The Need for Predictable Garbage Collection

Alastair Reid, John McCorquodale, Jason Baker,
Wilson Hsieh, Joseph Zachary
Department of Computer Science
University of Utah

Abstract

Modern programming languages such as Java are increasingly being used to write systems programs. By "systems programs," we mean programs that provide critical services (compilers), are long-running (Web servers), or have time-critical aspects (databases or query engines). One of the requirements of such programs is predictable behavior. Unfortunately, predictability is often compromised by the presence of garbage collection. Various researchers have examined the feasibility of replacing garbage collection with forms of stack allocation that are more predictable than GC, but the applicability of such research to systems programs has not been studied or measured. A particularly promising approach allocates objects in the nth stack frame (instead of just the topmost frame): we call this *deep stack allocation*. We present dynamic profiling results for several Java programs to show that deep stack allocation should benefit systems programs, and we describe the approach that we are developing to perform deep stack allocation in Java.

1 Introduction

Predictable behavior is an essential property for systems programs. By systems program, we mean a wide variety of "real" programs that people use or depend on: programs that run for long periods of time (Web servers), programs that provide important services (compilers), or programs that represent commercial workloads (expert systems and databases). Systems programmers like to be able to predict, for example, how much memory a program will consume or how long it will take to complete. For conventional C programs, this goal is viable (if sometimes difficult): the C runtime system is very lightweight and has a predictable effect on running time.

Unfortunately, predictability is not a given for programs written in garbage-collected languages such as Java because of several aspects of garbage collectors. First of all, garbage collection runs asynchronously; one cannot reason about when it will occur or how long it will last. Second, garbage collectors are complicated systems whose behaviors vary greatly. Accordingly, most programmers treat garbage collection as a black box which they hope will do the Right Thing since they can't hope to reason about how much memory will be recovered by each collection. Third, it is impossible to predict the order in which the garbage collector will execute finalizers.

We believe that techniques must be found to minimize the impact of these problems if garbage-collected languages are to be viable for systems programming. There are several standard approaches to this:

- Critical code can be dealt with by reserving memory for use in that code and disabling GC inside that piece of code. This solution is only viable if the memory requirements can be bounded reasonably well, which is hard to achieve for large sections of code.
- Real-time garbage collectors [3, 5] interleave garbage collection with program execution: some garbage collection occurs every time an object is allocated. This solution addresses the several problems, but it has a considerable cost in space and/or time.
- Reference counting also performs book-keeping continuously during program execution [11]. Unlike real-time garbage collection, we can predict when objects are deallocated, which allows us to predict how much memory is available at any given time. However, reference counting is costly and cannot collect cyclic data structures.

To address these problems, we are investigating "deep stack allocation" of objects; this research combines ideas from both Ruggieri and Murtagh's work on lifetime analysis [15] and Tofte and Talpin's work on region-based allocation [17]. Conventional stack allocation uses a static analysis to infer which objects will not outlive the topmost stack frame; these objects can be allocated in the topmost frame. Deep stack allocation uses a more sophisticated static analysis to infer which objects will not outlive the nth stack frame and allocates those objects there. Section 5 describes how we can allocate an object in a buried stack frame.

This research was supported in part by the Defense Advanced Research Projects Agency, monitored by the Department of the Army under contract number DABT63–94–C–0058, and the Air Force Research Laboratory, Rome Research Site, USAF, under agreement number F30602–96–2–0269. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation hereon. The opinions and conclusions contained in this document are those of the authors and should not be interpreted as representing official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Given our interest in systems software, it was natural to investigate deep stack allocation in the context of Java, since Java is increasingly being used to write systems software. In particular, colleagues at the University of Utah have written an OS kernel based on Java [19] and are continuing to work on Java-based systems.

