Analysis

The stack allocation analysis must compute conservative approximations of the fresh method, fresh variable and stackable variable properties defined in Section 2.2. To ensure the practicality of stack allocation, the analysis must be fast and scalable. We present an inter-procedural analysis whose memory and runtime complexity is $O(N + G)$, where N is the size of the program and G is the size of the program's static call graph. The size of a call graph is the number of caller-callee pairs plus the length of each pair's argument lists.

We define our algorithm assuming each method is represented by a control flow graph. Each node in the control flow graph has any number of effect statements and is terminated by a single control statement from Figure 2. The program must also be in static single assignment form (hence the ϕ statements). The nodes

[1] This transformation is not strictly legal in all circumstances, e.g., if `java.lang.Object.wait()` may be called on the variable.

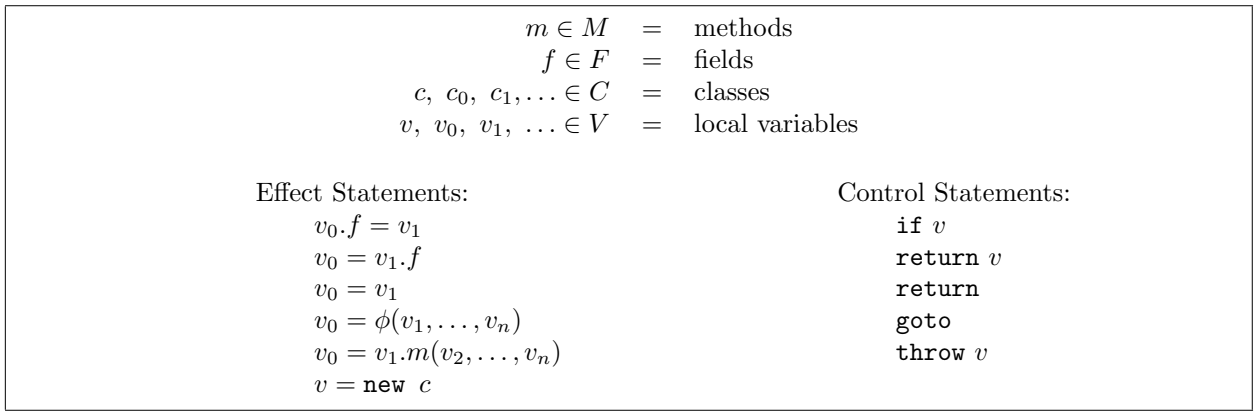

Figure 2: Representative Intermediate Language Statements

of the control flow graph are connected with labeled control flow edges and exception edges. If an exception occurs during the execution of the statements of a basic block, control is transferred to the destination of one of the outgoing exception edges (exception edges are labeled with the kind of exception they represent). This intermediate language is sufficient to explain the interesting parts of our stack allocation analysis and it can easily be extended to handle the whole of Java.

3.1 Stackable Analysis

This section presents analyses to compute the stackable variables assuming the fresh methods are known.

According to Defintion 2.3 (Stackable), an object cannot be allocated in a method's stack frame if references to the object *escapes* from that stack frame. We conservatively consider a reference to have escaped if it is ever stored in a field of an object or if it is returned from the method. This implies that an object is considered to have escaped even if the object is only stored in a field of an object that can be stack allocated in the method's stack frame. The *escape analysis* computes for each local variable whether the value assigned to the variable is returned as a result of the method or if it may be stored in a field of an object in either the method itself or in any called method.

Definition 2.3 also requires that stack allocated objects created by a given statement have non-overlapping lifetimes. The *loop analysis* computes for each variable whether the value assigned to the variable ever occurs in a loop-carried dependency in the method. Our loop analysis makes the very conservative assumption that if a value is ever used in a ϕ-statement, then the value is loop-carried.

Stackability is computed in terms of the results of the escape analysis and the loop analysis. A *fresh variable* is also a *stackable variable* if the values assigned to the variable do not escape and are not loop-carried.

The analyses are backward analyses expressed as sets of constraints. The *stored*, *returned*, and *loop* relations used by the escape and loop analyses associates local variables with elements from the boolean lattice

$$\tau_b = \{\bot, \top\}.$$

For all variables v $\mathit{stored}(v) = \top$ if the values of v may be stored, $\mathit{returned}(v) \sqcup \mathit{stored}(v) = \top$ if the values of v may be returned, and $\mathit{loop}(v) \sqcup \mathit{stored}(v) = \top$ if the values of v may be loop-carried,

The *mfresh*, *vfresh*, and *stackable* relations, representing fresh methods, fresh variables, and stackable variables respectively, associates methods and local variables with elements from the flat lattice

$$\tau_j = \{\bot, \top\} \cup C.$$

The partial order of the lattice τ_j has the Java reference types, C, as *discrete* proper elements.[2] A method

[2]By discrete we mean that the least upper bound of any two distinct proper elements is $\top$.

