Pretenuring Based on Escape Analysis

Maria Jump
mjump@cs.utexas.edu

Ben Hardekopf
benh@cs.utexas.edu

ABSTRACT

Our hypothesis is that escape analysis can estimate lifetime information for dynamically allocated objects. We then use this information to pretenure those objects that have long lifetimes. This technique avoids the cost incurred by a generational copying collector for copying long-lived objects from the nursery into an older generation. This approach is completely new – all past work on pretenuring has involved profiling; our approach instead employs static analysis.

1. INTRODUCTION

Garbage collection is a technique that automatically reclaims unreachable memory. It increases programmer productivity and code modularity by eliminating the need for often error-prone explicit memory management. The cost of this technique is a measurable amount of performance and memory overhead. Reducing this overhead is an active research area [2, 3, 7, 8].

The generational copying collector is a widely adopted type of garbage collector. This collector partitions the heap into a nursery and some number of older generations [14]. The memory allocator allocates all objects into the nursery, and the garbage collector collects the nursery whenever it becomes full. Collection consists of: (i) identifying root pointers; (ii) identifying live objects, defined as those objects which are transitively reachable from the root pointers; (iii) copying the live objects into an older generation; (iv) reclaiming the space vacated by the dead and copied objects. The garbage collector collects the older generations whenever they become full as a result of the copied objects. The rationale for this division is the *weak generational hypothesis*, which states that "newly-created objects have a much lower survival rate than older objects" [11]. Therefore most objects in the nursery will have died before the garbage collector collects the nursery, and hence do not need to be copied. A large part of the cost of generational copying collectors is copying into an older generation those objects that did *not* die [17].

Technical Report TR-03-48
Department of Computer Science
The University of Texas at Austin
Austin, TX 78712
.

A recent technique that has emerged for reducing the cost of generational copying collectors is *pretenuring*. The insight behind this technique is that if the compiler can insert code to tell the allocator to allocate long-lived objects directly into an older generation without going through the nursery, then the garbage collector will not have to pay the cost of copying those objects when it collects the nursery. In order to implement this optimization we must be able to identify long-lived objects. Previous work in pretenuring relies on profiling to provide this information [3, 6, 10, 18, 19]. Profiling requires that the optimizer run the application being optimized multiple times and gather statistics on observed object lifetimes.

The goal of our work is to enable pretenuring based on *escape analysis*, a static analysis technique that determines whether the lifetime of an object exceeds its static scope. Escape analysis is used for optimizations such as stack allocation and synchronization elimination. However its ability to help predict the lifespan of an object has not been explored in connection with pretenuring.

We implement escape analysis in the Jikes RVM dynamic optimizing compiler. The compiler uses the escape analysis and several heuristics to determine object age, categorizing the objects as short-lived, long-lived, or immortal. The compiler also uses this categorization to provide the memory allocator with pretenuring advice. We measure the effectiveness of this advice and the impact it has on memory usage and program execution time for a range of benchmarks using an Appel-style generational copying collector. We compare our results with the results generated by previous work on profile-driven pretenuring [3].

In Section 2, we provide a description of escape analysis and briefly outline an escape analysis algorithm. We also describe various known optimizations that are done using escape analysis. Section 3 describes four different heuristics that estimate object age based on the results of escape analysis. In Section 4, we describe the methodology used. Section 5 describes the results of this preliminary report. Finally, in Sections 6 and 7, we conclude and present future research directions.

2. ESCAPE ANALYSIS

There are two ways an object can escape. An object o *escapes* a method m if the lifetime of o exceeds the runtime of m. If it does not escape, then o is *captured* by m. Similarly, an object o escapes its allocating thread T if o is accessed by another thread other than T in its lifetime; otherwise o is captured by T.