In this paper we:

- Discuss how we use an instrumented Java virtual machine to measure, for a range of benchmarks, how much potential there is for stack allocation.
- Describe a straightforward adaptation of Serrano and Feeley's algorithm for statically identifying opportunities for stack allocation in Scheme and ML programs [16] to produce an analyzer for Java programs.
- Report the (negative) result that this analysis fails to detect most stack allocatable objects in our benchmark programs, and comment on how the analysis could be sharpened. (Java is much harder to analyze than Scheme.)
- Outline possible implementations of deep stack allocation.

2 The Potential Gains

To establish whether stack allocation has any potential at all, we measured object lifetimes in actual program runs. Dieckmann and Hölzle [9] gathered similar data for Java programs, but they measured object reachability in terms of the number of bytes allocated since an object was created. It is difficult to relate their data to stack allocation because they do not report how long stack frames live. In addition, we cannot determine from their data how deep in the stack objects can be allocated. To measure the potential benefits of deep stack allocation, we measure the actual liveness of every object in our benchmarks.

2.1 Methodology

We ran our experiments on an instrumented version of the Kaffe VM [18], a publicly available Java virtual machine. This version of Kaffe generates dynamic traces of allocation and use information for every object.

Our profiler collects the following information for every object: its allocation site, which thread allocated it, when it was allocated, when it was last touched, and when it became unreachable. We approximate the point at which an object becomes unreachable by determining when it is garbage collected; we minimize approximation error by forcing Kaffe to garbage collect very frequently. The profiler also collects the following information about each stack frame: which thread allocated it, when it was allocated, and when it was deallocated.

This information was used by a post-mortem analyzer to calculate the data in this paper. The analyzer determines the uppermost stack frame in which an object could have been allocated, as well as when the object would have died had it been allocated there. Times are measured in total instructions executed since the start of the run.

Our data are based on profile data rather than on a static analysis. We must take some care in how we interpret them: they show us the maximum possible gain from using a particular style of implementation, rather than the actual gain from an actual implementation. In addition, object lifetimes may be dependent on input data, and a static analyzer obviously cannot take advantage of such knowledge. Finally, Kaffe's garbage collector is conservative, so the amount of data that it considers reachable is greater than in systems with precise garbage collection.

2.2 Results

We measure several Java benchmarks that exhibit interesting "systems" behavior: a ray tracer written by a visualization researcher at the University of Utah; Webster, a simple Web server; and four of the SPEC JVM98 benchmarks: jess, an expert system; javac, a Java compiler; db, a small database; jack, a compiler compiler. We chose these benchmarks because they have a range of allocation behaviors, and because they exhibit some of the qualities that we associate with systems programs.

Table 1 shows the potential reduction in total heap allocation as the maximum allocation depth is increased. We observe that allocating objects only in the topmost frame reduces the total heap allocation by 11.7-38.5%, but allocating just a little deeper in the stack reduces the total allocation by more than 80%. This tremendous reduction in heap allocation suggests that there are enormous potential gains to be had by using deep stack allocation. Furthermore, the relatively shallow depths at which significant gains are obtained suggests that a static analyzer has a chance of achieving some of these gains. We consider it unlikely that a static analyzer will detect objects that can be allocated very deep on the stack.

3 Simple Deep Allocation

We have implemented an analysis technique that is similar to an algorithm developed by Serrano and Feeley [16]. Their analysis can determine when it is legal to replace heap allocations with stack allocations in Scheme and ML. They present an iterative dataflow technique that spreads a notion of "unstackability" among allocations. Upon convergence, an allocation which is found to be

Benchmark	Maximum depth in stack for allocations						
	heap only	0	1	2	3	5	10
jess	1941487	71.4%	36.9%	25.6%	13.0%	10.6%	2.1%
javac	1256829	72.8%	46.3%	30.0%	12.9%	5.6%	1.0%
rt	1146706	78.3%	6.4%	2.3%	0.2%	0.1%	0%
webster	760339	61.5%	40.1%	6.9%	1.1%	0.4%	0%
jack	18781083	76.3%	23.3%	12.9%	3.3%	2.4%	1.2%
db	484173	70.2%	45.4%	27.3%	18.5%	16.1%	12.7%
range		61.5-78.3%	6.4-46.3%	2.3-30.0%	1.6-19.4%	0.1-16.1%	0-12.7%