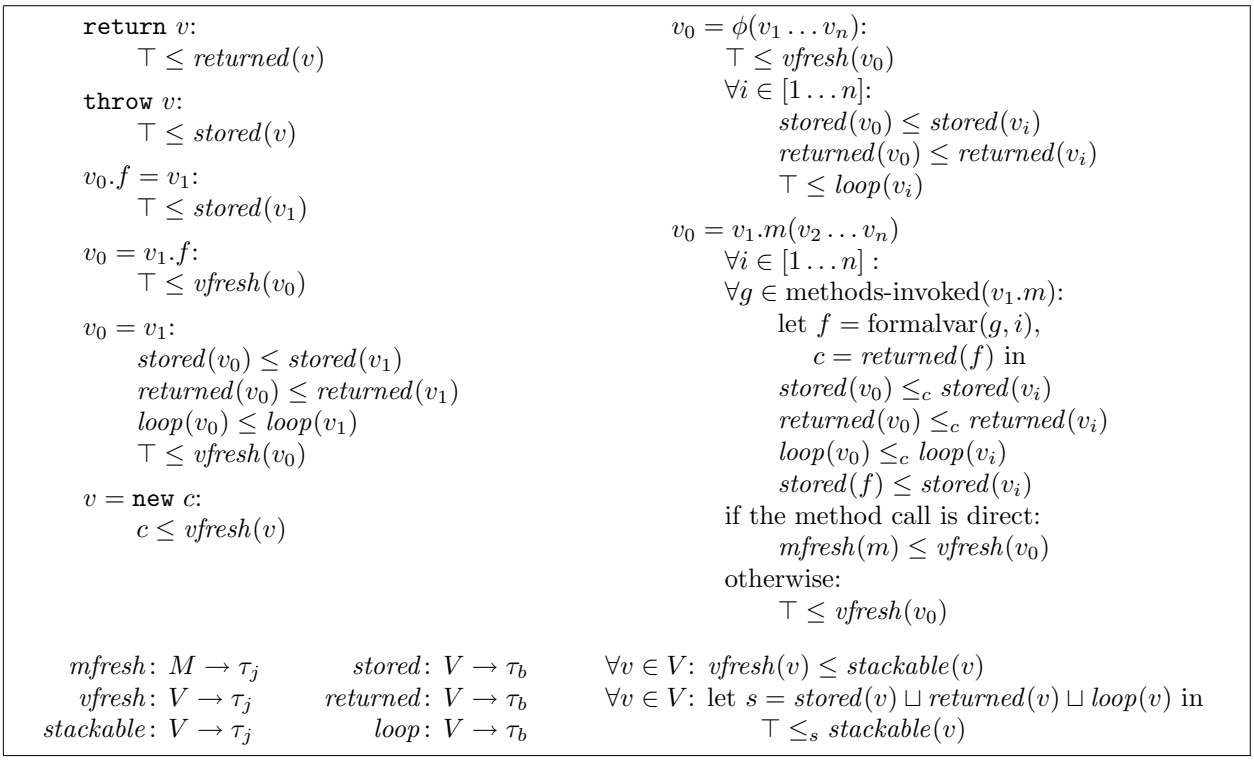

Figure 3: Constraints implied by the syntactic forms of statements.

m is fresh and returns an element of type c if $mfresh(m) = c$. A variable v is fresh (respectively stackable) and assigned an object of type c if $vfresh(v) = c$ (respectively $stackable(v) = c$).

Constraints are added for each statement in each method. The syntactic form of each statement implies a set of constraints as illustrated in Figure 3. There are regular constraints of the form $a \leq b$, and conditional constraints of the form $a \leq_c b$, where the regular constraint $a \leq b$ is imposed only if c is $\top$.

For an assignment statement, `x = y`, the constraints imply a simple propagation of properties. For example, the constraint

$$stored(\texttt{x}) \leq stored(\texttt{y})$$

implies that if the value in `x` is ever stored in a field, then so is the value in `y`.

The analysis does not track the flow of thrown exception objects, so it conservatively assumes that thrown objects may be stored.