Simple escape analysis determines for each object whether or not it escapes the method in which it was allocated. More extensive analysis establishes a mapping from each object to its capturing method, if such a method exists. There are a wide variety of algorithms for escape analysis in the literature [5, 7, 9, 15, 16, 20, 21]. For this work, we focus on the algorithm given by Whaley and Rinard [21]. This algorithm has several attractive features which influenced our decision – the analysis is interprocedural; each method can be analyzed independently of its callers; a method can be analyzed without any methods that it calls; and analysis results from the skipped methods can be integrated into the analysis at a later time to give more precise information. These features make the algorithm ideal for use in dynamic compilation.

2.1 An Algorithm for Escape Analysis

This section briefly outlines the algorithm described by Whaley and Rinard [21]. More details can be found in the referenced document, including detailed pseudo-code.

The basis of the analysis is an abstraction called a *points-to escape graph*. The graph nodes represent objects; edges represent references between objects. The abstraction also contains information about which objects escape to other methods or other threads. The graph maintains a distinction between objects and references from unanalyzed sections of the code and objects and references that are entirely within the analyzed sections of code. This distinction makes clear those objects and references for which complete information is available and those for which it is not. The analysis can compose results from related methods, resulting in a more precise points-to escape graph.

The intraprocedural analysis uses a dataflow analysis to construct the points-to escape graph for a particular method. It begins by constructing a graph for the first statement of the method, then propagates the points-to escape graph through the control-flow graph using the appropriate transfer functions. The transfer function for the exit statement of the method generates the final analysis results. The algorithm then uses a reachability analysis to determine all those objects in the graph reachable from objects that are known to exceed the method runtime (e.g. method parameters, static class variables, method return values) and labels those objects as escaping. All other objects are labeled as captured. The analysis saves the resulting points-to escape graph which invoking call sites use.

When the analysis encounters a call site, it chooses between skipping the method call (and possibly incorporating that method's analysis at a later time) or using interprocedureal analysis to incorporate that method's analysis immediately. The interprocedural analysis takes the points-to escape graphs from the current method and the set of methods that may be invoked by any method calls and composes them using a mapping function to produce a new points-to escape graph. The resulting graph contains more precise information about which objects escape and where they are captured.

2.2 Escape Analysis Enabled Optimizations

Stack Allocation is an optimization that has been closely tied to escape analysis [5, 7, 8, 9, 21]. The premise behind this optimization is that it is cheaper to allocate objects on the stack than on the heap. The end of the method invocation automatically reclaims memory allocated on the stack reducing garbage collector overhead. Escape analysis determines whether it is safe to allocate an object on the stack – if an object escapes a method, then its memory cannot be reclaimed at the end on the method and it must go on the heap.

The main disadvantage of stack allocation is that it may increase memory consumption by retaining objects on the stack after they have become unreachable. This strategy not only wastes the memory required by the object itself, but also for any objects in the heap that are reachable from the object on the stack (garbage collectors conservatively consider anything on the stack to be alive when doing reachability analysis for live objects in the heap). Stack allocation must also be done conservatively in the presence of loops and recursion because of the danger of stack overflow. Experiments with stack allocation have shown that it does provide an execution speedup. However the speedup comes almost solely from improving data locality and not from any benefit to the garbage collector [4]. Most stack frames have a lifetime shorter than the period between collections, which means that the objects on the stack also have short lifetimes. If they had been allocated in the heap, they still would have died before the next collection and wouldn't have been copied by the collector; therefore putting objects on the stack had virtually no effect on the performance of the collector.

Synchronization Elimination is another optimization that has been extensively studied with relation to escape analysis [5, 7, 16, 21]. This optimization is most relevant to Java applications. Java assumes that all objects are potentially accessible by multiple threads, and protects access to each object via synchronization routines. Escape analysis can be used to identify objects which are never accessed by multiple threads, and the synchronization routines can be omitted for those objects.

Exploding Objects refers to replacing an object by a set of local variables, one variable per field in the object being exploded eliminating memory overhead associated with the object. This optimization can be done for any stackable object o if it is possible to inline all methods that take o as an argument and all uses of o are only to read and write fields of o [8]. Escape analysis is used to identify stackable objects.