Table 1: Number of heap-allocated bytes under different stack allocation possibilities. The column headings represent the maximum depth in the stack at which objects could be allocated.

reachable from a global or is returned from the function that allocates it is marked “unstackable.” Any allocation not so marked is known to not outlive its function’s activation and can therefore be allocated on the activation stack. Our algorithm has modifications appropriate to Java control flow and naming:

- Polymorphic call-site binding is handled by conservatively binding all potential implementations.
- Unstackability by return from a function is extended to include exceptional function return: objects reachable from thrown exceptions are unstackable.
- Unstackability by global reachability is reinterpreted to include reachability from static variables or from any object allocated on the heap.

In addition, we have extended the Serrano and Feeley algorithm to determine when an object does not outlive the activation frame of some function on the call chain at the time of allocation. Our algorithm is given in Appendix A.

Figure 1 shows the projected reduction in heap allocation rate for the six programs if the results of our static analysis were used to allocate objects in the top 20 activation records on the stack. Allocation decisions are made at analysis time for each allocation site in the program.

Our simple analysis performs comparatively well on the ray tracer benchmark. The analysis is able to determine that many objects allocated in the tracer’s inner loop can be allocated in the activation frame of the method containing the loop. Doing so can reduce the rate of heap allocation by more than a factor of two.

Our analysis performs poorly on the rest of our benchmarks: it usually achieves a 0 to 10 percent reduction in the heap allocation rate. Two of the programs saw no appreciable decrease in heap allocation rate. As a result, more complex analyses will be necessary to achieve results closer to those indicated by the measurements in Section 2.

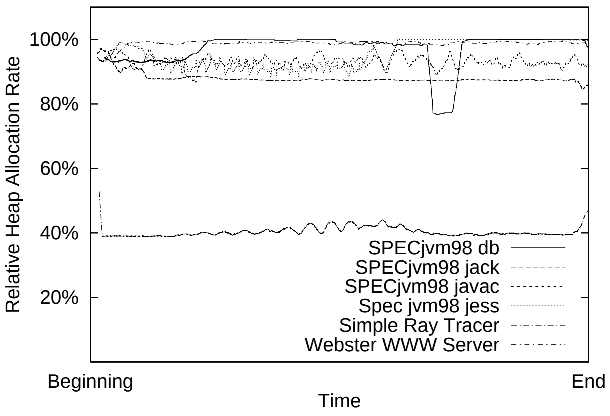

Figure 1: Reduction in heap allocation rate (bytes/time) achieved using static analysis for a maximum allocation depth of 20. The bottom curve is the ray tracer.

The lesson from implementing this analyzer is that Java requires a more sophisticated analysis than Scheme or ML. We believe that the analyzer is failing for three reasons:

1. The analyzer does not propagate type information. This forces the analyzer to be very pessimistic about method dispatch, since it is usually not possible to determine which method is called without knowing the type of the object.
2. Objects become unstackable if they are referenced by other objects. This is unfortunate since it penalizes the common object-oriented paradigm of constructing objects by aggregation. Analyses such as those used to inline objects [10] could help.
3. Java programs contain many more assignments than is typical in Scheme and ML programs. As usual, assignment complicates analysis because all objects

assigned to the same variable are assigned the same lifetime.

4 How We Can Do Better

We are working on an improved analyzer and better support for stack allocation at runtime; in this section we discuss some of the issues that we are dealing with. Chase [7] warns of a potential problem with stack allocation: allocating an object on the stack can change its lifetime. If an object is allocated near the top of the stack, then the object will probably be deallocated not too long after its real lifetime ends. If an object is allocated deep on the stack, however, the object may easily be retained in the stack long after it is dead. As a result, stack allocation could create space leaks.