The rule for method invocation has several components. The set 'methods-invoked$(v_1.m)$' contains all methods that may be invoked from that call site according to the static call-graph. If a formal parameter is stored, then the corresponding argument is also considered stored. If a parameter value may be returned, the corresponding argument may be assigned to the method's result. The analysis represents this with a degree of inter-procedural context sensitivity by using conditional constraints. For instance, if a method behaving like the identity function is invoked and the result is stored, then the argument to the method is also stored. Finally, if a fresh method is invoked in a direct call then the result variable is fresh.

3.2 Fresh methods

This section presents the analysis that approximates whether a method m is fresh, and if so, the type of the returned objects. Constraints on $mfresh(m)$ are added by inspecting m.

Name	LOC	Description
marmot	88K	Marmot compiling itself
jessmab	14K	Java Expert Shell System solving "Bananas and Monkeys" problem.
jessword	14K	Java Expert Shell System solving the "Word game" problem.
jlex	14K	JLex generating a lexer for sample.lex.
javacup	8.8K	JavaCup generating a Java parser
parser	5.6K	The JavaCup generated parser parsing Grm.java
slice	1K	Graphics animation based on SliceViewer

Table 1: Benchmarks

A fresh method must allocate and return an unaliased object of some type T (Defintion 2.1). The analysis approximates this condition by requiring that all return variables are assigned unstored, fresh objects of the same type. For each return variable v the following constraints are added:

$$\mathit{vfresh}(v) \leq \mathit{mfresh}(\mathit{method}) \qquad\qquad \top \leq_{\mathit{stored}(v)} \mathit{mfresh}(\mathit{method})$$

If a statement creating an object that may be returned is executed, then the object must be returned (modulo exceptions causing termination of the method) (Definition 2.1). This condition is true if for all return variables v:

- There is no path in m from the definition of v back to itself.
- All paths in m from the definition of v terminate in `return` v.

Each of these conditions on paths in m is computed in a depth-first traversal of m. If either condition is not satisfied, the following constraint is added:

$$\top \leq \mathit{mfresh}(\mathit{method})$$

3.3 Complexity

Constraints of the form $a \leq b$ and $a \leq_c b$ can be solved in linear time by using pending lists on a and c as long as all lattices are of constant height [RM98].

The lattices for *stored*, *returned*, *loop*, *mfresh*, *vfresh* and *stackable* are all of constant height, and the number of constraints is proportional to N (the program size) + G (the call graph size), so our analysis runs in $O(N + G)$ time and space.

4 Results

We implemented our stack allocation system and measured its performance on the set of benchmarks described in Table 1. On these benchmarks, our stack allocation system typically places 10–20% (and 73% in one case) of allocated objects on the stack (Section 4.1) and produced speedups of up to 11% (Section 4.2).

4.1 Stack Allocation Behavior

Table 2 summarizes the effectiveness of our stack allocation analysis. It shows for all benchmarks the increase in code and maximum stack size due to stack allocation, the total number of allocations and the allocation rate per second, the percentage of objects that were allocated on the stack, and the percentage of allocated objects that were exploded into local variables.

The data demonstrates that a significant fraction of all objects can be identified as stackable using a very simple analysis algorithm. The increase in stack size is relatively small. We were unable to measure the reduction of heap size to a useful degree of precision due to the nature of the garbage collector.

Name	% Code increase	Stack increase	Allocated objs	Allocated bytes	Allocation rate objs/s	Allocation rate bytes/s	% Stack allocated objects	% Stack allocated bytes	% Exploded objects	% Exploded bytes
marmot	8.5	24.4KB	98M	2289MB	160K/s	3.7MB/s	20.6	13.9	11.1	7.1
jessmab	1.2	936 B	510K	19MB	160K/s	5.9MB/s	19.1	10.5	0.3	0.2
jessword	1.2	1080 B	305K	11MB	77K/s	2.7MB/s	9.7	5.3	0.1	0.1
jlex	2.5	328 B	41K	1.6MB	110K/s	4.3MB/s	15.7	7.2	1.8	0.5
javacup	6.9	328 B	785K	21MB	547K/s	14.5MB/s	12.6	8.8	1.8	1.3
parser	0.3	116 B	1201K	32MB	204K/s	5.4MB/s	1.7	1.2	0.0	0.0
slice	3.3	220 B	463K	16MB	32K/s	1.1MB/s	72.9	62.4	72.2	62.1

Table 2: Objects allocated on the stack

Name	Incidental	Stack	Explode
marmot	0.8%	9.6%	11.0%
jessmab	0.3%	2.6%	3.0%
jessword	2.6%	1.1%	3.6%
jlex	0.0%	-2.1%	0.0%
javacup	-0.6%	0.6%	6.6%
parser	1.0%	-2.2%	-2.3%
slice	-0.6%	-3.9%	1.2%

Table 3: Speedup due to the various transformations. The transformations are additive from left to right.