Liveness Accuracy refers to identifying the root pointers that the garbage collector uses to determine which objects on the heap are still alive. The collector must be conservative in identifying root pointers in order to ensure that live objects are not prematurely reclaimed. Because of this conservative identification, the set of root pointers used may include pointers that are actually dead, leading to the collector copying dead objects because they were incorrectly labeled as live. Hirzel et al. show that using interprocedural analysis to improve the accuracy of identifying live root pointers can lead to large gains in performance [12]. Escape analysis has the potential to be used for this type of analysis. However a preliminary scan of the literature has not found any reference to this possibility presenting an area for future work.

3. PRETENURING ADVICE

Our central hypothesis is that the information provided by escape analysis is useful in determining object lifetimes, and

hence can furnish accurate pretenuring advice. We can also integrate other information, such as the depth of a procedure in the call graph, to increase the precision of our analysis. There are three types of advice we can give based on object classification:

- an object is *short-lived* if it will die before the next nursery collection;
- an object is *long-lived* if it would survive the next nursery collection; and
- an object is *immortal* if it dies more than halfway between its time of birth and the end of the program.

The allocator should allocate short-lived objects into the nursery, long-lived objects into an older generation, and immortal objects into a special partition that is never collected.

When furnishing advice, we can be either conservative or aggressive. Conservative advice labels objects as short-lived by default, and only promotes objects to long-lived or immortal status if there is a very good reason to do so. Aggressive advice labels objects as long-lived or immortal by default, and labels an object as short-lived only if there is a good reason to believe it will not survive long. Conservative advice tends to leave long-lived and immortal objects in the nursery where they will be copied by the collector; aggressive advice minimizes copying but tends to waste space by labeling short-lived objects as long-lived or immortal because the older space is collected less often.

The information provided by escape analysis for an object o can be one or more of the following statements:

- o is captured by its allocating method,
- o is captured by a set of methods M,
- o escapes into a global variable, and
- o escapes the allocating thread.

Since escape analysis is static, categorizing an object is equivalent to categorizing its allocation site – i.e. all objects allocated at the same site will be given the same advice. While individual objects can only be captured by a single method, objects created by a single allocation site may be captured by a set of methods; the elements of that set are all possible methods which may capture an object allocated at that site.

3.1 Global and Thread Escape Heuristics

Hirzel et al. show that there is a strong correlation between long lifetimes and objects that escape into global variables or escape their allocating thread [13]. Usually most of these objects are immortal, however there are some applications where this correlation does not hold. An aggressive approach can be used to label all objects which escape into global variables or escape their allocating thread as immortal. A more conservative approach can be used to label all such objects as long-lived.

3.2 Escape Distance Heuristic

Typically, stack frames have a relatively short lifetime. It follows that objects allocated on the stack (i.e. objects that do not escape their allocating method) also have short lifetimes. Hirzel et al. show that there is a correlation between long lifetimes and objects that escape their allocating method [13]. These insights are behind this heuristic.

For an object o which is captured by a method $m \in M$, define the *escape distance* $ED(o)$ as the length of the path between m and o's allocating method in the application call graph.

Since we can determine statically only that o is captured by a method in M, but not which method, we must calculate the lengths of the paths from the allocating method to all the methods in M. There may be a number of possible paths between the methods in the call graph, and there is no way to determine statically which path is taken for any particular object. We can estimate the actual escape distance of o using either the longest or shortest possible path. Using the longest possible path gives us an aggressive estimate, which we label $ED_a(o)$. Using the shortest possible path gives us a conservative estimate, which we label $ED_c(o)$. We define our heuristic in terms of the actual escape distance; which estimate to use when applying the heuristic is implementation dependent. Our heuristic also uses two threshold values, T_l and T_i, where $T_l \leq T_i$. The heuristic is defined as:

For each object o, if $T_l \leq ED(o) < T_i$, then label o as long-lived. If $ED(o) \geq T_i$, then label o as immortal. Otherwise label o as short-lived.