4.1 Improving Precision

One potential source of space leaks is placing an object deeper in the stack than is required. Our profiling data may seem to preclude this possibility, but recall that allocation depths are decided on a per-site basis: objects are allocated at the maximum depth required by any object allocated at the same allocation site. It is easy to find examples where this is overly pessimistic. For example, a program might use the String.concat function in many places. In some places, the result might be discarded almost immediately (e.g., it might be printed on the screen); in others, the string might be propagated a long way up the call stack. This problem can be avoided in several ways:

- When a method is called, the caller could pass an additional parameter indicating how deep to allocate the result in the stack. Tofte and Talpin's ML compiler implicitly passes regions to functions [17] and Gay and Aiken's region-based C compiler [12] allows programmers to explicitly pass regions to functions.
- Instead of passing an extra parameter, create specialized versions of methods for specific stack depths. Obviously this approach must be used sparingly to avoid code-size-explosion.

Table 2 examines the potential benefits of more accurate lifetime prediction using greater contextual information. The table illustrates heap occupancy as a time-space product using no context and using the list of return addresses on the stack to provide additional context. In all cases, adding additional context information dramatically reduces heap occupancy. Interestingly, the figures for the jess benchmark initially drops but then rise again as allocation depth increases. This is caused by allocating objects with medium lifetimes in stack frames with long lifetimes and is addressed in the next section.

4.2 Better Liveness Knowledge

Another source of space leaks is that objects are only deallocated at the end of a method call, and so may be retained long after their last use. This problem is particularly severe in methods that contain loops: objects allocated on the first iteration might be dead, but cannot be deallocated until the method completes. This problem can be seen very clearly in the ray tracer benchmark. A ray tracer has a relatively simple structure: for each pixel in the output image, it generates a ray from the "camera lens" through the "film"; computes the color of the ray; and stores the color in the output image.

```
void traceRays() {
  for(int y=0; y<image.yres(); y++) {
    for(int x=0; x<image.xres(); x++) {
      Ray ray=camera.makeRay(x,y,image);
      Color c=scene.traceRay(ray, 0);
      image.set(x, y, c);
    }
  }
}
```

These nested loops are the source of a space leak: the Ray and Color objects obviously have to be allocated in the traceRays stack frame; most of the raytracer's execution time is spent in a single call to traceRays; so the Ray and Color objects will last for most of the raytracer's execution time.

To demonstrate that this is the source of a space leak, we modified traceRays by moving the inner loop into its own method allowing the short-lived Ray and Color objects to be deallocated after each iteration of the outer loop. This change eliminated a space leak which made the raytracer require space proportional to the number of pixels being traced.

```
void traceX(int y) {
  for(int x=0; x<image.xres(); x++) {
    Ray ray=camera.makeRay(x, y, image);
    Color c=scene.traceRay(ray, 0);
    image.set(x, y, c);
  }
}

void traceRays() {
  for(int y=0; y<image.yres(); y++) {
    this.traceX(y);
  }
}
```

We conclude that inserting extra deallocation points based on scope information could prove useful. Alternatively, for the specific problem of the ray tracer, a precise liveness analysis could detect that all of the Ray and Color objects that are allocated in the loop can reuse the same space.

Benchmark	Maximum depth in stack for allocations					
	0	1	2	3	5	10
jess	98%,60%	98%,46%	98%,40%	98%,50%	98%,50%	98%,62%
javac	99%,57%	99%,48%	99%,42%	99%,35%	99%,19%	99%,18%
rt	88%,74%	73%,36%	73%,48%	73%,47%	73%,46%	73%,45%
webster	100%,65%	100%,48%	100%,24%	100%,20%	100%,20%	100%,19%
jack	99%,62%	96%,45%	96%,46%	96%,41%	96%,41%	96%,41%
db	100%,57%	99%,46%	99%,38%	99%,35%	99%,34%	99%,32%

Table 2: Space-time product (bytes×instructions, or heap occupancy) for varying maximum allocation depth on the stack, relative to not using stack allocation. The first value assumes that every allocation at a given allocation site must be at the same stack depth; the second value assumes that objects allocated at the same allocation site could be allocated at different stack depths (which would reflect the availabilty of a maximal amount of calling context at each allocation site).