For the programs in our test suite, we found that the restriction to only stack allocate objects with trivial finalize methods had no effect on the number of stack allocated objects. The restriction to not pre-allocate objects returned by indirect calls of a set of fresh methods had only minimal effects. For Marmot compiling itself, the percentage of stack allocated objects could be increased by half if we could stack allocate arrays and objects referenced in fields of other stack allocated objects.

4.2 Performance

Table 3 shows the speedup[3] provided by our stack allocation system. The machine used is a Gateway E-5000 with a Pentium II at 300MHz and 512MB of RAM, running Windows NT 4.0 SP3. All execution times are measured with the Pentium cycle counter and represent the best of 10 runs. We successively enable the incidental optimizations of Section 2.4 (shown in the 'Incidental' column), stack allocation ('Stack' column) and object explosion ('Explode' column) so that we can distinguish their contribution. Marmot currently uses a simple conservative garbage collector. We expect to see somewhat smaller, but still positive, gains when using an improved garbage collector.

The data demonstrates that stack allocation generally results in performance gains for the test cases we tried. There are, however significant speed variations which cannot be attributed to stack allocation, e.g., 'parser' has a slowdown of 2.3% but hardly does any stack allocation. The biggest performance gain observed was for Marmot compiling itself, which is also the test case with the highest relative garbage collection cost (not shown). However, the speedup from stack allocation for Marmot is still 7.4% if the costs of garbage collection are ignored. It is interesting to note that for many programs a significant fraction of the performance gain is achieved by eliminating object entities and replacing their field constituents with local variables.

[3]Computed as $(oldtime - newtime)/oldtime$.

5 Related Work

A number of different approaches have been considered for moving some dynamic allocations from the heap to the stack (stack allocation) or for the very similar problem of explicitly freeing objects that are known to be unreachable (compile-time garbage collection). The two major approaches are escape analysis and region inference.

Stack allocation and compile-time garbage collection based on escape analysis have mostly been performed on functional languages like SML [MTH90] or Lisp [Bla98, Deu97, Hug92, ISY88, Moh95, PG92]. The precision of the escape analyses considered in these papers is much greater than ours. For instance, they can find that a function's result does not contain elements from the spine of its second argument (a list). This extra precision appears to be necessary for effective stack allocation in these list-oriented languages. The only algorithm with a near-linear complexity is that of Alain Deutsch [Deu97] (the complexity is $n \log^2 n$ where n is the program's size). A study of the performance of this algorithm on some SML benchmarks was performed by Bruno Blanchet [Bla98]. On the only large program considered, this algorithm placed 25% of allocated bytes on the stack and gave a performance improvement of 3-4%. As in our system, the performance gains are only partially attributable to reduced garbage collection overhead.

The region inference system of Tofte and Talpin [TT94, TT97] infers a set of *regions* in which to place all allocated objects. The only way to free memory is to delete a region; the allocation and deallocation of regions must follow a stack discipline. That requirement was relaxed in [AFL95]. The regions are dynamically sized, but in the cases where a bound on their size can be computed they can be placed on the stack. This system is much more general than ours, but the analysis is also much more complex. In addition, the programmer must sometimes modify the program to avoid leaking memory. To perform these modifications the programmer must understand the region inference process. Ruggieri and Murtagh [RM88] propose a similar but less general system.

Alias and points-to analysis [CWZ90, Deu90, Hic93, JM81, SF96, VHU92] and other miscellaneous techniques [Bar77, BS93, JM90, JL89, Sch75] have also been considered for compile-time garbage collection. These analyses are mostly expensive and their effectiveness is unclear.

McDowell instrumented a Java virtual machine to find the number of potentially stackable objects in a small collection of Java programs [McD98]. However this study did not consider the possibility of stack allocating objects returned by 'fresh' functions, so its results are incomparable with ours.

6 Conclusion

We have designed and implemented a simple and inexpensive stack allocation system that allocates Java objects on the call stack instead of on the heap. This system is effective, giving speedups of up to 11%.

The stack allocation transformation could be improved by reusing memory in the stack frame for objects with non-overlapping lifetimes. Better results may also be achieved by improving the quality of the escape analysis.

References

[AFL95] Alexander Aiken, Manuel Fahndrich, and Raph Levien. Better static memory management: improving region-based analysis of higher-order languages. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, pages 174–185, La Jolla, CA, June 1995.