Increasing the threshold values make the heuristic more conservative. Decreasing the threshold values and including the depth of the call graph below the allocating method in the measure of escape distance make the heuristic more aggressive.

3.3 Age Estimation Heuristic

Object age is usually calculated in terms of bytes allocated. We can estimate the age of an object o captured by method $m \in M$ by estimating the number of bytes allocated from the time o was allocated to the time it is last used in m.

Again we can determine statically only that o is captured by a method in M, but not which method. Therefore we must calculate the age using each method in M. Since there may be a number of paths through the control flow graph that the application could take between the object's birth-point and each possible death-point, we have to either make a conservative estimate $age_c(o)$ by taking the lowest possible value, or make an aggressive estimate $age_a(o)$ by taking the highest possible value. We have threshold values T_a and T_b, where $T_a \leq T_b$. The heuristic is very similar to the escape distance heuristic:

For each object o, if $T_a \leq age(o) < T_b$, then label o as long-lived. If $age(o) \geq T_b$, then label o as immortal. Otherwise label o as short-lived.

3.4 Connectivity Heuristic

In Hirzel et al., the authors also found that there was a strong correlation in lifetimes between objects that were either directly linked or were part of the same strongly connected component in the application's points-to graph – they all tended to die at the same time [13]. We can use this result to help refine the escape distance and age estimation heuristics defined above. Given a set of objects for which we have generated pretenuring advice, all of which are directly linked or strongly connected, this heuristic states that we should give the same advice to all of them. We can either conservatively downgrade the lifetime advice of all the objects to that of the shortest-lived, or aggressively upgrade the lifetime advice of all the objects to that of the longest-lived.

Configuration	Leaves	Thread	Global
000	nursery	nursery	nursery
001	nursery	nursery	older
002	nursery	nursery	immortal
010	nursery	older	nursery
020	nursery	immortal	nursery
100	older	nursery	nursery
111	older	older	older

Table 1: Per-Configuration Mapping of Method of Escape to Allocation Advice

4. METHODOLOGY

Escape analysis allows us to identify allocation sites that may produce long-lived objects and to modify the site to allocate directly into less frequently collected regions. Our escape analysis identifies three ways an object can escape: *global*, *thread*, and *leaves*. An object that escapes *global* does so through a reference to a static class object. An object that escapes *thread* escapes through a thread. An object that *leaves* escapes in some other way (e.g. being passed to another method). We use this escaping information to make allocation decisions for a particular site. Since we generate per-allocation-site advice, we directly manipulate the intermediate representation to use the advice.

In this preliminary work on escape analysis, we implement only the intraprocedural analysis as described by Whaley and Rinard [21]. We use the escaping information to pretenure object based on the findings of Hirzel, et al. [13]. Table 1 describes these experiments. For each method of escape, we experiment with pretenuring objects into both the older and immortal spaces. Additionally, we experiment with pretenuring all escaping objects.

4.1 The Jikes RVM and the GCTk

We implement escape analysis in the Jikes RVM. The Jikes RVM is a high-performance VM written in Java that includes an aggressive optimizing compiler [1]. Additionally we use the GCTk, a garbage collector toolkit for the Jikes RVM [2]. It is an efficient and flexible platform for garbage collector experimentation that exploits the object orientation of Java and the JVM-in-Java property of the Jikes RVM. We use the Appel-style generational collector that uses a well-tuned and fast address-order write barrier and includes an uncollected region (for immortal objects) implemented for Blackburn et al. [3].

4.2 Experimental Setting

We perform our experimental timing runs on a Macintosh PowerPC G4 with a 933 MHz processor, 32K on-chip L1 data and instruction caches, 256KB unified L2 cache, 1MB L3 off-chip cache, and 512MB of memory running PPC Linux 2.4.19.