4.3 Further Extensions

Although both of the above solutions can significantly reduce space leaks caused by extending object lifetimes, it is unlikely that they will eliminate all space leaks. One way around this incompleteness is to use a hybrid scheme: rely on stack deallocation, but revert to garbage collection within stack frames when memory is running low, as proposed by Ruggieri and Murtagh [15].

Finally, in order to increase predictability for programmers, a static analyzer needs to be able to give feedback to the programmer concerning which objects can be stack-allocated. Otherwise, if a compiler uses a fragile algorithm for determining stack allocations, the programmer will not be able to predict program behavior at all.

5 Implementing The Stack

Until now, we have ignored two big problems in exploiting correlations between object and activation lifetimes:

- How do we allocate objects in the nth stack frame?
- How do we garbage collect stack frames?

5.1 Implementing Deep Allocation

It is easy to allocate objects in the topmost stack frame by incrementing the stack pointer, but it is hard to allocate objects deeper down the stack. The obvious way around this problem is for the caller to allocate space in its stack frame before calling methods that need to allocate objects in the caller's stack frame. Such a solution is restricted to cases where we can statically determine the maximum space required by the callee.

In cases where we cannot statically bound the space required by the callee, we need an overflow mechanism with which to "stack allocate" objects. One approach is to allocate objects in the heap and maintain a linked list of the objects "allocated" in each stack frame. This approach is simple to implement, but it increases the size of objects and the overhead of allocating and deallocating objects: popping a stack frame is no longer a constant time operation.

A more efficient (but more complex) approach is to allocate objects from buffers of contiguous storage. If objects with the same lifetime are allocated in a list of large contiguous blocks, we can deallocate many objects in a single operation: popping a stack frame is a near constant-time operation.

Reserving a large block of contiguous storage for every stack frame is expensive: we expect that some stack frames will have very few objects allocated in them. One way to avoid this problem is to recognize that we only need to be able to allocate into the top m stack frames if the maximum allocation depth is m. This observation makes it possible to maintain a ring of buffers such that objects for the nth stack frame are stored in the same buffer as the $n + m$th stack frame.

5.2 Garbage collecting stack frames

It might seem that objects stored in "stack frames" using this scheme could be garbage collected as easily as objects in the heap. After all, the buffers consist of contiguous blocks of memory and can be resized at will. There are two complications [15]:

1. Our motivation for allocating stack objects in contiguous blocks of memory is to allow near constant-time deallocation. Therefore, the garbage collector must compact objects in the stack frames and must preserve the relative order of the stack frames.
2. Our criteria for deciding how deep to allocate an object is *liveness*: when is an object last *used*? Because the compiler would allocate an object in a stack frame based on liveness, but the garbage collector would trace objects based on reachability, it

is possible to have a situation where a live object contains a pointer to a dead object. For example, a long-lived object (which is allocated deep in the stack) could contain a pointer to a short-lived object (which is allocated near the top of the stack). This scenario would cause a problem for the garbage collector, since it would dereference a dangling pointer. We are considering two possible solutions: we could use reachability to decide where to allocate objects; or we could generate code to null out dangling pointers.

6 Conclusions

Many researchers [2, 14] have argued about the performance tradeoffs between using the heap and using the stack. Jones and Muchnick [13] were among the first authors to examine the language and performance implications of stack vs. heap allocation. Other authors have shown how hybrid heap-stack strategies for allocating both continuations [8] and data structures [4] can improve performance. Our work is most closely related to Aiken et al.'s work [1] which describes an ML compiler with implicit region operations and to Gay and Aiken's work [12] which describes a compiler for a C-like language with explicit region operations. Our work differs in that we try to provide implicit region allocation in an object-oriented imperative language.