[Bar77] Jeffrey M. Barth. Shifting garbage collection overhead to compile time. *Communications of the ACM*, 20(7):513–518, July 1977.

[Bla98] Bruno Blanchet. Escape analysis: Correctness proof, implementation and experimental results. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 25–37, San Diego, California, 19–21 January 98.

[BS93] E. Barendsen and S. Smetsers. Conventional and uniqueness typing in graph rewrite systems. In Rudrapatna K. Shyamasundar, editor, *Proceedings of Foundations of Software Technology and Theoretical Computer Science*, volume 761 of *LNCS*, pages 41–51, Bombay, India, December 1993. Springer-Verlag.

[BS96] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 31, 10 of *ACM SIGPLAN Notices*, pages 324–341, New York, October6–10 1996. ACM Press.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[Cha88] D. R. Chase. Safety considerations for storage allocation optimizations. In David S. Wise, editor, *Proceedings of the SIGPLAN '88 Conference on Programming Lanugage Design and Implementation (SIGPLAN '88)*, pages 1–10, Atlanta, GE, USA, June 1988. ACM Press.

[CWZ90] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. *ACM SIGPLAN Notices*, 25(6):296–310, June 1990.

[Deu90] Alan Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In ACM-SIGPLAN ACM-SIGACT, editor, *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages (POPL '90)*, pages 157–168, San Francisco, CA, USA, January 1990. ACM Press.

[Deu97] Alain Deutsch. On the complexity of escape analysis. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 358–371, Paris, France, 15–17 January 97.

[FKR+98] Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: an optimizing compiler for Java. In preparation. Submitted to PLDI'99, October 1998.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, June 1996.

[Hic93] James Hicks. Experiences with compiler-directed storage reclamation. In R. John M. Hughes, editor, *Record of the 1993 Conference on Functional Programming and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, Copenhagen, June 1993. Springer-Verlag.

[Hug92] Simon Hughes. Compile-time garbage collection for higher-order functional languages. *Journal of Logic and Computation*, 2(4):483–509, August 1992.

[ISY88] Katsuro Inoue, Hiroyuki Seki, and Hikaru Yagi. Analysis of functional programs to detect run-time garbage cells. *ACM Transactions on Programming Languages and Systems*, 10(4):555–578, October 1988.

[JL89] S. B. Jones and D. Le Métayer. Compile-time garbage collection by sharing analysis. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture '89, Imperial College, London*, pages 54–74, New York, NY, 1989. ACM.

[JM81] Neil D. Jones and Steven S. Muchnick. Flow analysis and optimization of Lisp-like structures. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 102–131. Englewood Cliffs, N.J.: Prentice-Hall, 1981.

[JM90] Thomas P. Jensen and Torben Mogensen. A backwards analysis for compile-time garbage collection. In Neil D. Jones, editor, *ESOP'90 3rd European Symposium on Programming, Copenhagen, Denmark, May 1990. (Lecture Notes in Computer Science, vol. 432)*, pages 227–239. Springer-Verlag, 1990.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1997.

[McD98] C.E. McDowell. Reducing garbage in Java. *SIGPLAN Notices*, 33(9):84–86, September 1998.

[Moh95] Markus Mohnen. Efficient compile-time garbage collection for arbitrary data structures. Technical Report 95-08, RWTH Aachen, Department of Computer Science, 1995.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[PG92] Young Gil Park and Benjamin Goldberg. Escape analysis on lists. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI)*, pages 116–127, 1992.

[RM98] Jakob Rehof and Torben Æ. Mogensen. Tractable constraints in finite semilattices. *Science of Computer Programming*, 1998. to appear.

[RM88] Cristina Ruggieri and Thomas P. Murtagh. Lifetime analysis of dynamically allocated objects. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 285–293, San Diego, California, January 88.

[Sch75] J. T. Schwartz. Optimization of very high level languages – I. Value transmission and its corollaries. *Computer Languages*, 1(2):161–194, 1975.

[SF96] Manuel Serrano and Marc Feeley. Storage use analysis and its applications. In *Proceedings of the 1st International Conference on Functional Programming*, page 12, Philadelphie, June 1996.

[TT94] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *Conference Record of the Twenty-first Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 188–201. ACM Press, January 1994.

[TT97] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, February 1997.

[VHU92] Jan Vitek, R. Nigel Horspool, and James Uhl. Compile-time analysis of object-oriented programs. In *Proceedings of the 4th Int. Conf. on Compiler Construction, CC'92*, Paderborn, Germany, 1992. Springer-Verlag.