We use benchmarks from JVM98 and SPEC2000, choosing those that vigorously exercise the garbage collector (see Table 2). We run each benchmark on a range of heap sizes, ranging from the smallest one in which the program completes up to three times that size. We execute the benchmark five times for each configuration and pick the best execution time (i.e., the one least disturbed by other effects in the system).

Benchmark	Live	Alloc	Alloc/Live
jess	5,485,280	511,317,988	93
javac	12,068,700	647,267,620	54
jack	5,810,536	562,055,988	97
pseudojbb	30,024,524	620,019,384	21

Table 2: Benchmark Characteristics: (Live) is maximum live size in bytes, (Alloc) is total allocation in bytes.

5. RESULTS

In this section we examine the quality of the advice generated by escape analysis and the execution speedup resulting from following that advice.

5.1 Quality of Advice

We define the quality of the advice similarly to Blackburn et al. [3] where

- an object is *short-lived* if it will die before the next nursery collection;
- an object is *long-lived* if it would survive the next nursery collection; and
- an object is *immortal* if it dies more than halfway between its time of birth and the end of the program.

We examine exact object lifetimes (subscripted with o) and per-site (subscripted with s) decisions for each object to establish a level of error in the per site decisions. This division defines nine decision pairs that are further categorized:

- *good advice* allocates long-lived objects into longer-lived regions, but not too long-lived ($< i_o, i_s >$, $< l_o, l_s >$, $< i_o, l_s >$);
- *neutral advice* allocates objects into the nursery ($< s_o, s_s >$, $< l_o, s_s >$, $< i_o, s_s >$); and
- *bad advice* allocates objects into longer-lived regions than appropriate ($< i_o, l_s >$, $< l_o, i_s >$, $< s_o, i_s >$).

Table 3 summarizes the level of error in our classifications. It shows that pretenuring the small number global escaping objects into the older generation (0.13% good vs 0.02% bad) is better than pretenuring them into the immortal generation (0.06% good vs. 0.09% bad). Objects that are identified as thread-escaping results in as much good advice as bad when pretenured into the older generation and overwhelmingly bad advice when pretenuring into the immortal space. Pretenuring objects that leave generate as much bad advice (4.82%) as good (3.26%).

In Table 4, we summarize the quality of the advice given per benchmark. It describes how many objects were allocated into the correct generation as a percentage of the total number of objects allocated. It shows pretenuring objects that escape globally is a good idea though it may be better to make these objects long-lived rather than immortal. However, since the number of globally-escaping object is so small compared to the overall number of objects, it is insignificant. Additionally, it shows that those objects that our implementation identifies as thread-escaping are not necessarily long lived which is contrary to the work done by Hirzel et al. [13]. We believe that this is an artifact of our implementation which uses a very aggressive method for identifying

Configuration	% good %			% neutral %			% bad %		
	$< i_o, i_s >$	$< l_o, l_s >$	$< i_o, l_s >$	$< s_o, s_s >$	$< l_o, s_s >$	$< i_o, s_s >$	$< l_o, i_s >$	$< s_o, i_s >$	$< s_o, l_s >$
000	0	0	0	78.73	6.81	14.47	0	0	0
001	0	.07	.06	78.70	6.74	14.41	0	0	.02
002	.06	0	0	78.70	6.74	14.41	.07	.02	0
010	0	.83	.44	77.55	5.98	14.03	0	0	1.17
020	.44	0	0	77.55	5.98	14.03	.83	1.17	0
100	0	1.88	1.38	73.91	4.93	13.08	0	0	4.82
111	0	1.93	1.43	73.90	4.87	13.04	0	0	4.83

Table 3: Per-Configuration Pretenuring Decisions Accuracy (by object over all benchmarks)

	_202_jess	_213_javac	_228_jack	pseudojbb
Min	20.02%	31.64%	19.45%	20.17%
Max	23.36%	37.86%	25.20%	30.17%
Mean	21.60%	33.43%	22.50%	25.52%

Table 5: Memory Overhead per Benchmark

thread-escaping objects designed more for synchronization elimination (for which it is conservative) than for pretenuring. Finally, we show that it is too aggressive to pretenure everything that escapes because it increases the number of full-heap collections.