We are investigating deep stack allocation algorithms in Java in order to make it feasible to achieve more predictable memory behavior for systems programming. We view the combination of performance and predictability as the primary reasons for exploring this optimization. Previous research has explored different aspects of more predictable garbage collection, but none have looked at real systems programs in a language as widespread as Java. As our data shows, deep stack allocation is a promising means to reducing heap allocation. We have implemented a fast and simple analysis that detects feasible stack allocation sites. This algorithm performs acceptably on one of our benchmarks (a ray tracer), but not very well on the other benchmarks. Our ongoing work is to extend our algorithm to handle the other benchmarks.

More generally, we would like to extend our work into operating systems, which have even more stringent requirements on predictability. For example, device drivers run very close to the hardware: they must often run with guarantees about when GC will not run. Having predictable memory behavior is an absolute necessity for writing low-level systems code in high-level languages.

Acknowledgements

We would like to thank Godmar Back for initiating this project, Jay Lepreau for supporting this project, Sean McDirmid for suggestions and advice and the anonymous reviewers for valuable comments on this paper.

References

[1] A. Aiken, M. Fähndrich, and R. Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 1995.

[2] A. W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, 1987.

[3] H. G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, Apr. 1978.

[4] H. G. Baker. CONS should not CONS its arguments, or, a lazy alloc is a smart alloc. *ACM Sigplan Notices*, 27(3):24–34, Mar. 1992.

[5] H. G. Baker. The treadmill: Real-time garbage collection without motion sickness. *ACM Sigplan Notices*, 27(3):66–70, March 1992.

[6] C. Chambers and D. Ungar. "Customization: Optimizing Compiler Technology for SELF, A Dynamically-Typed Object-Oriented Programming Language". In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 146–160, Portland, OR, June 1989.

[7] D. R. Chase. Safety considerations for storage allocation optimizations. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 1–10, Atlanta, GA, June 1988.

[8] W. D. Clinger, A. H. Hartheimer, and E. M. Ost. Implementation strategies for continuations. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, pages 124–131, Snowbird, UT, July 1988.

[9] S. Dieckmann and U. Hölzle. A study of the allocation behavior of the SPECjvm98 java benchmarks. Technical Report TRCS-98-33, University of California at Santa Barbara, December 1998.

[10] J. Dolby. Automatic inline allocation of objects. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 7–17, Las Vegas, NV, June 1997.

[11] S. Dorward, R. Pike, D. L. Presotto, D. Ritchie, H. Trickey, and P. Winterbottom. The Inferno operating system. *Bell Labs Technical Journal*, 2(1):5–18, Winter 1997.

[12] D. Gay and A. Aiken. Memory management with explicit regions. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 313–323, Montreal, Canada, June 1998.

[13] N. D. Jones and S. S. Muchnick. Binding time optimization in programming languges: Some thoughts toward the design of an ideal language. In *Proceedings of the 3rd Symposium on Principles of Programming Languages*, pages 77–94, Atlanta, GA, Jan. 1976.

[14] J. S. Miller and G. J. Rozas. Garbage collection is fast, but a stack is faster. A.I. Memo 1462, MIT Artificial Intelligence Laboratory, Cambridge, MA, Mar. 1994.

[15] C. Ruggieri and T. 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 1988.

[16] M. Serrano and M. Feeley. Storage use analysis and its applications. In *Proceedings of the 1996 International Conference on Functional Programming*, pages 50–61, 1996.

[17] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *Proceedings from the 21st annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1994.

[18] Transvirtual Technologies Inc. http://www.transvirtual.com/.

[19] P. Tullmann and J. Lepreau. Nested Java processes: OS structure for mobile code. In *Proc. of the Eighth ACM SIGOPS European Workshop*, pages 111–117, Sintra, Portugal, Sept. 1998.

A Analysis

Our analysis first determines a conservative approximation to the set of all methods that can possibly be called assuming that a program begins execution at a specific main method. It then statically analyzes and annotates each of the methods. These annotations can be used to make decisions at runtime as to whether objects can be allocated on the stack.

Each method is annotated with four lists of byte code addresses: invocations of NEW that are stack-allocatable, invocations of NEW that are partially stack-allocatable, call-sites that are stack-friendly, and call-sites that are partially stack-friendly:

- An object is *stack-allocatable* if the method that creates it does not store it in a static variable, store it in another object, throw it, return it, or pass it to another method that causes one of the above to happen.
- An object is *partially stack-allocatable* if it would be stack-allocatable but for the fact that the creating method returns the object as a result.
- A call-site is *stack-friendly* if the value returned by the called method is not stored in a static variable, stored in another object, thrown, returned, or passed to another method that causes one of these things to happen.
- A call-site is *partially stack-friendly* if it would be stack-friendly but for the fact that the value returned by the called method is itself returned by the current method.

At any time in the execution of a program, there will be n activation records corresponding to methods M_1 (at the base of the stack) through M_n (at the top of the stack), where M_i invoked M_{i+1} at call-site C_i, for $i = 1 \ldots n-1$. Then:

- An object created in M_n can be allocated in the activation record of M_n if that object is stack-allocatable.
- An object created in M_n can be allocated in the activation record of M_j if the object is partially stack-allocatable, the call-sites $C_{j+1} \ldots C_{n-1}$ are partially stack-friendly, and the call-site C_j is stack-friendly.

There are three logical phases to the algorithm: call graph determination, an intraprocedural flow analysis, and an interprocedural analysis.

A.1 Call Graph Determination

In its first phase, the algorithm finds the smallest set M of methods and C of classes such that

- `main` $\in M$
- If $m \in M$ and m creates a new c, then $c \in C$.
- If $m_1 \in M$ and m_1 calls a static or special method m_2, then $m_2 \in M$.
- If $m_1 \in M$, $c_1 \in C$, m_1 invokes a virtual $c_2.m_2$, and c_1 is a subclass of c_2, and $c_1.m_2$ would be dynamically bound to m_3, then $m_3 \in M$.
- If $m_1 \in M$, $c_1 \in C$, m_1 invokes an interface method $c_2.m_2$, c_1 implements the interface c_2, and $c_1.m$ would be dynamically bound to m_3, then m_3 ∈M.
- If $c \in C$ and $c.finalize$ would be dynamically bound to m, then $m \in M$.
- If $m \in M$ and m references a class c, then c.`<clinit>` $\in M$.

A.2 Intraprocedural Analysis

In its second phase, the algorithm does intra-procedural flow analysis on all methods in the set M. It determines the disposition of three kinds of objects: objects that are passed into the method as parameters, objects that are returned back to the method as results, and objects that are created locally.

For each such object in a method, the flow analysis:

- Determines whether the object can be stored in a static variable, stored in another object, or thrown as an exception.
- Determines whether the object can be returned as a result.
- Determines whether the object can be passed as a parameter. For each call-site at which the object can be passed, the flow analysis also determines which formal parameter(s) the object can be bound to.

A.3 Interprocedural Analysis

In its third phase, the algorithm associates two boolean values, partial and full, with each object from the second phase. Initially, for each object o in every method m,

- o.partial is true if the object is not stored in a static variable, stored in another object, passed as a parameter, or thrown within m.
- o.full is true if o.partial is true and the object is not returned from m.

The analyzer iterates over all methods until fixpoint is reached. Specifically,

- o.partial = o.partial AND for every place p that o can be passed, (p.full is true OR p.partial is true and x.partial is true).
- o.full = o.full AND for every place p that o can be passed, (p.full is true OR p.partial is true and x.full is true).

where x is the object that corresponds to the return value of p's method.

The output of the analyzer is as follows: a NEW object o is stack-allocatable if o.full is true; a NEW object o is partially stack-allocatable if o.partial is true; a call-site with return object o is stack-friendly if o.full is true; and a call-site with return object o is partially stack-friendly if o.partial is true.