5.2 Analysis Overhead

We examine the overhead of our implementation of escape analysis in terms of overall runtime and additional memory requirements to represent the points-to escape graph. As expected, there is an increase in both runtime and memory due to the overhead of creating the points-to escape graph. Memory usage increases are shown in table 5 and total execution time in figure 1. Normalized execution times are shown in figure 2.

Figure 3 shows the total execution without the compilation overhead. It shows that pretenuring globally-escaping objects has negligible affect on overall execution time due primarily to the extremely small number of globally escaping objects in the benchmarks we ran. Additionally, it shows pretenuring objects that escape *thread* or *leaves* increase the overall execution time. This is not unexpected due to the quality of the advice we generate.

6. CONCLUSIONS

Escape analysis in its naïve implementation is expensive in both space and time.

The quality of advice shows that it is a good idea to pretenure objects that escape globally. Generally it is better to put globally-escaping objects in the long-lived space. However, since globally-escaping objects represent such a small percentage of total, pretenuring them into the immortal space does not significantly effect memory usage or total execution time. No conclusions can be made about the lifetimes of thread-escaping objects since our implementation is so aggressive. Its aggressiveness results from its adaptation from a non-dataflow sensitive lock-removal algorithm. Finally, we conclude it is too aggressive to pretenure all escaping objects. This is not surprising as it leads to both the older geneneration filling up too quickly and to an increase in the number of full-heap collections.

7. FUTURE WORK

We plan to examine ways in which our naïve implementation can be improved. First we plan to examine how we can improve our implementation to make it more efficient in both space and time. How much of the analysis overhead can be eliminated? Can adding interprocedural analysis giving us more precise information from which to make our decisions? Additionally can we tweak escape analysis algorithm for the pretenuring case? For example, can *escape distance* provide a smarter mechanism for determining lifetimes? Do either *hot-spot analysis* and *function cloning* provide more opportunities for pretenuring at a reduced cost?

We can also ask questions about using this type of algorithm to create *build-time* advice for libraries and shared code. Then an application in the JIT could still benefit from escape analysis without suffering the expense.

If the cost of performing the analysis remains too high, we plan to amortize the cost by looking for other optimizations that can share the information generated from escape analysis. Examples of this include traditional optimizations such as synchronization elimination as well as new work such as write-barrier elimination [22].

8. ACKNOWLEDGMENTS

We would like to thank Kathryn McKinley for pointing us to this work and for her valuable advice along the way; Steve Blackburn for his experitise and assistance in understanding the Jikes RVM and previous pretenuring work; and Matthew Hertz for providing valuable information regarding the handling of traces.

9. REFERENCES

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The jalapeno virtual machine. *IBM System Journal*, 39(1):211–238, February 2000.

[2] S. M. Blackburn, R. Jones, K. S. McKinley, and J. E. B. Moss. Beltway: Getting around garbage collection gridlock. In *Proceeding of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 153–164. ACM Press, 2002.

[3] S. M. Blackburn, S. Singhai, M. Hertz, K. S. McKinely, and J. E. B. Moss. Pretenuring for Java. In *Proceedings of the OOPSLA '01 conference on Object*

	jess				javac			
	%good%	%neutral%	%bad%	%correct%	%good%	%neutral%	%bad%	%correct%
000	0	100	0	95.58	0	100	0	89.64
001	2.21E-5	99.99	7.37E-6	95.58	1.68E-5	99.72	.28	89.36
002	2.21E-5	99.99	7.37E-6	95.58	1.68E-5	99.72	.28	89.36
010	.17	85.99	13.83	81.86	.53	96.93	2.55	87.53
020	.05	85.99	13.95	81.80	.09	96.93	2.98	87.19
100	.20	66.18	33.62	62.08	5.54	81.25	13.22	80.65
111	.20	66.18	33.62	62.08	5.54	81.17	13.29	80.58
	jack				**pseudojbb**			
	%good%	%neutral%	%bad%	%correct%	%good%	%neutral%	%bad%	%correct%
000	0	100	0	91.91	0	100	0	95.41
001	1.25E-3	99.99	7.25E-6	91.91	2.73E-5	99.99	0	95.41
002	9.42E-5	99.99	1.17E-3	91.91	1.64E-5	99.99	1.09E-5	95.41
010	.32	99.65	.04	92.16	.03	99.49	.48	94.95
020	.02	99.65	.33	91.89	.01	99.49	.50	94.94
100	.58	89.07	10.35	82.10	1.0	88.17	10.82	85.18
111	.58	89.07	10.35	82.10	1.0	88.17	10.82	85.18

Table 4: Per-Benchmark Pretenuring Decisions Accuracy (by object)

Oriented Programming Systems Languages and Applications, pages 342–352. ACM Press, 2001.

[4] B. Blanchet. Escape analysis: correctness proof, implementation and experimental results. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 25–37. ACM Press, 1998.

[5] B. Blanchet. Escape analysis for object-oriented languages: Application to Java. *ACM SIGPLAN Notices*, 34(10):20–34, 1999.

[6] P. Cheng, R. Harper, and P. Lee. Generational stack collection and profile-driven pretenuring. In *Proceedings of the ACM SIGPLAN '98 conference on Programming language design and implementation*, pages 162–173. ACM Press, 1998.

[7] J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Escape analysis for Java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 1–19, 1999.

[8] D. Gay and B. Steensgaard. Stack allocating objects in Java (extended abstract). Work done at Microsoft Research.

[9] D. Gay and B. Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *Compiler Construction, 9th International Conference (CC 2000)*, pages 82–93, 2000.

[10] T. L. Harris. Dynamic adaptive pre-tenuring. In *Proceedings of the second international symposium on Memory management*, pages 127–136. ACM Press, 2000.

[11] B. Hayes. Using key object opportunism to collect old objects. In *Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*, pages 33–46. ACM Press, 1991.

[12] M. Hirzel, A. Diwan, and A. Hosking. On the usefulness of liveness for garbage collection and leak detection. *To appear in ACM Transactions on Programming Languages and Systems*, ??:??, 2002.

[13] M. Hirzel, J. Henkel, A. Diwan, and M. Hind. Understanding the connectivity of heap objects. In *Proceedings of the Third International Symposium on Memory Management*, pages 36–49. ACM Press, 2002.

[14] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, 1996.

[15] F. Qian and L. Hendren. An adaptive, region-based allocator for Java. In *Proceedings of the Third International Symposium on Memory Management*, pages 127–138. ACM Press, 2002.

[16] A. Salcianu and M. Rinard. Pointer and escape analysis for multithreaded programs. In *Proceedings of the eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 12–23. ACM Press, 2001.

[17] D. Tarditi and A. Diwan. The full cost of a generational garbage collection implementation. In *Proceedings of the OOPSLA '93 Workshop on Memory Management and Garbage Collection*, pages 1–8, September 1993.

[18] D. Ungar and F. Jackson. Tenuring policies for generation-based storage reclamation. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 1–17. ACM Press, 1988.

[19] D. Ungar and F. Jackson. An adaptive tenuring policy for generation scavengers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 14(1):1–27, 1992.

[20] F. Vivien and M. Rinard. Incrementalized pointer and escape analysis. In *Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation*, pages 35–46. ACM Press, 2001.

[21] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 187–206. ACM Press, 1999.